Spécification de la mémoire dans un langage synchrone flot de données doté de tableaux de taille statique

Baptiste Pauget

PSL UNIVERSITÉ PARIS

Ansys

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à l'École Normale Supérieure de Paris

Memory Specification in a Data-flow Synchronous Language with Statically Sized Arrays

Spécification de la mémoire dans un langage synchrone flot de données doté de tableaux de taille statique

Soutenue par
Baptiste Pauget
Le 8 Décembre 2023

École doctorale n°386
Sciences Mathématiques de Paris Centre

Spécialité
Informatique

Composition du jury :

Jean-Louis Giavitto IRCAM	*Président*
Albert Cohen Google	*Rapporteur*
François Pottier Inria	*Rapporteur*
Yamine Ait-Ameur Toulouse INP	*Examinateur*
Laure Gonnord Grenoble INP	*Examinatrice*
Alain Girault Inria, Univ. Grenoble-Alpes	*Invité*
Jean-Louis Colaço Ansys	*Co-directeur*
Marc Pouzet École normale supérieure	*Co-directeur*

ENS | PSL

Remerciements

Ce manuscrit clôt trois années de recherches partagées entre l'équipe PARKAS de l'Inria et l'équipe Core d'Ansys, une aventure qui doit beaucoup à Marc et Jean-Louis. Leur entente et leur complicité ont permis un contexte d'échange particulièrement favorable au déroulement de cette thèse. Nos nombreuses et régulières discussions m'ont guidées dans l'exploration des différents sujets. Grâce au climat de confiance qu'ils ont su instaurer, je n'ai jamais hésité à leur présenter mes idées, même les moins abouties. Je leur en suis profondément reconnaissant.

Merci à Yamine Ait Ameur, Albert Cohen, Jean-Louis Giavitto, Alain Girault, Laure Gonnord et François Pottier qui ont accepté de participer à mon jury. La qualité de mon manuscrit doit beaucoup aux conseils et remarques avisés de François et d'Albert. Leurs regards extérieurs m'ont permis de considérer le résultat de ses trois années avec davantage de recul. Je les remercie également de s'être adaptés au calendrier resserré de cette fin de thèse. Un grand merci à Timothy Bourke pour sa relecture minutieuse tant sur le fond que sur la qualité de la langue. Merci à Laure Gonnord et Alain Girault pour le suivi de ma thèse et leurs conseils lors de nos rencontres.

Cette thèse CIFRE n'aurait pu voir le jour sans l'accueil de Günther Siegel au sein de ses équipes, je l'en remercie. Ce contexte partagé entre le monde universitaire et celui de l'industrie a été pour moi une source de motivation. Merci également à Bruno Pagano pour l'intérêt porté à mes travaux. Nos échanges m'ont aidé à ancrer mes recherches dans les perspectives d'évolution de Scade à long terme. Je lui suis également redevable pour sa persévérance et son efficacité à faire avancer les démarches administratives grâce auxquelles je peux poursuivre dans l'équipe Core d'Ansys.

Je remercie l'équipe PARKAS qui m'a si bien accueilli lors de mes stages de recherche puis pendant ces trois années de thèse. À chacun de mes retours au -1 de l'aile Rataud, j'ai eu l'impression d'en être parti la veille. L'atmosphère chaleureuse et bienveillante de l'équipe doit beaucoup à Guillaume, Marc et Tim, toujours prêts à s'interrompre pour discuter d'un point épineux ou d'une idée nouvelle. Je n'oublierai pas les rudiments de baby-foot acquis grâce à Paul, Basile et Grégoire, ni les dégustations de thés et infusions à toute heure. Merci de m'avoir accompagné lors de sorties vélo et montagne plus ou moins réalistes et parsemées d'imprévues.

Merci aux membres de l'équipe Core d'Ansys, Adnan, Bruno, François, Hervé, Jean-Louis, Loïc, Olivier et Yannick. J'ai pu dès mon arrivée suivre les développements en cours et mesurer la variété des sujets traités autour de Scade. Cette diversité et la facilité des échanges au sein de l'équipe m'ont donné envie de la rejoindre.

J'ai eu l'occasion de partager mes recherches lors de séminaires et de colloques : Synchron, les JFLAs, le séminaire du DI. Ces moments plus ou moins formels ont été particulièrement constructifs, aussi bien pour l'avancée de mes travaux que pour le mûrissement de mon projet professionnel. Je pense en particulier à Alain, qui a tenté — en vain — de me faire succomber aux charmes de la recherche académique. Je tiens à remercier Laurence Bourcier et Valérie Mongiat qui ont grandement facilité les démarches indispensables à ma participation à ces événements.

Je remercie Xavier Crégut et Aurélie Hurault de m'avoir accordé leur confiance pour leurs TPs et TDs à l'ENSEEIHT. Cette expérience d'enseignement m'a beaucoup plu et m'a permis de m'ouvrir à des sujets autres que celui de ma thèse.

Mes passages à Paris n'auraient été aussi agréables sans l'accueil toujours renouvelé d'Alexandre, Jean-François, Antoine, Marc et Lucile. Ces repas et soirées passés ensemble m'ont offert des pauses bienvenues dans le rythme soutenu de ces séjours parisiens. Merci également à Raphaël pour nos midis au pot. Du côté de Toulouse, merci à MuLun et Aldo pour nos sorties gastronomiques, sportives et musicales qui m'ont aussi donné l'occasion de souffler.

Je remercie mes parents qui m'ont accompagné et soutenu pendant ces longues années d'études. Merci à mes sœurs, Solène et Lucile, qui m'ont toujours rappelé l'importance de s'aérer. Nos excursions en montagne m'ont permis de garder la tête froide même lorsque que mes travaux piétinaient. Pendant ces trois ans, j'ai pu m'appuyer sur le soutien sans faille de Justine. Nos aventures à la découverte du sud-ouest comptent parmi les plus beaux moments de ces années et ont été indispensables à mon équilibre. Merci pour sa patience pendant les longues semaines où je lui ai préféré la rédaction de ce manuscrit. Ce n'est que partie remise.

Résumé

SCADE est un langage de programmation synchrone utilisé depuis la fin des années 1990 pour concevoir et implémenter des systèmes critiques embarqués tels ceux que l'on trouve dans l'aviation. Cette criticité requiert la formalisation (i) des programmes, afin de garantir qu'ils s'exécutent sans erreurs ou retards pendant une durée arbitraire et (ii) des outils de développement, en particulier le compilateur, pour s'assurer de la préservation des propriétés des modèles lors de la génération de code. Ces activités reposent sur une description flot de données des modèles, inspirée des schémas-blocs, dont la sémantique s'exprime par des fonctions de suites, ainsi que d'une compilation précisément documentée.

SCADE descend de LUSTRE. La version 6 a introduit des évolutions majeures, dans les pas de LUCID SYNCHRONE, dont les automates hiérarchiques et les tableaux. Pour ces derniers, leur taille est connue et vérifiée statiquement afin d'assurer un temps d'exécution et une mémoire bornés. L'utilisation croissante des tableaux de SCADE pour des opérations intensives a révélé plusieurs axes d'amélioration: verbosité des modèles, manque de contrôle de la compilation ou description peu commode de certains algorithmes.

Dans cette thèse, ces trois aspects ont été étudiés à l'aide d'un prototype de compilateur. (i) Nous avons développé un système de types *a la* Hindley-Milner spécifiant les tailles sous la forme de polynômes multivariés. Cette proposition permet de vérifier et d'inférer la plupart des tailles de manière modulaire. (ii) Nous avons exploré une méthode de compilation alternative, fondée sur un *langage déclaratif conscient de la mémoire*. Il vise à concilier le style flot de données avec une spécification précise des emplacements mémoire. La description modulaire des tailles en est un élément clé. (iii) Enfin, nous proposons une construction d'itération inspirée de SISAL qui complète les itérateurs actuels. En traitant les tableaux comme des suites finies, elle donne accès aux constructions séquentielles de SCADE (automates) lors d'itérations. De plus, elle permet de décrire de manière déclarative des implémentations efficaces d'algorithmes comme la décomposition de Cholesky. Cette compilation contrôlable est un premier pas nécessaire pour la compilation vers des GPUs.

Systèmes embarqués ⋆ Programmation synchrone ⋆ Tableaux ⋆ Systèmes de types ⋆ Compilation

Abstract

The synchronous programming language SCADE has been used since the end of the 1990s to design safety critical embedded systems such as those found in avionics. This context requires to formally reason about (i) the programs, to ensure that they run for an arbitrary long time without errors or missed deadlines and (ii) the tools, in particular the compiler, to give high confidence in the preservation of model properties through the code generation process. These activities build on a data-flow description of models, inspired by block diagrams, that enjoys a formal semantics in terms of streams and a well-documented compilation process.

SCADE stems from LUSTRE. The SCADE6 version introduced major evolutions following LUCID SYNCHRONE, notably hierarchical automata and arrays. For the latter, sizes are statically known and checked, so as to fulfill the bounded resources and execution time constraints. The increasing use of SCADE arrays for data-intensive computations has revealed areas for improvement: concision of the models, control of the generated code and idiomatic description of array computations.

Using a prototype implementation of a compiler, the present work investigates these three aspects. (i) We designed a Hindley-Milner-like type system for representing sizes with multivariate polynomials. This proposal allows to check and infer most sizes in a modular way. (ii) We explored an alternative compilation process based on a *memory-aware declarative language*. It aims at reconciling the data-flow style with a precise specification of memory locations. The underlying memory model builds on our modular description of sizes. (iii) Last, we propose an iteration construct inspired by SISAL that supplements the available iterators. By viewing arrays as finite streams, iteration can benefit from the sequential constructs of SCADE, e.g., automata. Moreover, it allows declarative descriptions of efficient algorithm implementations such as the Cholesky decomposition. This controllable compilation process is a mandatory step toward code generation for GPUs.

Embedded systems ★ Synchronous programming ★ Arrays ★ Type systems ★ Compilation

Présentation

Ce chapitre propose un résumé étendu du contexte de cette thèse et ses contributions.

This chapter gives an extended summary of the context of this thesis and its contributions.

Introduction

Le language synchrone Scade est utilisé pour concevoir et implémenter du logiciel embarqué pour des systèmes critiques. Ces systèmes interagissent en permanence avec leur environnement, sur lequel ils ont un contrôle *indirect*. Par exemple, un régulateur de vitesse fait coïncider la vitesse mesurée avec un objectif défini en ajustant la puissance fournie par le moteur.

Ces logiciels, appelés *contrôleurs*, contiennent une part croissante de calculs intensifs (traitement du signal, vision artificielle, IA, *etc*). Ces traitements utilisent en particulier les tableaux. Bien que l'introduction de tableaux dans les langages de programmation a été largement étudiée, leur utilisation dans Scade pose des questions spécifiques au domaine d'application du langage.

Dans cette section, nous présentons rapidement le contexte et les différents axes de recherches liés à ce travail. Les travaux connexes sont détaillés plus longuement dans le chapitre 1.

Langages synchrones Scade [CPP17] hérite des principes et du style de Lustre [Hal+91]. Le temps y est modélisé par une succession d'instants logiques et les durées de calcul et de communication sont supposées nulles en théorie et en pratique compatibles avec les temps de réaction nécessaires au contrôle. Scade est un langage purement fonctionnel. Les fonctions dont les arguments sont des suites, applées *nœuds*, sont constituées de définitions de suites mutuellement récursives. Les valeurs de base sont des suites, appelées *flots*, qui peuvent être combinées point-à-point avec des opérations *combinatoires* (calculs scalaires, *etc*) ou à l'aide d'opérateurs *temporels* tels que le retard unitaire `pre` ou l'initialisation `->`. Ce paradigme de programmation bénéficie d'une spécification mathématique simple, où les nœuds sont modélisés par des fonctions de suites.

Les programmes synchrones sont compilés vers du code impératif qui définit l'*état initial* et la *fonction de transition*. Chaque nœud est représenté par une fonction qui, étant donné un *état* et les valeurs courantes des entrées, calcule les valeurs des sorties et un nouvel état. Les langages synchrones tels que Scade garantissent des propriétés fortes sur le code généré. Lorsque la compilation réussit, la fonction de transistion produite est déterministe et termine, avec une mémoire et un pire temps d'exécution bornés. Ces deux bornes sont cruciales dans le cas de systèmes *temps-réels critiques* pour lesquels une erreur à l'exécution peut avoir des conséquences dramatiques. La première assure que les ressources en mémoire ne viendront pas à manquer. La seconde permet de garantir la réactivité des programmes, dont la stabilité du système peut dépendre. Pour assurer ces propriétés, Scade impose une restriction importante : la mémoire doit être connue à la compilation. En particulier, la taille des tableaux est statique et le code généré ne requiert pas de gestion dynamique de la mémoire (allocation, recyclage).

La description purement fonctionnelle en termes de suites est le principal atout de Scade. Elle permet une vérification des propriétés des programmes réactifs plus aisée qu'avec une forme impérative. Pour que les résultats de ces analyses donnent des garanties sur l'exécution, la correction du compilateur est essentielle. Dans le cadre de Scade, qui vise les plus hauts niveaux de certification (la DO178C level A pour l'avionique), la confiance en le générateur de code repose sur un processus de compilation éprouvé et précisément décrit [Bie+08]. Plus récemment, le langage Vélus [Bou+17] a été formalisé dans l'assistant de preuve Coq et son compilateur, qui suit les mêmes procédés, dispose d'une preuve formelle vérifiée par ordinateur de la préservation de la sémantique.

Correction des tableaux Les tableaux représentent des collections de données homogènes, ce qui permet des traitements uniformes sur leurs éléments. Les tableaux sont des objets composites. Ils ne peuvent pas être manipulés par des opérations atomiques des processeurs. Dans des langages bas-niveau tels que C, les opérations sur les tableaux sont donc implémentées de manière *extensionnelles*, c'est-à-dire élément par élément. Les indices des éléments sont calculés à l'exécution, ce qui introduit une difficulté quant à la correction des programmes : les accès aux tableaux doivent respecter leurs bornes.

La correction des accès peut être garantie en protégeant les accès aux tableaux par des branchements qui déclenchent un mécanisme de secours — valeur par défaut, levée d'exception, *etc*— en cas de non respect des bornes. Cette solution possède deux inconvénients. (i) Elle ajoute une complexité importante aux accès. (ii) Elle requiert du code pour la gestion des erreurs qui est inutile si le programme est correct.

	Extensionnelle	**Intentionnelle**
Application point-à-point	$\forall i.\ B[i] = f\ A[i]$	$B = \texttt{map}\ f\ A$
Retournement de tableaux	$\forall i.\ B[i] = A[n-1-i]$	$B = \texttt{rev}\ A$

Les langages déclaratifs proposent une vision plus abstraite. Les tableaux y sont vus comme des entités manipulées avec des opérations *intentionnelles* qui évitent les calculs explicites d'indices. La table ci-dessus illustre les deux types de descriptions pour des opérations simples. Par exemple, le retournement d'un tableau `A` s'écrit en extension par un calcul d'indice — `A[n-i-1]` — alors que la version intentionnelle utilise une opération dédiée — **`rev`** — qui représente cette transformation.

Les opérations intentionnelles fournissent des schémas d'accès valides sous condition de certaines égalités des tailles des tableaux. Elles réduisent le problème du respect des bornes à une propriété de cohérence des tailles. Cela simplifie la vérification. Si elle est dynamique, il suffit de vérifier les propriétés de tailles avant d'effectuer les opérations avec des accès non protégés. Pour une vérification statique, établir des égalités de tailles est plus simple que d'assurer des inégalités entre indices et tailles.

Plusieurs extensions de systèmes de types *a la* ML ont été proposées pour assurer statiquement des propriétés plus fortes que celles de la forme des données. Ces extensions ont été appliquées en particulier à la cohérence des tailles. Deux principes se distinguent. (i) Les types indexés [Zen97] définissent des familles de types à l'aide d'*indices*. L'exemple emblématique est l'ajout de dimensions physiques aux scalaires [Ken94]. (ii) Les types raffinés [XP99] restreignent les domaines des types à l'aide de prédicats. Dans chaque cas, le choix du langage d'indice ou de prédicat est l'élément clé de l'équilibre entre l'expressivité du système de types et les possibilités de vérification et d'inférence.

Compilation de calculs intensifs L'efficacité de l'implémentation de calculs intensifs dépend principalement de deux aspects interdépendants: l'*allocation* (stockage des valeurs) et le *parallélisme* (exécution simultanée de calculs indépendants). Les performances des accès en mémoire ne sont pas uniformes. Cela est dû aux spécificités du matériel comme les caches ou les accès coalescents des GPUs. Pour une opération intensive fixée, le choix de l'organisation des calculs est crucial. Par exemple, les résultats intermédiaires peuvent être stockés ou recalculés, les itérations peuvent être divisées en plusieurs parties (tuilage), *etc.* Ces transformations peuvent augmenter le parallélisme ou améliorer l'utilisation des caches.

Une implémentation impérative optimisée est difficilement compréhensible et vérifiable. De plus, elle est spécifique à l'architecture visée. Il est alors nécessaire de réécrire tout ou partie du programme afin de l'adapter à un autre environnement d'exécution. Pour pallier ce manque de modularité, le langage Halide [Rag+13], conçu pour programmer les algorithmes de traitement d'images, propose de séparer l'algorithme de l'implémentation. Le premier est décrit dans un style purement fonctionnel, lisible mais non optimisé. La seconde est constituée de directives qui guident l'application de transformations du programme spécifiques à la cible, qui préservent sa sémantique, afin de l'optimiser.

Dans les langages où les tableaux sont manipulés de manière intentionnelle, les transformations d'algorithmes sont effectuées sur les primitives de tableaux, appelées *combinateurs* dans le cadre de Futhark [Hen+17]. Ces règles de réécriture, comme $\texttt{map}\ g \circ \texttt{map}\ gf = \texttt{map}\ (g \circ f)$, permettent

notamment d'éliminer les nombreux résultats intermédiaires inhérents au style intentionnel qui décrit des opérations au niveau des tableaux plutôt que des éléments.

Les tableaux en Scade Les tableaux de Scade sont manipulés à l'aide de plusieurs constructions. (i) Les opérations intentionnelles sont composées d'itérateurs du second ordre (`map`, `fold`) et d'opérateurs du premier ordre (`reverse`, `transpose`, `concat`). (ii) Les manipulations extensionnelles sont des accès statiques (vérifiés à la compilation) et dynamiques (protégés à l'exécution) ainsi qu'une mise-à-jour fonctionnelle dont la sémnantique est celle d'une copie avec modification.

Les nœuds de Scade peuvent être paramétrés par des types et des tailles. Cependant, les points d'entrée des programmes doivent être *monomorphes*, c'est-à-dire non paramétrés. Cette restriction permet une vérification simple des tailles et des accès statiques aux tableaux, car leurs valeurs sont connues à l'*instanciation*, lors de l'utilisation, des nœuds polymorphes. Cependant, cette vérification n'est pas modulaire. Les erreurs de tailles ne sont pas détectées à la définition de ces opérateurs.

La qualité du code généré dépend principalement de l'élimination des copies introduites par la compilation, en particulier pour les tableaux et leurs mises-à-jour fonctionnelles. Éviter une copie permet de diminuer à la fois l'empreinte mémoire et le pire temps d'exécution. Cependant, ces transformations reposent sur des optimisations fragiles que le programmeur contrôle mal et qui, lorsqu'elles échouent, conduisent à du code peu efficace.

Contributions La nature purement fonctionnelle de Scade permet de rapprocher les programmes de fonctions de suites mathématiques. Cependant, du côté du langage, les constructions de haut niveau telles que l'ordre supérieur ou la récursivité ne peuvent être utilisées afin de préserver les garanties d'exécution bornée, Du côté de la compilation, la description purement fonctionnelle des programmes laisse au compilateur l'entière responsabilité du choix d'implémentation, en particulier sur la gestion de la mémoire et l'ordre des calculs. Ainsi, Scade et son compilateur assurent une mémoire et un temps d'exécution bornés mais ne donnent aucun moyen de contrôler ces bornes.

Nous proposons un langage déclaratif appelé MADL, pour *Memory-Aware Declarative Language*, dans lequel l'allocation mémoire est dirigée par les programmes. Ce langage pourrait servir d'intermédiaire dans la compilation de Scade, mais cet aspect n'a pas été étudié. Nous présentons ici MADL comme un langage autonome, doté de ses propres vérifications statiques. Il repose sur les éléments suivants, que nous détaillons plus longuement dans la suite de cette présentation.

- *Un système de types avec tailles.* Nous proposons une extension des systèmes de types *a la* Hindley-Milner pour décrire des tailles. Les tailles sont des polynômes multivariés à coefficients entiers. Cela permet d'exprimer une part importante des calculs intensifs de tableaux tout en gardant la vérification décidable.

- *Une description statique de la mémoire.* Notre modèle de la mémoire est simple : elle contient des données pures — scalaires, tuples et tableaux — disjointes deux-à-deux. Les emplacements mémoire sont décrits à l'aide de *foncteurs de projection*. Ils représentent des indirections statiques qui peuvent être composées et comparées. Ces foncteurs permettent de construire des vues d'une même donnée avec des structures différentes.

- *Un langage déclaratif avec une discipline d'emplacements.* MADL est un langage purement fonctionnel doté d'un système de types pour les emplacements. Comme pour les types, les emplacements sont contraints pas les constructions du langage. En particulier, les opérations de tableaux comme la concaténation introduisent des contraintes d'emplacement qui assurent que ces opérations ne nécessiteront pas de calculs dans le code généré.

- *Une génération de code prévisible.* La compilation repose sur la distinction fondamentale entre les expressions *structurelles* et *calculatoires* de MADL. Les premières permettent de décrire de manière fonctionnelle des transformations de la structure des données. Elles ne nécessitent pas de construire leur résultat en mémoire car les valeurs y sont déjà stockées. Les secondes sont les calculs du programmes. Elles sont traduites dans le langage cible (C) de manière directe, sans nécessiter d'optimisation. Les accès à la mémoire sont construits à partir des emplacements.

- *Une itération basée sur les suites.* Nous proposons une construction `forward` qui traite les tableaux comme un flot plus rapide. Itérer revient alors à effectuer plusieurs pas de

la fonction de transition à chaque cycle. Elle permet d'utiliser les opérateurs temporels des langages synchrones, comme les automates en SCADE, lors d'itérations sur les tableaux. Cette construction offre une plus grande souplesse pour la programmation de parcours multidimensionnels et peut être étendue pour définir récursivement des tableaux, en permettant d'utiliser les éléments déjà construits pour calculer l'élément courant.

1 Décrire statiquement la mémoire

Le contrôle de la mémoire s'appuie sur une description statique de l'allocation. Celle-ci associe à chacun des flots du programme un *emplacement*. Parmi les objectifs de ces emplacements, ils doivent permettre de décrire des projections et des opérations de tableaux telles que la concaténation. Pour cela, il est nécessaire de connaître la taille des données manipulées, en particulier celle des tableaux.

1.1 Types et tailles

Nous proposons une extension pour un système de types *a la* Hindley-Milner. Celle-ci est indépendante des caractéristiques de SCADE (synchrone, premier ordre). Elle pourrait être utilisée dans un cadre plus large. Pour cette raison, le chapitre 2 présente notre système de types sur noyau de langage fonctionnel, d'ordre supérieur, sans traits synchrones. Ce langage est volontairement réduit à son minimum : un λ-calcul avec des déclarations locales (`let`) enrichi de quelques constructions spécifiques aux tailles.

Raffinements, sous-typage et polymorphisme Notre extension consiste en deux *raffinements* du type `int`, paramétrés par des tailles, notées η. Le type *taille* — $\texttt{<}\eta\texttt{>}$ — représente le singleton $\{\eta\}$ et le type *indice* — $\texttt{[}\eta\texttt{]}$ — désigne l'intervalle $[\![0, \eta - 1]\!]$. Le type $\texttt{[}\eta\texttt{]}\,\tau$ des tableaux de taille η d'éléments de type τ est un raccourci pour le type $\texttt{[}\eta\texttt{]} \to \tau$: les tableaux sont vus comme des fonctions d'un intervalle sur le type des éléments.

Les tailles — η — sont des polynômes multivariés à coefficients entiers, dont les variables formelles sont notées ν, μ, κ, *etc.* Ce langage est suffisant pour décrire des opérations telles que la concaténation ou l'échantillonnage. De plus, la forme normale des polynômes — une somme pondérée de produits variables — autorise une comparaison symbolique des tailles.

Comme pour les types, les déclarations sont rendues génériques en taille à l'aide de polymorphisme. Les opérateurs de concaténation et d'application point-à-point ont les schémas de type suivants. Le deuxième argument de l'opérateur `map`, de type $\texttt{<}\nu\texttt{>}$, permet de définir la taille du tableau attendu.

$$\texttt{concat} : \forall \nu \cdot \mu \cdot \alpha.\ \texttt{[}\nu\texttt{]}\,\alpha \to \texttt{[}\mu\texttt{]}\,\alpha \to \texttt{[}\nu + \mu\texttt{]}\,\alpha$$
$$\texttt{map} : \forall \nu \cdot \alpha \cdot \beta.\ (\alpha \to \beta) \to \texttt{<}\nu\texttt{>} \to \texttt{[}\nu\texttt{]}\,\alpha \to \texttt{[}\nu\texttt{]}\,\alpha$$

Notre système de types autorise un sous-typage (<:) rudimentaire. Les types raffinés sont comparables au type de base — $\texttt{<}\eta\texttt{>} <: \texttt{int}$ et $\texttt{[}\eta\texttt{]} <: \texttt{int}$ — mais ils sont incomparables entre eux, même lorsque les relations sont sémantiquement valides : $\texttt{[}\eta\texttt{]} \not<: \texttt{[}\eta + 1\texttt{]}$. Cette restriction permet la décidabilité de la correction des types en limitant les contraintes de tailles à des égalités au lieu d'inégalités. Contrairement à la plupart des systèmes de types avec sous-typage, le polymorphisme est *simple* : les variables généralisées ne sont soumises à aucune contrainte.

Les types singletons — $\texttt{<}\eta\texttt{>}$ — font le pont entre le monde des tailles et celui des expressions. Le langage fournit une *expression de taille*, dont la syntaxe est également $\texttt{<}\eta\texttt{>}$, qui permet d'utiliser une taille comme l'unique valeur de type $\texttt{<}\eta\texttt{>}$. Pour les fonctions, les paramètres de type $\texttt{<}\eta\texttt{>}$ introduisent des contraintes de tailles qui peuvent aider l'inférence des tailles. Ainsi, l'expression `map` f $\texttt{<}42\texttt{>}$ a pour type $\texttt{[}42\texttt{]} \to \texttt{[}42\texttt{]}$. Lorsqu'elles ne sont pas nécessaires, ces tailles peuvent être omises : `map` f $\texttt{<\_>}$.

Sémantique et propriétés Dans les langages dont le système de types autorise du sous-typage, la sémantique dépend des types. Par exemple, le terme $(\lambda x : \texttt{<}4\texttt{>}.\ x)\ \texttt{<}5\texttt{>}$ n'a pas de sémantique alors que l'expression $(\lambda x : \texttt{int}.\ x)\ \texttt{<}5\texttt{>}$ en a une. Les annotations de types peuvent contrôler l'existence de la sémantique. Dans notre proposition, l'utilisation de tailles comme expressions

a une conséquence supplémentaire : la sémantique *observable* des programmes — les résultats numériques — peut dépendre de la valuation des variables de taille.

Ce système de types ne permet pas l'existence de types principaux. Une déclaration peut être typée de plusieurs façons, avec des schémas de type non comparables, pour deux raisons. (i) L'expressivité du langage des tailles conduit à des contraintes — des polynômes nuls — dont l'ensemble des solutions ne peut être exprimé par une unique substitution des variables libres. Par exemple, la contrainte $\nu * \nu = 1$ possède deux solutions $\nu := 1$ and $\nu := -1$. (ii) Le polymorphisme simple impose de résoudre toutes les contraintes de sous-typage à la définition. Il faut donc choisir où sont propagés ou ignorés les raffinements, ce qui peut également conduire à des schémas de type incompatibles.

La première cause est intrinsèque à nos objectifs d'expressivité. La second découle de ceux de modularité. Le polymorphisme contraint, souvent utilisé avec du sous-typage, conduirait à reporter une partie de la résolution des tailles à l'instanciation. De plus, cette source de types non-principaux est limitée en restreignant le sous-typage, par exemple avec un type dédié pour les tableaux qui empêche la comparaison à celui des fonctions. Si elle n'a pas d'importance pour la vérification des types, l'absence de types principaux doit être prise en compte pour l'inférence.

Correction des programmes En SCADE, les tailles des tableaux sont connues à la compilation car l'opérateur principal ne peut être générique en taille ou en type. Cela permet une vérification facile et complète de la cohérence des tailles à l'instanciation, où les tailles prennent des valeurs concrètes. Cependant, cette stratégie de vérification n'est pas modulaire : les erreurs de tailles sont détectées à l'usage des opérateurs et non à leur définition.

Notre système de types remédie en partie à ce problème. Il assure la *cohérence* des tailles — les tailles qui doivent être égales le sont — mais n'impose rien quant à leur signe. Pour les tableaux, le type $[\eta]$ pour η négatif est vide ; un tableau de taille négative est donc vide. Cependant, certaines opérations intentionnelles, telles que l'échantillonnage, nécessitent que leurs paramètres de taille soient positifs afin que les schémas d'accès soient corrects. Comme pour SCADE, ces vérifications doivent être menées à l'instanciation.

Ce système de types permet toutefois d'éliminer de façon modulaire une part significative des contraintes de tailles, en particulier celles qui proviennent d'opérateurs simples (par exemple, `map`). De plus, l'intérêt principal de ce système de types ne réside pas dans la vérification. Il fournit également une information statique précieuse pour la compilation : la forme des données.

Inférence Dans sa version explicitement typée, le langage requiert la spécification des tailles dans les types et les expressions de taille ($<\nu>$). Ces informations peuvent être redondantes et encombrantes lorsque les tailles grossissent. Il est utile de les inférer. Les propriétés mentionnées ci-dessus sont alors cruciales. Du fait de l'absence de types principaux, l'inférence doit choisir entre plusieurs schémas de type incompatibles, lesquels peuvent conduire à des comportements différents puisque la sémantique observable dépend des tailles.

L'inférence doit rejeter les programmes ambigus, c'est-à-dire ceux qui possèdent plusieurs sémantiques observables. Pour cela, la résolution des contraintes de sous-typage ne doit éliminer aucune solution possible pour les tailles. En particulier, l'inférence ne doit pas introduire de contraintes de taille arbitraires.

Les algorithmes d'inférence pour des extensions de systèmes de types *a la* Hindley-Milner collectent les contraintes de type et les simplifient aux points d'unification (`let`) [AW93]. Nous suivons la même approche. Une fois rassemblées, les contraintes de sous-typage sont résolues en construisant une substitution de la manière suivante:

1. *Unification des types.* Les squelettes des types sont inférés en ignorant les raffinements, par unification du premier ordre.

2. *Sélection des raffinements.* Pour chaque occurrence du type `int` dans la substitution obtenue, les contraintes de sous-typages sont utilisées pour insérer les raffinements — singletons ou intervalles, indépendamment des tailles — lorsque cela est nécessaire, c'est-à-dire pour les variables en position négative (*ex.* $\alpha <: [\_]$).

3. *Résolution des tailles.* Les contraintes de tailles, qui proviennent d'égalités entre raffinements, sont résolues en préservant l'ensemble de solutions.

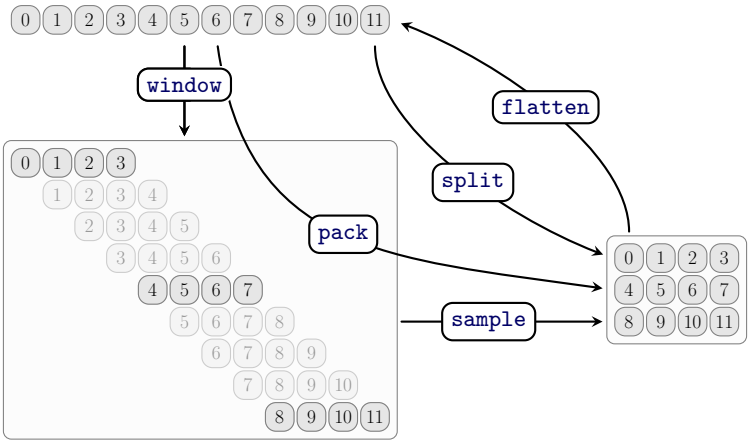


Figure 1: *Combinateurs de tableau du premier ordre*

4. *Propagation des raffinements.* Les occurrences du type `int` en position positive (*ex.* `[_]` $<:$ α) sont raffinées si les raffinements qui apparaissent dans les contraintes le permettent, en particulier s'ils contiennent les mêmes tailles.

Différer la propagation des raffinements après la résolution des tailles assure que le choix des raffinements n'introduit pas de contrainte de taille supplémentaire. De plus, nous proposons une stratégie *ad hoc* pour résoudre les contraintes de taille. Elle assure la préservation de l'ensemble des solutions. L'utilisation d'un solveur externe pour cette résolution semble délicate car la vérification de la généricité d'une solution n'est pas aisée.

Pour garantir l'unicité de la sémantique, il faut enfin s'assurer lors de la généralisation que les variables de taille libres apparaissent dans le type afin de leur donner une valeur à l'instanciation. Alors que les variables de type non contraintes sont uniquement le reflet de code mort, les variables de taille non contraintes peuvent conduire à une sémantique observable arbitraire.

Opérations de tableaux Plutôt que de surcharger les opérateurs arithmétiques pour les tailles et les indices, nous proposons d'introduire les calculs sur les indices par des opérations intentionnelles sur les tableaux. Elles complètent celles que l'on trouve actuellement en SCADE: `transpose`, `reverse` et `concat`. Ces primitives sont illustrées dans la Figure 1, et leurs schémas de type suivent.

$$\begin{aligned}
\texttt{window} &: \forall \iota\cdot\kappa\cdot\alpha.\ \texttt{<}\kappa\texttt{>} \to [\iota + \kappa - 1]\,\alpha \to [\iota]\,[\kappa]\,\alpha \\
\texttt{sample} &: \forall \iota\cdot\kappa\cdot\alpha.\ \texttt{<}\kappa\texttt{>} \to [\iota * \kappa - \kappa + 1]\,\alpha \to [\iota]\,\alpha \\
\texttt{split} &: \forall \iota\cdot\kappa\cdot\alpha.\ \texttt{<}\kappa\texttt{>} \to [\iota * \kappa]\,\alpha \to [\iota]\,[\kappa]\,\alpha \\
\texttt{flatten} &: \forall \iota\cdot\kappa\cdot\alpha.\ \texttt{<}\kappa\texttt{>} \to [\iota]\,[\kappa]\,\alpha \to [\iota * \kappa]\,\alpha \\
\texttt{pack} &: \forall \iota\cdot\kappa\cdot\delta\cdot\alpha.\ \texttt{<}\delta\texttt{>} \to [\iota * \delta - \delta + \kappa]\,\alpha \to [\iota]\,[\kappa]\,\alpha
\end{aligned}$$

L'opérateur `window` construit la matrice des sous-tableaux de taille κ d'un vecteur donné. L'opérateur `sample` sélectionne un élément sur κ. Le résultat doit contenir le premier et le dernier élément. Leur composition permet de définir un opérateur d'échantillonnage général, `pack`, qui extrait des sous-tableaux régulièrement espacés et couvrant les deux extrémités. Pour une longueur de sous-tableau égale au pas d'échantillonnage, cet opérateur transforme un vecteur en une matrice (opérateur `split`). Ce cas particulier est inversible, d'inverse `flatten`.

1.2 Emplacements mémoire

La connaissance des tailles des données permet d'envisager une description statique de la mémoire, présentée dans le chapitre 3. Notre proposition est construite sur deux restrictions de l'usage de la mémoire. (i) Les données stockées sont pures : elles ne contiennent pas d'indirections (pointeurs). (ii) La mémoire est un collection de segments typés indépendants.

Structures de données et projections La mémoire est typée. Chaque partie est associée à un type de données, constitué de scalaires — entier et booléen — et de structures de données composites — tuples et tableaux — dont les composants sont accédés à l'aide de projections.

- Les *tuples* — de type $\tau_1 * \ldots * \tau_n$ — rassemblent des données hétérogènes. Pour cette raison, ils sont accédés de manière statique uniquement. Les projections de tuple sont notées `(k/m)` ou `m` est l'arité du tuple et `k` est un entier entre 1 et `m`.
- Les *tableaux* — de type $[\eta]\tau$ — contiennent des données homogènes ce qui permet des accès dynamiques. Ces projections sont notées $[e<\eta]$ ou η est la taille du tableau. L'expression d'indice e est un polynôme qui peut contenir des tailles et des variables représentant des indices. Sa valeur doit être comprise entre 0 et $\eta - 1$.

Comme les parties de la mémoire, les projections sont typées. Par exemple, la projection `(1/2)` ne s'applique qu'à une paire. Cela permet d'assurer la cohérence des types des projections indépendamment des données accédées. Les structures de données peuvent être imbriquées. L'accès aux éléments se fait donc à l'aide d'une liste de projections.

Foncteurs de projection Certaines opérations de tableaux comme la transposition réorganisent les éléments d'une donnée composite sans modifier leur valeur. Elles ne nécessitent donc aucun calcul. Ces opérations sont modélisées par des transformations d'accès, appelées *foncteurs de projection*.

Les foncteurs de projection permettent de définir plusieurs *vues* structurées différemment d'une même valeur, par exemple, une matrice et sa transposée. Ces foncteurs sont typés suivant des règles simples qui s'appuient sur le type des projections. Écrits dans un style inspiré du filtrage de OCaml, les opérateurs de transposition et de retournement de tableau ainsi que la permutation d'une paire sont représentés par les foncteurs de projection suivants :

```
let transpose ν μ α = function
  | [j<μ] :: [i<ν] :: l_α → [i<ν] :: [j<μ] :: l_α

let reverse ν α = function
  | [i<ν] :: l_α → [ν - i - 1<ν] :: l_α

let swap α β = function
  | (1/2) :: l_β → (2/1) :: l_β
  | (2/2) :: l_α → (1/2) :: l_α
```

Ces foncteurs ne sont pas des programmes. Ils représentent des indirections statiques qui décrivent comment les accès aux éléments d'une vue sont calculés à partir des accès à une autre vue.

Les foncteurs ci-dessus sont paramétrés par des tailles ν, μ et des types α, β. Ils transforment une liste de projections en une autre. Dans chacun des cas des filtrages, l'identifiant l_α ou l_β représente une liste de projections pour le type α, qui est transmise dans la liste de projections produite. Le foncteur `transpose` échange les deux premières projections. Il transforme un accès à un tableau de type $[\mu][\nu]\alpha$ en un accès de type $[\nu][\mu]\alpha$. Il permet donc d'interpréter un tableau de type $[\nu][\mu]\alpha$ en un tableau de type $[\mu][\nu]\alpha$. Le foncteur `reverse` préserve le type de la projection, mais introduit un calcul pour l'indice de l'élément accédé. Le foncteur `swap` permet de voir une paire de type $\alpha * \beta$ comme une paire de type $\beta * \alpha$.

L'expressivité du langage des foncteurs de projection est limitée par celle du langage d'indices : des polynômes multivariés. Il est par exemple impossible de décrire une rotation. Comme pour les tailles, cette limitation permet des manipulations formelles.

La correction des foncteurs de projection dépend d'inégalités polynômiales qui ne peuvent pas être vérifiées en général. Par exemple, dans le cas de `reverse`, l'indice $\nu - i - 1$ doit être une valeur entre 0 et $\nu - 1$ lorsque i est un entier entre 0 et $\nu - 1$. Comme pour les calculs d'indices, les foncteurs de projection sont introduits dans le langage à l'aide d'un catalogue d'opérations de tableaux qui garantissent leur correction sous certaines hypothèses sur les tailles. Les foncteurs de projection fournissent une représentation extensionnelle unifiée d'une partie des opérations intentionnelles qui ne modifient que l'organisation apparente des données.

Opérations et propriétés La composition est la principale opération formelle sur les foncteurs de projection. Elle permet de construire des transformations complexes à partir d'un nombre restreint de primitives en assurant leur correction.

Dans le but de décrire des emplacements mémoire à l'aide de foncteurs de projection, plusieurs propriétés sont nécessaires, en particulier leur injectivité, ou l'existence d'un inverse. Comme pour

la correction, ces propriétés ne peuvent être établies en général. Elles sont attachées aux différentes primitives et les propriétés des compositions sont déduites de celles des foncteurs composés. Par exemple la composée d'une injection et d'une bijection est une injection.

Grâce à la forme normale des polynômes, l'égalité de foncteurs de projection est décidable, et de faible complexité. Ainsi, la composition de `reverse` avec lui-même est égale à l'identité, que la taille soit connue ou non. Il est en revanche impossible de déterminer si deux foncteurs permettant des vues d'une même donnée sont disjoints, c'est-à-dire si leurs images — l'ensemble des listes de projections produites — sont disjointes. Cette propriété est nécessaire à l'analyse des interférences entre données. Elle est approximée de manière conservative.

2 Spécifier la mémoire dans un langage déclaratif

Nous proposons un langage synchrone, déclaratif, appelé MADL, dans lequel l'utilisation de la mémoire est spécifiée. Afin qu'il reste purement fonctionnel, cette spécification ne joue aucun rôle dans la sémantique des programmes, qui n'est définie que par les dépendances de données. À chaque flot est associé un emplacement mémoire. Pour les définir, les constructions du langage introduisent des contraintes entre les emplacements de leurs arguments et résultats. Lorsqu'elles ont une solutions, ces contraintes définissent l'allocation.

2.1 Deux niveaux sémantiques

Le langage est présenté dans le chapitre 4. Bien que l'allocation ne joue pas de rôle sémantique, les principes de MADL sont construits sur l'existence d'une allocation adéquate. Ils reposent sur une distinction essentielle entre deux types de constructions. Cette séparation sert à la fois la sémantique fonctionnelle du langage et la spécification de l'allocation.

- Les expressions *structurelles* sont des artefacts de la définition déclarative. Elles mettent en relation des flots qui seront représentés en mémoire par les mêmes données. Il s'agit en particulier des projections, d'opérations comme la transposition, *etc.*

- Les expressions *calculatoires* sont les parties du programme qui nécessitent des calculs à l'exécution, comme les opérations arithmétiques. Ces constructions créent de nouvelles valeurs qui doivent être stockées en mémoire.

Les expressions structurelles sont utilisées dans plusieurs contextes. Parce qu'elles ne nécessitent aucun calcul, elles peuvent apparaître en partie gauche des équations (sous condition d'inversibilité). Elles servent également à spécifier les emplacements à l'aide d'annotations.

Éléments distinctifs de MADL La description de la mémoire nécessite également des constructions qui distinguent MADL de langages synchrones tels que Scade.

- *Une copie explicite.* En MADL, la duplication de données est explicite. Elle est représentée par une expression calculatoire dédiée dont les occurrences sont les seules expressions à générer des copies. En particulier, la construction de données structurées (tableaux, tuples) n'introduit pas de copies, les éléments doivent être calculés et stockés à leur emplacement dans le résultat.

- *Des annotations d'emplacement.* Comme pour les types et les tailles, les emplacements des flots peuvent être contraints à l'aide d'annotations. Ces annotations sont constituées d'expressions structurelles et de variables d'emplacements semblables aux variables de types que l'on trouve en OCaml.

- *Un état explicite.* Les opérateurs temporels — `pre`, `->` — des langages synchrones dérivés de Lustre opèrent au niveau des flots. Le compilateur se charge d'extraire un état et une fonction de transition pour implémenter les fonctions de flots. En MADL, les programmes spécifient directement des fonctions de transitions et leurs états. Un opérateur `last`, qui généralise celui de Scade, permet d'accéder à l'état. Il est initialisé par une construction `init` basée sur la mémoire : les flots qui partagent les mêmes données partagent également leur initialisation.

Sémantique hybride Afin de rendre compte du traitement de la mémoire dans MADL, nous proposons une sémantique opérationnelle *hybride* qui évalue les programmes à l'aide d'une mémoire qui stocke la valeur des flots. Pour donner une sémantique aux programmes sources, l'allocation, qui associe à chaque flot un emplacement mémoire, est quantifiée existentiellement. Cette mémoire est *déclarative*: elle assure le respect des dépendances de données. Un flot ne peut être lu que si son emplacement n'a pas été modifié depuis sa définition. Pour cela, la mémoire enregistre en plus des valeurs l'ensemble des variables qui y font références. L'évaluation suit les principes suivants:

- Les constructions structurelles introduisent des contraintes entre les emplacements de leurs arguments et leurs résultats. Elles ne modifient pas la mémoire.
- Les constructions calculatoires produisent de nouvelles valeurs qui sont écrites en mémoire. Ces écritures invalident les flots précédemment déclarés à l'emplacement modifié.

Sémantique purement fonctionnelle L'allocation n'est pas explicite dans les programmes MADL, les emplacements sont inférés (voir section 2.2). En suivant les principes de conception du langage, il est possible de donner une sémantique *purement fonctionnelle*, indépendante de la mémoire. Elle est utile pour concevoir des analyses indépendantes de l'allocation, par exemple pour la correction de l'initialisation. Cela permet de mener ces vérifications directement sur le programme source, sans avoir besoin du résultat de l'inférence des emplacements. Les diagnostiques qui en résultent sont plus pertinents car ils ne nécessitent pas de comprendre l'inférence d'emplacements. Comparé aux sémantiques fonctionnelles usuelles, plusieurs éléments nécessitent des traitements dédiés :

- Les opérations structurelles peuvent apparaître en partie gauche. Pour la sémantique hybride, ces constructions sont évaluées indirectement, en introduisant des contraintes d'emplacements. Pour une sémantique fonctionnelle, il faut inverser ces opérations.
- L'initialisation de l'état dépend des emplacements. Pour une sémantique purement fonctionnelle, cela nécessite de propager les valeurs d'initialisation à travers les expressions structurelles. Pour cette raison, les copies ne peuvent pas être ignorées, car elles définissent les limites de cette propagation.

2.2 Une discipline d'emplacements

La spécification de l'allocation est indirecte. Les emplacements des flots sont déduits de contraintes intrinsèques aux constructions du langage et des annotations d'emplacement. L'allocation est décrite comme un système de types, présenté dans le chapitre 6.

Emplacements et signatures Le langage d'emplacement est composé de deux constructions : (i) des variables et (ii) des vues construites à l'aide de foncteurs de projection. Les expressions sont décrites par des *signatures d'emplacement*, qui sont généralisées à l'aide de *polymorphisme* au niveau des opérateurs. Par exemple, l'opération de transposition est décrite par le *schéma d'emplacement* suivant :

$$\texttt{transpose}: \ \forall\, \nu\, \mu.\ \forall\, \alpha.\ \forall\, \delta : [\nu]\,[\mu]\,\alpha.\ \delta \longrightarrow \delta.\texttt{transpose}\ \ \nu\ \ \mu\ \ \alpha$$

Ce schéma se lit : pour toutes tailles ν et μ, pour tout type α et pour tout emplacement δ de type $[\nu]\,[\mu]\,\alpha$, l'opérateur `transpose` attend un argument stocké à l'emplacement δ et renvoie un résultat stocké comme une vue de l'emplacement δ par le foncteur de projection `transpose` ν μ α présenté dans la section 1.2.

La correction de l'allocation est définie par un système de règles de déduction qui assurent la cohérence des emplacements des flots. Comme les variables d'emplacement et les foncteurs de projection sont typés, la correction de l'allocation permet de déduire la correction du typage.

Contrairement aux types et aux tailles, le langage d'emplacement ne contient pas de construction de base, telles que les entiers pour les tailles. Les emplacements sont donc constitués d'une variable vue à travers une liste de foncteurs de projection. Ces variables sont quantifiées existentiellement ou généralisées. Les premières représentent les variables locales de l'opérateur alors que les secondes permettent de communiquer avec le contexte appelant (arguments, résultats, état).

Comme pour les types et les tailles, les opérateurs sont *instanciés*. À leur utilisation, les variables d'emplacement sont substituées par des emplacements de l'opérateur appelant. L'allocation d'un opérateur est donc constituée des emplacements des flots ainsi que de l'instanciation des opérateurs utilisés.

Expressivité des schémas d'emplacement Les limites d'expressivité du langage des foncteurs de projection imposent des contraintes sur la description de certaines opérations structurelles. Par exemple, l'opération de tableau `flatten`, qui transforme une matrice en vecteur, ne peut être décrite directement car elle nécessite, pour un indice du résultat, de calculer le quotient et le reste d'une division euclidienne. Cependant, l'opération inverse, nommée `split` peut être représentée par le foncteur de projection :

```
let split ν κ α = function
  | [i<ν] :: [j<κ] :: l_α → [i*κ+j<ν*κ] :: l_α
```

Cela permet de décrire l'opération `flatten` par le schéma d'emplacement suivant. Contrairement au cas de `transpose`, l'emplacement de l'argument est une vue — $\delta.$ `split` $\nu\ \kappa\ \alpha$ — alors que l'emplacement du résultat est une variable.

$$\texttt{flatten} : \forall \nu\, \kappa.\ \forall \alpha.\ \forall \delta : [\nu * \kappa]\alpha.\ \delta.\texttt{split}\ \nu\ \kappa\ \alpha \longrightarrow \delta$$

La situation est similaire pour la concaténation, où les emplacements des deux arguments sont les sous-tableaux gauche et droit du résultat. Dans ces deux cas, l'emplacement des entrées est déduit de l'emplacement de la sortie.

Inférence Comme pour les types et les tailles, les contraintes d'emplacement sont générées à l'aide de variables libres. L'inférence construit alors une substitution de ces variables d'emplacement qui satisfait les contraintes. Le typage de la mémoire permet de voir les emplacements comme des raffinements des types. Ainsi, une allocation qui spécifie un unique emplacement pour toutes les valeurs scalaires d'un même type serait trivialement correcte. Cependant, elle empêcherait l'ordonnancement de la plupart des programmes (voir section 2.3). Bien que l'allocation n'ait pas de contenu sémantique, son inférence ne doit pas introduire plus de partage d'emplacement que nécessaire, afin de préserver les possibilités d'ordonnancement.

Trouver les foncteurs de projection les plus généraux permettant de satisfaire les égalités d'emplacement est impossible en général. Pour cette raison, notre algorithme d'inférence se limite à composer les foncteurs introduits dans les contraintes. Le système à résoudre se ramène alors à un graphe orienté dont les sommets représentent les variables d'emplacement et les arêtes représentent les foncteurs de projection qui les lient. L'inférence ajoute des arêtes de façon à unifier les définitions multiples (chemins entrants) d'un même emplacement. Pour cela, les emplacements sont manipulés de deux façons :

- *Syntaxique.* Les emplacements sont décrits par une variable d'emplacement et une liste de foncteurs de projection. Cela permet d'unifier des emplacements dont l'une des listes de foncteurs est un préfixe de la seconde.
- *Symbolique.* Les contraintes restantes sont vérifiées en calculant les compositions de ces listes de foncteurs afin de s'assurer que leurs résultats coïncident.

S'il est facile de construire des programmes corrects sur lesquels l'inférence échoue, il est également aisé d'y ajouter des annotations d'emplacement pour guider la résolution des contraintes. Ce contexte favorable apparaît naturellement lors de la construction ou de la destruction d'une donnée structurée, par exemple avec une concaténation. La plupart des exemples écrits ne nécessitent pas d'annotation supplémentaire.

2.3 Une génération de code directe

La génération de code est présentée dans le chapitre 7. Le langage est conçu pour permettre une compilation sans optimisation. Une fois les types, les tailles et les emplacements inférés, la production de code impératif est scindée en deux étapes :

1. Les dépendances de données et les emplacements sont analysés afin de trouver un ordre correct pour les opérations contenues dans le programme.

2. Les opérateurs sont traduits vers le langage cible (C) en générant une fonction de transition et une fonction d'initialisation. La représentation des flots en mémoire est dérivée de leurs emplacements.

Ordonnancement Dans SCADE, l'ordonnancement des opérations du programme est dirigé uniquement par les dépendances de données. Pour MADL il faut aussi prendre en compte l'allocation. Le principe est le suivant : deux valeurs différentes qui partagent un même emplacement ne peuvent exister en même temps. Ce principe est similaire à celui de l'allocation de registres. À la différence, l'ordonnancement choisit l'ordre des opérations en fonction de l'allocation, et non l'inverse. Pour chaque nœud, un ordre cohérent est construit à l'aide des étapes suivantes.

1. *Assurer l'atomicité des opérations.* Les opérations sont analysées individuellement pour vérifier que les emplacements permettent leur implémentation. La condition principale repose sur l'injectivité des emplacements écrits. Pour que les composants d'une donnée structurée puissent exister simultanément, ils doivent être stockés à des emplacements disjoints.

   L'opération de copie introduit une contrainte supplémentaire. Les emplacements de la source et de la destination doivent être égaux ou disjoints. Il est par exemple impossible de transposer une matrice en place par une simple copie. Ces vérifications permettent de caractériser les opérations par un ensemble d'emplacements modifiés.

2. *Construire les interférences.* Les contraintes d'ordonnancement introduites par l'allocation sont décrites par des interférences dans le graphe du flot de données, où les dépendances sont étiquetées par un emplacement. Un nœud du graphe, qui représente une opération, *interfère* avec une arête — une dépendance — si les emplacements modifiés par l'opération intersectent l'emplacement de la dépendance. Dans ce cas, l'opération ne peut être effectuée entre la source et la cible de la dépendance car elle corromprait les données utilisées.

3. *Propager les dépendances.* La construction de l'ordonnancement complète l'ordre partiel défini par les dépendances de données, qui doivent être acyclique. L'idée est simple : si un nœud est ordonnancé avant la cible (resp. après la source) d'une dépendance avec laquelle il interfère, il doit alors être ordonnancé avant la source (resp. après la cible). L'interférence est alors *résolue*. Lorsqu'aucune interférence ne peut être éliminée par ce procédé, l'ordre d'une interférence restante est choisi arbitrairement avant de reprendre la propagation.

Les possibilités d'ordonnancement dépendent de la précision du calcul d'intersection pour les emplacements. Celui-ci repose sur l'analyse des foncteurs de projection, nécessairement approximative. En suivant une approche comparable à celle de l'inférence des emplacements, nous proposons une représentation à la fois syntaxique et symbolique des dépendances qui permet une approximation plus précise de l'intersection de foncteurs de projections.

Génération de code La production d'une fonction de transition consiste en deux étapes. (i) Allouer des variables locales pour représenter les variables d'emplacement quantifiées existentiellement et (ii) traduire les expressions calculatoires en instructions qui accèdent à la mémoire selon les emplacements déterminés par l'inférence. Les expressions structurelles ne construisent pas de valeurs en mémoire. Elles apparaissent dans le code généré à travers les accès à la mémoire, dont elles ont déterminé la structure.

Les calculs scalaires sont traduits de manière transparente. La compilation des copies demande plus de travail, elle ne peuvent pas toutes être implémentées par de simple `memcpy`. Pour des données composites dont l'emplacement est défini par une vue, par exemple, une matrice transposée, elles nécessitent de dupliquer les éléments un à un, lus selon la vue de la sources et écrits selon la vue de la cible. Les copies peuvent donc introduire des boucles et de multiples instructions.

Polymorphisme Notre chaîne de compilation est constituée d'inférences successives (types et tailles, emplacements, ordonnancement) et d'une unique traduction vers du code impératif. Ces étapes sont modulaires. Elles ne nécessitent pas d'*inliner* les appels d'opérateurs. Les étapes d'inférence opèrent naturellement sur des opérateurs *polymorphes*. Ils peuvent être génériques en taille, type ou emplacement. Pour la génération de code, la traduction d'opérateurs polymorphes soulève plusieurs difficultés. La première est celle de la représentation dynamique des informations de taille, type ou emplacement.

- *Polymorphisme de taille.* Pour implémenter en C des fonctions génériques en la taille des tableaux manipulés, il est nécessaire de transmettre séparément la taille (un entier) et les données (un pointeur). Ces tailles sont utilisées dans les bornes des itérations et pour les accès aux tableaux multi-dimensionnels.

- *Polymorphisme de type.* La manipulation de fonctions génériques en types peut suivre une représentation similaire. Chaque type abstrait est représenté par sa taille et les valeurs sont manipulées par des pointeurs génériques (`void *`). Il faudrait alors introduire des conversions de types (*cast*). Il en résulte des problèmes d'alignement qui n'ont pas été étudiés.

- *Polymorphisme d'emplacements.* Proposer une représentation dynamique des emplacements demanderait de fournir pour chaque variable de l'interface une fonction permettant de calculer les accès. Cela impacterait fortement les performances. Pour cette raison, notre prototype limite les instanciations à des emplacements représentables par un unique pointeur. Cette restriction permet en particulier de traiter les projections et les sous-tableaux sans introduire de surcoût à l'exécution.

La seconde difficulté provient des restrictions de l'usage de la mémoire. Le contexte de l'embarqué critique impose de maîtriser les besoins en temps et mémoire. Pour cela, l'allocation en Scade est statique. Autoriser le polymorphisme de taille ou de type conduit à des variables locales dont la taille n'est pas connue à la compilation. Pour éviter de recourir à une allocation dynamique, De tels opérateurs pourraient être compilés à l'aide d'un contexte alloué par l'appelant monomorphe dans lequel sont stockées ces valeurs temporaires. Ces aspects de la génération de code sont à un stade préliminaire et restent largement à explorer.

3 Itérer sur les tableaux avec des suites

Les opérateurs de tableaux présentés ci-dessus (`transpose`, *etc*) permettent de modifier la structure des tableaux sans modifier les éléments. Pour effectuer des calculs sur les tableaux, Scade propose des itérateurs du second ordre (`map`, `fold`, *etc*) que l'on trouve fréquemment dans les langages fonctionnels. Ces opérateurs utilisent une fonction pour traiter ou combiner les éléments. Dans Scade, le langage des opérateurs (les fonctions) est séparé car Scade est un langage du premier ordre.

Parmi les faiblesses de ces itérateurs, la compilation actuelle des accumulations (`fold`) introduit des copies qui ne sont pas toujours éliminées. Nous proposons dans le chapitre 5 une construction de premier ordre pour itérer sur les tableaux. Elle adosse les accumulations à l'utilisation de l'état des fonctions de transition. Cela permet de bénéficier des contraintes d'allocation propres aux constructions temporelles de MADL (`last`) et évite l'introduction de copies.

3.1 Tableaux comme des suites

Les langages déclaratifs Compel [TE68] et plus récemment Lucid [AW77] et Sisal [FCO90] proposent d'interpréter les tableaux comme des suites *finies*. Les itérations sur les tableaux sont alors représentées par des définitions purement fonctionnelles de suites. Dans ces langages, les manipulations de suites sont confinées aux constructions d'itération.

Des flots plus rapides Dans les langages synchrones, la notion de suite est omniprésente : les programmes sont des fonctions sur des suites *infinies*. Nous proposons d'unifier l'itération fondée sur les suites avec les flots de Scade en interprétant un tableau de flots comme un unique flot dont les valeurs arrivent par paquets. Ainsi, un calcul itéré correspond à la production d'un paquet de valeurs pour un flot plus rapide que le rythme des cycles. Ces idées de calculs plus rapides ont été explorées dans le carde de la vérification de Lustre [MC05] et pour la programmation réactive avec RML [MPP15].

Nous proposons une construction `forward` que nous illustrons avec l'exemple suivant qui calcule la somme de deux vecteurs. Cette définition est acceptée par notre compilateur qui produit le schéma de type ci-dessous.

```
let vec_add (u, v) returns (w):
  forward ([ui] = u) ([vi] = v) returns (w = [wi]):
    wi = ui + vi;
```

```
val vec_add: size i. [i]int, [i]int → [i]int
```

Dans cet opérateur, les tableaux — u et v — sont parcourus en nommant des *éléments courants* — `[ui] = u` et `[vi] = v` — qui sont des flots dont les valeurs à l'itération i sont celles des $i$ème éléments des tableaux. Le résultat — w — est obtenu en *agrégeant* — `w = [wi]` — les valeurs du flot wi.

Accumulation et état Dans cette construction, l'état joue une rôle central. Il permet de communiquer entre les itérations. L'état sert donc d'accumulateur. L'exemple ci-dessous présente une définition du produit scalaire en MADL.

```
// Scalar product
let dot (u, v) returns (s):
  forward ([ui] = u) ([vi] = v) returns (s = {si}):
    si init 0 = last si + ui * vi
```

```
val dot: size i. [i]int, [i]int → int
```

Ici, le résultat de l'itération — `s = {si}` — est un *accumulateur*. Ce flot doit faire partie de l'état et être initialisé — `si init 0` — afin de posséder une valeur dans le cas où les tableaux itérés sont vides. L'expression itérée accède à la valeur précédente de l'accumulateur avec la construction `last`. Dans le cadre de SCADE, une telle construction d'itération permettrait l'utilisation des constructions temporelles du langage comme les automates pour implémenter des algorithmes itératifs.

Réinitialisation L'opérateur `dot` ci-dessus est combinatoire, son résultat w ne dépend que de la valeur de ses arguments u et v. Cependant, le corps de la boucle utilise un état : l'accumulateur si. Il faut le réinitialiser à chaque cycle. C'est le comportement par défaut de la construction `forward`, qui peut être explicité à l'aide d'une *condition de réinitialisation* `restart`. Une autre condition — `resume` — spécifie le comportement inverse. Dans ce cas, la valeur de l'état à la dernière itération est stockée et l'itération du cycle suivant commence avec celle-ci.

L'exemple ci-dessous calcule la somme des produits des éléments de deux vecteurs u et v : $s = \sum_{i,j} u_i v_j$. Pour cela, deux itérations imbriquées sont nécessaires. Afin d'utiliser un unique accumulateur pour les deux itérations, sans que celui-ci ne soit réinitialisé au début de l'itération la plus interne, la construction `forward` qui lui correspond a pour condition de réinitialisation `resume`. Afin d'insister sur ce point, nous avons aussi ajouté la condition par défaut de l'itérateur externe.

```
let tensor_prod_sum (u,v) returns (s):
  forward ([ui] = u) returns (s = {t}):
    forward ([vj] = v) returns (t = {u}):
      u init 0 = last u + ui * vj
    resume
  restart
```

```
val tensor_prod_sum: size i j. [i]int, [j]int → int
```

Itération multidimensionnelle Ce schéma d'itération multidimensionnelle où les boucles internes ne sont jamais réinitialisées est courant dans les traitements utilisant des tableaux, en particulier pour les opérations d'algèbre linéaire. Pour cette raison, la construction `forward` fournit directement de telles itérations. Ainsi, l'exemple `tensor_prod_sum` ci-dessus peut s'écrire:

```
let tensor_prod_sum (u,v) returns (s):
  forward ([ui] = u), ([vj] = v) returns (s = {{u}}):
    u init 0 = last u + ui * vj
```

```
val tensor_prod_sum: size i j. [i]int, [j]int → int
```

Dans cette définition, la virgule — `,` — est cruciale. Elle sépare les deux dimensions d'itération. Elle correspond à l'opérateur `cross` de SISAL. La clause de retour — `s = {{u}}` — reflète l'accumulation sur les deux dimensions.

3.2 Définition récursive de tableaux

Les algorithmes de tableaux utilisant un pivot, par exemple, la décomposition de Cholesky, nécessitent d'accéder aux éléments déjà construits pour calculer la valeur de l'élément courant. En SCADE, leurs définitions miment des implémentations impératives. Un accumulateur est initialisé avec un tableau de valeurs par défaut dont les éléments sont modifiés un à un à l'aide d'un itérateur et de mises-à-jour fonctionnelles.

De tels programmes présentent deux sources d'inefficacité. (i) L'initialisation, obligatoire en SCADE, est inutile si l'implémentation est correcte car seuls les éléments calculés sont utilisés. Cependant, le compilateur ne peut s'en apercevoir et l'éliminer. (ii) La modification d'un élément et les accumulations sur des tableaux sont des sources de copies qui ne sont pas toujours éliminées. Si notre proposition de langage déclaratif spécifiant la mémoire peut résoudre le second point, il ne peut éviter l'initialisation superflue.

Afin de permettre une définition déclarative de tels algorithmes sans calculs inutiles, nous proposons une extension de la construction d'itération **forward**. Elle permet d'utiliser à l'intérieur de l'itération la partie du tableau déjà définie. Pour l'illustrer simplement, nous proposons de calculer le tableau des puissances de 2 à l'aide de la formule suivante :

$$2^{n+1} = 1 + \sum_{k=0}^{n} 2^k$$

Pour construire un élément de ce tableau, il faut donc sommer l'ensemble des éléments déjà calculés. En MADL, cet algorithme s'écrit de la façon suivante. L'itérateur **forward** externe construit le tableau des puissances — *p* = [*pi*] **as** *l* — en nommant la partie déjà construite — · **as** *l* —. L'itérateur interne somme les éléments de ce tableau en commençant à 1.

```
1 // Powers of 2, using:  p[i] = 1 + sum (p[0:i])
2 let pow2 () returns (p):
3   forward [i] returns (p = [pi] as l):
4     forward [k] ([pk] = l) returns (pi = {s}):
5       s init 1 = last s + pk
```

```
val pow2: size i. [i]int
```

```
1 void pow2 (size_t I, int *p) {
2   for (int i = 0; i < I; i++) {
3     p[i] = 1;
4     for (int k = 0; k < i; k++)
5       p[i] += p[k];
6   }
7 }
```

Notre prototype produit pour **pow2** la fonction C ci-dessus, générique en la taille du tableau construit. Dans le programme MADL, les constructions [*i*] et [*k*] permettent de capturer les indices des itérateurs. Ici, elles servent uniquement d'indications pour le nommage des indices de boucle dans le code généré. Comme attendu, la taille de la boucle interne dépend de l'indice de la boucle externe et ce programme ne nécessite pas d'initialiser le tableau construit.

Conclusion et perspectives

Nous proposons un langage synchrone déclaratif de bas niveau nommé MADL. Il a pour but de concilier les atouts formels des langages purement fonctionnels avec la spécification de la mémoire propre aux langages impératifs. La mémoire dans MADL n'a pas de rôle sémantique, mais elle élimine les programmes qui ne peuvent être implémentés sans respecter le partage d'emplacements spécifié.

Notre solution se nourrit d'une connaissance précise de la forme des données, en particulier de la taille des tableaux, qui permet de décrire l'allocation sous la forme d'un système de types. La définition de l'allocation repose sur la distinction entre les expressions structurelles et calculatoires de MADL. Les premières sont des artefacts de la description purement fonctionnelle qui représentent des modifications statiques de la structure des données alors que les secondes sont les calculs effectués à l'exécution sur les valeurs contenues. À ces contraintes structurelles sont ajoutées des annotations qui guident l'inférence d'emplacements ou requièrent des partages supplémentaires.

Cette proposition dessine les contours d'un langage purement fonctionnel permettant de spécifier la mémoire. Parmi les sujets qui n'ont pas été abordés, le support des conditionnelles

nous semble essentiel. Il nécessiterait certainement d'étendre notre modèle de la mémoire et la description des emplacements.

À l'inverse du schéma de compilation suivi pour Scade, MADL donne la priorité à l'allocation sur l'ordonnancement. Vu comme un intermédiaire de compilation, notre langage permet d'aborder les aspects mémoire de l'implémentation des programmes dans une forme déclarative dont la manipulation devrait être plus aisée qu'une représentation impérative. La spécification indirecte de la mémoire — copies explicites, constructions sans copies et annotations — nous semble s'accorder mieux avec la nature purement fonctionnelle du langage qu'une définition directe de l'allocation. Nous pensons que cette proposition est un premier pas indispensable vers la compilation pour des cibles exigeantes sur la gestion de la mémoire comme les GPUs.

Contents

Notations

Functions

- *Total functions*: $D \to C$ or C^D
- *Partial functions*: $D \rightharpoonup C$
- *Image*: $Im\,(f) \stackrel{def}{=} \{f(x) \mid x \in D\}$ where $f : D \to C$
- *Composition*: $g \circ f$ denotes the function $x \mapsto g(f(x))$
- *Reversed composition*: $f \gg g$ denotes the function $x \mapsto g(f(x))$, i.e., $g \circ f$

Notation (Functional update)**.** Given a function $f : D \to C$, an argument $x \in D$ and a value $v \in C$, the functional update of f in x with v, denoted $f\,\{x \mapsto v\}$, is the function that coincides with f everywhere except for x where it evaluates to v:

$$f\,\{x \mapsto v\} \;=\; \left\{ \begin{array}{lcl} D & \longrightarrow & C \\ x' & \longmapsto & \left\{ \begin{array}{ll} f(x') & \text{if } x \neq x' \\ v & \text{if } x = x' \end{array} \right. \end{array} \right.$$

Sets

- *Integer interval*: $[\![a, b]\!] \stackrel{def}{=} \{\; n \in \mathbb{N} \;\mid\; a \leq n \leq b \;\}$
- *Power set*: $\mathcal{P}\,(S) \stackrel{def}{=} \{\; S' \;\mid\; S' \subseteq S \;\}$
- *Disjunction*: $S_1 \,\#\, S_2$ if and only if $S_1 \cap S_2 = \emptyset$
- The union of disjoint sets will be denoted $S_1 \uplus S_2 \stackrel{def}{=} S_1 \cup S_2$ where $S_2 \,\#\, S_2$. We abusively call it *Disjoint union*, although we impose the disjunction condition rather than adding labels to distinguish common elements.

The notation for disjoint union is also used for functions: given $f : A \to C$ and $g : B \to C$ then

$$f \uplus g \;=\; \left\{ \begin{array}{lcl} A \uplus B & \longrightarrow & C \\ x & \longmapsto & \left\{ \begin{array}{ll} f(x) & \text{if } x \in A \\ g(x) & \text{if } x \in B \end{array} \right. \end{array} \right.$$

Collections

Notation (List)**.** Given a set S, the set of *finite lists*, denoted $S^{(\mathbb{N})}$, is built recursively with the following equation, where ε denotes the empty list and $\cdot$ builds non-empty lists:

$$S^{(\mathbb{N})} = \{\varepsilon\} \cup \left\{\; x \cdot l \;\mid\; x \in S,\; l \in S^{(\mathbb{N})} \;\right\}$$

The *concatenation* of two lists $l_1,\ l_2$ is denoted $l_1 \bullet l_2$.

Notation (Vector)**.** Given a set S, we denote ordered collections of elements of S as a vector $\vec{x}$. Contrary to lists, vectors are used when the elements are independent, i.e., the order is mainly conventional. Vectors are concatenated with a comma: $\vec{x}, y, \vec{z} = x_1, \ldots, x_n, y, z_1, \ldots, z_p$ where $\vec{x} = x_1, \ldots, x_n$ and $\vec{z} = z_1, \ldots, z_p$.

Substitutions

Various formal languages with binders are used across this thesis. We will use a unified notation for substitutions.

Notation (Substitution)**.** Given terms t and t' and a variable x, the substitution of the free occurrences of x in t by t' is denoted $t\{t'/x\}$. This notation is extended to simultaneous substitution: $t\{\vec{t}/\vec{x}\}$.

We take for granted the following equalities, that are extended to simultaneous substitutions:

- Composition: $t\{u/x\} = u\{y/x\}\{u/y\}$ if y not in u
- Commutation: $t\{u/x\}\{v/y\} = t\{v/x\}\{u/y\}$ if x, y not in u, v
- Commutation: $t\{u\{v/x\}/y\} = t\{u/y\}\{v/x\}$ if x not in t

1 Introduction

1.1 Synchronous Programming Languages

Engineered systems can be coarsely classified according to whether they manufacture something or evolve dynamically. The former build from ingredients a product (e.g., an assembly line) whereas the latter "maintain a certain ongoing relationship with their environment" [HP84],e.g., a centrifugal governor that uses physical laws to regulate speed. In the context of software engineering, Harel and Pnueli [HP84] coined the term *reactive* to denote these latter systems that interact permanently with their environment, e.g., a user interface. They contrast with *transformational* systems that generate outputs once inputs are given without further interaction, e.g., a compiler. Compared to transformational ones, a formal description of a reactive system as a function that maps input values to output values is inadequate: their behaviors depend on the environment that evolves dynamically according to its intrinsic laws. This evolution must be modeled or at least taken into account.

With such a definition, a large variety of systems are reactive, from circuits to user interfaces. They can be designed with various methods. We focus here on the restricted class of *controllers*, which aim at driving a physical device (e.g., a plane), according to user commands, named *set-points*. These set-points define desired values for properties that can be measured with *sensors*, e.g., speed or temperature, but that cannot be set directly: the controller only disposes of an indirect action on them by driving *actuators*, that modify the dynamics of the measured values according to the physical laws inherent to the environment. The controller interacts with these different parts, as illustrated in Figure 1.1.

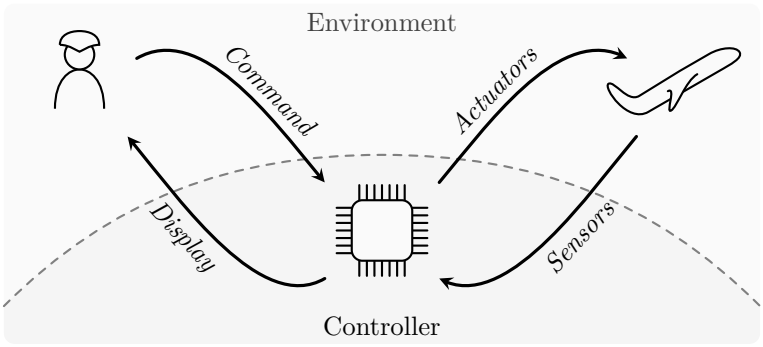


Figure 1.1: *A schematic view of a reactive system*

The dichotomy between the user and the physical system distinguishes the part of the environment that evolves *by itself* from the one that can be set *on purpose*. From the controller point of view, this makes little difference: the user is a (loosely) responsive external system, that is actuated through displays and probed with input devices (joystick, buttons). This environment — both the user and the physical device — may itself be composed of other reactive systems. The importance of parallelism emerges here: complex systems may be decomposed into smaller ones that run *in parallel*, without interactions except through their inputs and outputs.

Controllers must handle values that evolve both *continuously* and *discretely*. The former typically comes from measurements (e.g., speed) or smooth commands (e.g., a joystick) while the latter may represent mode changing (let's land now!) or event watching (e.g., temperature overtaking a critical threshold, a clock tick).

1.1.1 Modeling Continuous Controllers

From a mathematical point of view, the interface of controllers, i.e., input and output *signals*, can be modeled as values that evolve over time, i.e., functions. Then, a controller is a transformation that maps input functions to output ones. An extra requirement is expected: controllers react *on-line*. This is described by a causality property: the output values at time t only depend on the values of signals at previous instants $t' \leq t$.

Although fairly general, this formalism allows to tackle some of the functional properties of a reactive system. In particular, the crucial question of stability can be formulated: given some expected inputs, does the system converge or at least stay within a safe domain?

Proportional-Integral-Derivative controller (PID) The PID controller provides a generic answer to the above stability issue. It is one of the most widely used controllers because it does not require extra knowledge about the underlying dynamics. Given a *set-point* $c(t)$ for a measured variable $v(t)$, it computes the *error value* $e(t) = c(t) - v(t)$ and returns a *control variable* $u(t)$, that is assumed to affect $v(t)$ dynamics, by correcting the error term in the following way:

$$u(t) = K_p \left(e(t) + \frac{1}{T_i} \int_0^t e(\tau) d\tau + T_d \left. \frac{de(\tau)}{d\tau} \right|_{\tau=t} \right)$$

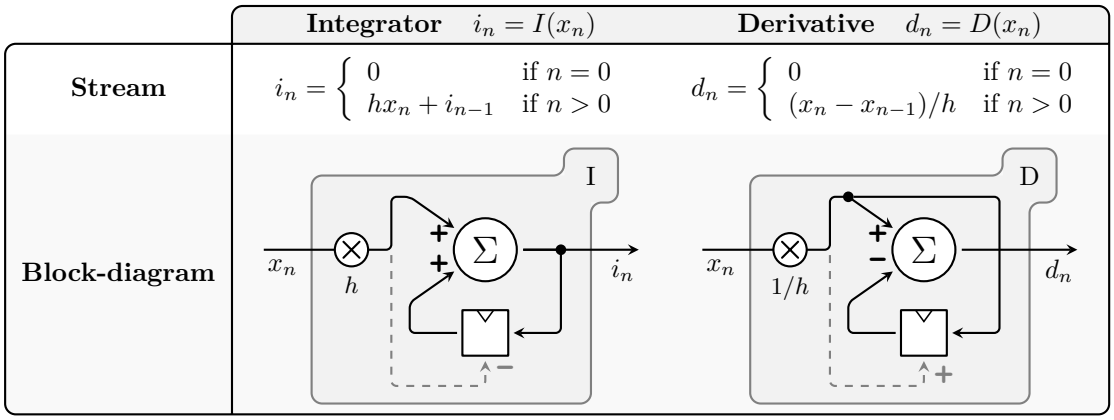
where K_p, T_i and T_d are constants that must be tuned for each system — a balance known as *loop tuning*. Physically, the time T_i/K_p represents how long the error may keep the same sign (the integral accounts for the sum of past errors) while $K_p T_d$ is the characteristic time in which the controller approaches the requested value.

The behavior of a PID controller may be studied independently of any implementation. In particular, to explore control stability (*does it finally reach the set-point?*) or optimal control (*how quickly does it reach the set-point?*).

Discrete-time controllers While physical implementations of PID controllers exist, e.g., pneumatic ones, such a continuous model cannot be implemented in a computer-based controller, whose reactions are inherently discrete. In this context, the dynamic is approximated by *discretizing* signals, i.e., handling them as (infinite) sequences, called *flows* or *streams*.

Controllers are thus stream functions, that relate input and output discretized signals. Beside this mathematical specification, controllers are usually designed with block diagrams.[1] They connect *wires*, that represent streams, to operations.

As an example, let us define the two main components of a PID controller, the integrator and derivative sub-systems, that compute respectively approximations of the integral and the derivative of a flow using a time step h. Both operators require to *look into the past*, by accessing a value of some flows at the previous instant:

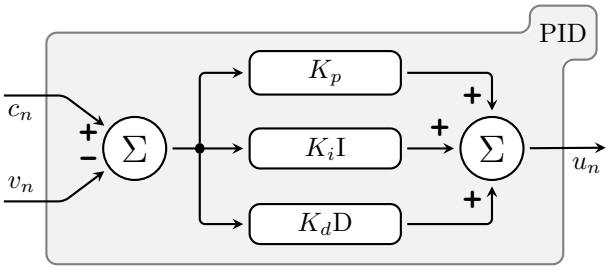


In the diagrams, the inputs are first scaled with the appropriate time constants. The accesses to the previous values, i.e., i_{n-1} and x_{n-1}, respectively, are represented by the boxes, that denote *unit delays* also called *synchronous registers*. At the first instant, these delays are initialized with the flows denoted by the dashed arrows, i.e., $-hx_0$ for `I` and x_0/h for `D`. These two components allow to define a PID controller, whose entries are the set-point c_n and the measured value v_n and

[1] Block-diagrams are also used to describe continuous reactive systems.

that defines the value of the control u_n. In the equations and the diagram, we use $K_i = K_p/T_i$ and $K_d = K_p T_d$.

$$\begin{cases} e_n &= c_n - v_n \\ i_n &= I(e_n) \\ d_n &= D(e_n) \\ u_n &= K_p e_n + K_i i_n + K_d d_n \end{cases}$$



Safety critical embedded systems Controllers are found in *embedded software*, that is code that controls some physical device, e.g., driving the motor of an automatic door. Whereas errors may have benign consequences, e.g., a bruise if a door is too slow to open, some reactive systems, known as *safety critical embedded systems*, may have dramatic consequences in the event of failures, e.g., a braking system. To protect them against errors, synchronous languages [Ber89] have been designed with safety requirements as a primary concern. We briefly review some crucial aspects of safety-critical embedded software.

1. They should execute for an arbitrary long time: no runtime failures are possible. As a consequence, they must not run out of resources, in particular memory.
2. They should ensure some reactivity, i.e., respond to their inputs within a given latency. This imposes a precise knowledge on the execution time of a reaction, notably, unbounded loops are not permitted.
3. They should meet some functional requirements. To this end, determinism simplifies their verification.

For legal reasons, safety critical systems must be certified by independent authorities before being deployed. Although they do not prove the absence of bugs, these certification processes require an obligation of means (extensive testing, constraints on coding practices and project management) that aims at delivering highly reliable systems, both for hardware and software sides.

In the context of Scade [CPP17], that targets the highest levels of certification (e.g., DO178C, level A of avionics), a major restriction on the language expressiveness follows from the above considerations. Memory management is known at compile-time. As a consequence, no dynamic allocation or freeing is needed, which responds partially to item 1. The execution time can be bounded more easily, for example, by knowing iteration bounds, which helps for item 2. This static memory constraint is an opportunity. The language we propose in this thesis builds on it to define a statically managed memory model.

1.1.2 The synchronous approach

Controllers introduce a notion of time in programs. They are made of multiple tasks that execute in parallel, either periodically or sporadically. The historically most used programming model, concurrent threads, gives little help for writing correct programs. It even leads to defects such as deadlocks or data-races that are notoriously hard to detect. In this context, reconciling parallelism and modularity is a source of non-determinism [Lee06], a major obstacle to the verification of program properties.

The synchronous model Another programming model, synchronous programming [BB91], emerged in the 1980s. It abstracts time away, with a fundamental idea, called the *synchrony hypothesis*. Computations and communications are considered synchronous, they take no *observable time*. This idealization allows to separate the outside timing issues, i.e., when do input signals arrive, from the logical timing of the internal computation. This separation enjoys a simple mathematical formulation.

The synchrony hypothesis is in strong connection with circuit design: computations occur at the ticks of a global clock, by waiting for outputs to stabilize while keeping inputs constant. Once stable, the results of a cycle are then saved for the next one. Among the strengths of this abstraction, the global agreement on a base clock is the key for deterministic parallel composition.

```
node integ (x,h: real)
returns (ix: real)
let
    ix = 0 -> pre (ix + x * h);
tel
```

(a) The integrator

```
node deriv (x,h: real)
returns (dx: real)
let
    dx = 0 -> (x - pre x) / h;
tel
```

(b) The derivative

	1	2	3	4	5	···
x*h	1	1	0	-1	-1	···
ix + x*h	1	2	2	1	0	···
pre (ix + x*h)	*nil*	1	2	2	1	···
0 -> pre (ix + x*h)	0	1	2	2	1	···
0	0	0	0	0	0	···

(c) Histogram of the integrator ($h = 1$)

```
node pid (kp,ki,kd,c,v,: real)
returns (u:  real)
var e: real;
let
    e = c - v;
    u = kp * e
        + ki * integ (e)
        + kd * deriv (e);
tel
```

(d) The main controller

Figure 1.2: *A Lustre implementation of a PID controller (with a unit time step)*

Tasks are either simultaneous or ordered by data-dependencies, without any possibility for data-races. This synchronous approach was historically provided with two programming styles:

- The Esterel programming language, proposed by Berry and Cosserat [BC84], provides an imperative style to describe synchronous automata and their composition. This style is suitable for control-flow based software, that is composed of various modes.
- The Signal [BLJ91] and Lustre [Hal+91] languages rely on a data-flow style: inputs and outputs are related by a set of (clocked) recurrent equations. This style fits the needs of signal processing software, that naturally describes values as streams.

These two styles are not mutually exclusive: the Scade language, that inherits from Lustre, has been extended with control flow constructs, e.g., automata whose representation is inspired from Esterel and its graphical representation SyncChart[And96a; And96b]. We will return shortly to the history of Scade.

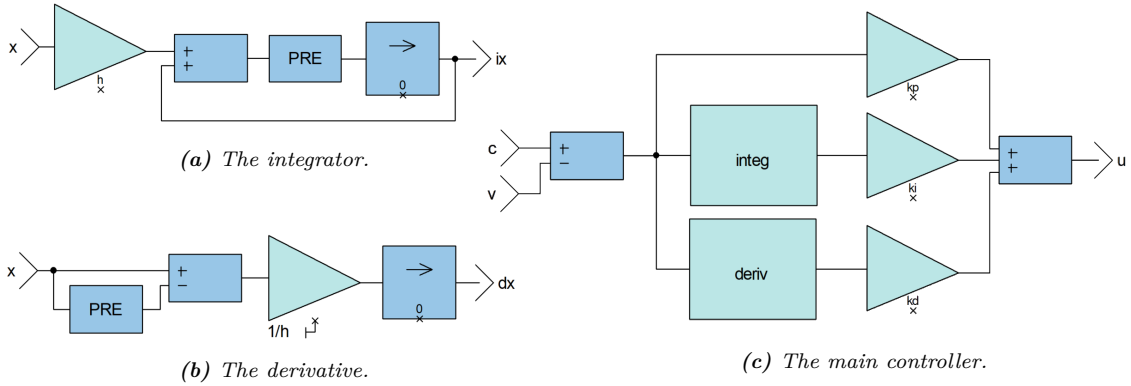
A synchronous data-flow core The Lustre language conciliates the synchronous principle with two older works: (i) the concurrent process networks of Kahn [Kah74], a model of concurrency that ensures determinism and (ii) the data-flow language Lucid of Ashcroft and Wadge [AW77]. It is a purely functional language whose values are streams.

The core of the language consists of a few stream constructs: constants and scalar operations are lifted to flows, the `pre` operator introduces an uninitialized unit delay and the `->` binary operator allows to complete an uninitialized (the right operand) flow with the first value of another (the left one). This is also the core of the Scade language.[2]

As an example, we can implement the discrete PID controller, as shown in Figure 1.2. The program is divided into *nodes* that each consist in a set of recursive flow definitions. So as to define a unique stream, a causality constraint applies: every feedback loop must cross a `pre` operator. This condition is illustrated by the histogram of the `integ` node in Figure 1.2c. The `ix` flow appears recursively in its definition, but it is shifted right by the `pre` operator. The first undefined value *nil* gets dropped with the `->` one. We present in Figure 1.3 screen captures of the Scade graphical editor that defines the same PID controller.

Semantics and fix-points Nodes may be modeled as a system of recursive equations over streams. Purely functional programming languages such as Haskell provide a direct way to define such a semantics: streams are represented by coinductive data-structures that are evaluated lazily. Caspi and Pouzet [CP96] drew on this idea to define a succinct interpreter of the Lucid Synchrone

[2] In this thesis, we ignore clocks, which are an important part of the core of the Scade language.



(a) The integrator.

(b) The derivative.

(c) The main controller.

Figure 1.3: *Block diagram representation of a PID in* SCADE.

language. With this interpretation, a causally incorrect program such as `x = x` or `x = x+1` leads to a diverging term.

Scott [Sco76] proposed to handle mutually recursive definitions as a fix-point computation: the domain of values is turned into a flat complete partial order (CPO) by adding a special value $\bot$ that represents diverging computations. Operations are lifted into this extended domain so as to be continuous. Recursive definitions are then the least fix-point of a continuous function. For a bounded height CPO, this fix-point can be computed in finite time by iterating the definitions, starting from diverging ($\bot$) values.

This idea was used to define a state-based semantics of synchronous programming languages [CP98] and block diagrams [EL03]. It consists in computing a fix-point at every instant. It has been recently extended to define an interpreter for a more expressive language like SCADE [Col+23].

Such a fix-point-based definition has also been applied at the level of streams [Cas+08], by using the *prefix order* proposed by Kahn [Kah74]. Contrary to the value-based fix-point, streams are infinite values which leads to a CPO of infinite height. This method elegantly defines the semantics, but does not provide an interpreter.

Model checking Formal verification of synchronous programs has been studied since the early ages of synchronous languages. The proposed methods inherited from symbolic model checking methods [Bie+99]. Raymond [Ray18] proposed LESAR, a model checker for LUSTRE programs that allows to verify safety properties. Shortly after, Sheeran, Singh, and Stålmarck [SSS00] proposed k-induction, a proving technique that allows to check properties over time using SAT solvers, that are primarily designed to solve combinatorial problems. This method serves as basis for the PKIND [KT11] model checker and its successor, KIND2 [Cha+16].

Hybrid programming Reactive systems interact with physical environments, that are usually described with ordinary differential equations (ODEs). The environments are worth modeling, either to simulate the interactions between a controller and its environment, or to predict the behavior of the driven system.

Continuous models are approximated with numerical methods such as the Euler or Runge-Kutta ones, that discretize time, in a similar way to the abstract model of time of synchronous languages. However, these two time sampling have different properties: whereas the synchronous ticks are usually thought of as regularly spaced, approximation samples should be more frequent when the dynamics is steeper and more loosely distributed when the dynamics is smoother, so as to conciliate the precision and performance of numeric methods.

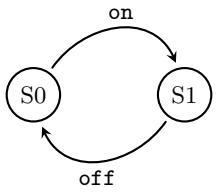
To use external solvers with varying time steps, a hybrid synchronous language called ZELUS [Ben+11; BP13] was proposed. ZELUS programs consist of a mix of continuous (ODEs) and discrete (flows) definitions. The runtime conciliates the time of external ODE solvers, that depends on the dynamic of the functions being approximated, with the logical time of synchronous reactions, that occur at events defined by continuous time zero-crossing event.

1.1.3 Code Generation

The compilation of synchronous programs has been studied for various targets: circuits, high-level languages such as OCAML or HASKELL, and imperative languages such as C or ADA. Because we

```
node switch (init, on, off: bool)
returns (state: bool)
let
    state = if (init -> pre state)
            then not off else on;
tel
```

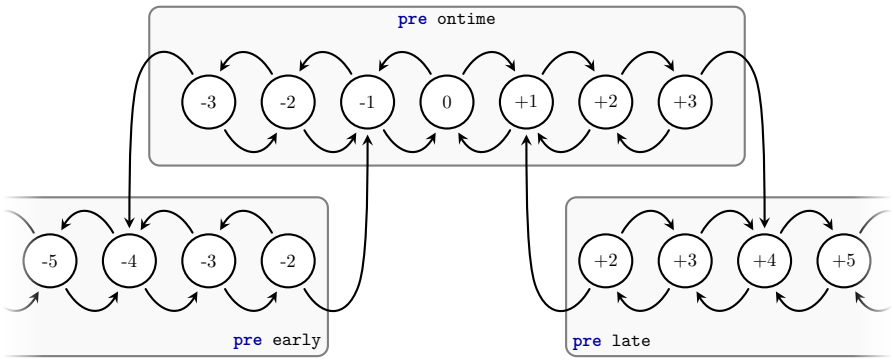
(a) A flip-flop switch



(b) The two-modes switch automaton

```
node counter (second, beacon: bool)
returns (late, early, ontime: bool)
var diff: int;
let
    diff   = (0 -> pre diff)
             + (if beacon then  1 else 0)
             + (if second then -1 else 0);
    late   = switch (false, diff < -3, diff > -2);
    early  = switch (false, diff >  3, diff <  2);
    ontime = not (early or late);
tel
```

(c) The counter



(d) Partial representation of the beacon counter states, based on [Ray18]. The systems moves left if `second and not beacon`*, it moves right if* `not second and beacon` *and it stays in the same state if* `second = beacon`*.*

```
while (true) switch (stt) {
case Early:
  diff  += beacon - second;
  early  = true;
  late   = false;
  ontime = false;
  if (diff > -2) stt = OnTime;

  break;
}
```

```
case OnTime:
  diff  += beacon - second;
  early  = false;
  late   = false;
  ontime = true;
  if (diff < -3) stt = Early;
  if (diff >  3) stt = Late;
  break;
```

```
case Late:
  diff  += beacon - second;
  early  = false;
  late   = true;
  ontime = false;

  if (diff <  2) stt = OnTime;
  break;
```

(e) Multi-mode implementation

```
while (true) {
  diff  += beacon - second;
  early  = early ? !(diff > -2) : (diff < -3);
  late   = late  ? !(diff <  2) : (diff >  3);
  ontime = !(early || late);
}
```

(f) Single-loop implementation

Figure 1.4: *The beacon counter [Ray18]*

target embedded software, this thesis focus on the imperative languages.

To illustrate the difference between various compilation techniques, we borrow in Figure 1.4 the *beacon counter* example proposed by Raymond [Ray18]. This program computes whether a train is early, on-time or late according to a pair of signals: one is received each time the train crosses one of the beacons that are regularly deployed on the railway and the other ticks once every second. The average cruise speed should be one beacon per second. The late and early properties are modeled with *switches*. These components represent a two-mode automaton whose enable (S0→S1) and disable (S1→S0) transitions are controlled by distinct signals (see Figure 1.4b). In the `counter` node, the asymmetries between the `on` and `off` signals of the switches prevent the resulting flags from oscillating, a mechanism called *hysteresis*.

Multi-mode execution The compilation of synchronous languages was inspired by Mealy machines [Mea55]. This kind of finite automata react to their inputs according to an internal state. Synchronous programs can be viewed as Mealy machines by encoding each reaction as a transition of an automaton. For real-time systems, which are submitted to latency constraints, there is a implementation requirement: the worst case execution time (WCET) of each transition must not exceed the time credit for a single step, i.e., the period of the base clock if the computation is periodic.

Optimizing this WCET demands in particular a clever choice of the control structures in the modes: the fewer conditionals the better. Thus, adding more (well-chosen) states to the automaton allows to further specialize the code in each mode, hence eliminating some conditionals. This principle, that we refer to as *multi-mode execution*, has been heavily studied for the compilation of both Lustre [Cas+87] and Esterel [BG92].

For the beacon counter example, some of its possible states are depicted in Figure 1.4d. They are defined by (i) the value of the `diff` variable, represented inside the nodes and (ii) the state of the switches, that correspond to the `early` and `late` flows:[3] only `early` (resp. `late`) is enabled in the bottom-left (resp. right) region, while both flows equal `false` in the top region. This representation gives a clear hint of a mode slicing, by separating the regions. Figure 1.4e depicts a possible imperative implementation of the automaton. In each mode, the code is specialized: except the computation of `diff`, only constants are written.

Single-loop execution The specialization of modes has a major drawback: the size of the automaton may explode, i.e., small programs may become large automata. A simpler translation of Lustre programs as an infinite loop that reads inputs, computes and writes outputs [Hal05] has long been considered as folklore and received little attention. In the context of Lustre, which has a data-flow style, the choice of the control structure is derived from some boolean state variables. Halbwachs, Raymond, and Ratel [HRR91] presented several strategies to minimize the number of modes while preserving specialization benefits. To compare the various approaches, they used as a basis the simplest implementation: a single mode automaton. In this case, called *single-loop execution*, the implementation reduces to a transition function that is encapsulated in an infinite loop. As expected, they reported much smaller execution times for the multi-mode versions than for the single loop implementation.

As evidence for this result, an optimized single-loop implementation of the beacon counter program (with the same assumptions as above) is given in Figure 1.4f. The differences are obvious: the single loop version is much shorter, but cycles of the multi-mode one contain fewer conditional instructions. Even if no mode-splitting is involved, Giavitto, de Vito, and Michel [GdM97] pointed out that single-loop compilation still has optimization possibilities. They involve sharing computation guards, to avoid redundant conditionals in the generated code.

Despite this unfavorable observation, the naive approach was to have a bright future: it turned out to be unavoidable for the compilation of the industrial language Scade [Bie+08]. The reason for giving up multi-mode compilation lies in the quality of the generated code. All the multi-mode compilations require the model to be flattened first: the modular program is turned into a possibly large monolithic description. The code that results is hardly traceable, a requirement for the certification of highly critical embedded systems.

Modularity Compared to the multi-mode approach, single-loop compilation is more compatible with modular compilation. The compiler may generate one transition function per node instead of a

[3] Because a step corresponds to a transition of the automaton, the starting state is actually defined by the previous values of `late` and `early`, i.e., `pre late` and `pre early`.

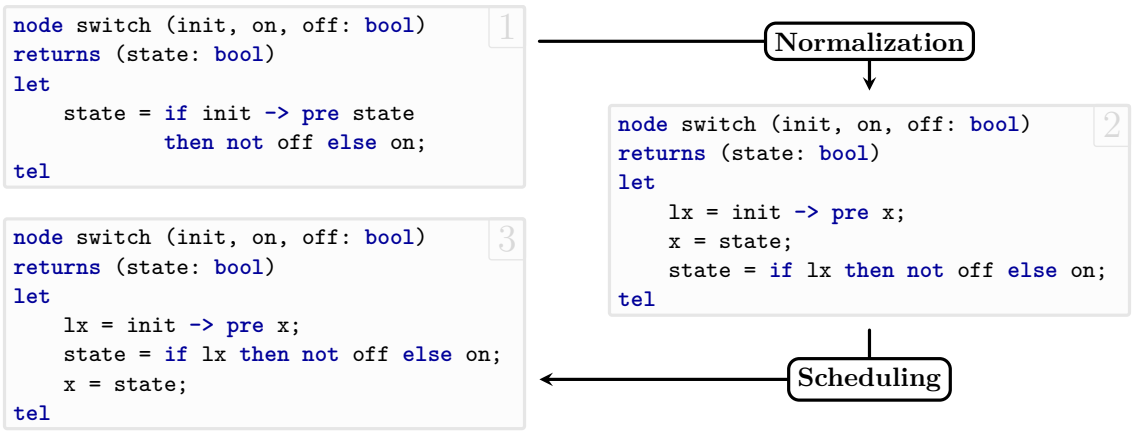


Figure 1.5: *Source-to-source compilation. After scheduling, the data-flow program can be directly translated into an imperative form, where the state is made of the variables that appear in* `pre` *(*`x` *here)*

unique automaton. The structure of the source program (nodes) is thus transparently forwarded to the generated code (transition functions). However, modularity comes at the expense of causality. This compilation scheme assumes that functions are atomic, they are evaluated once all their arguments are available. In other words, the registers, that break causality loops, may not be encapsulated in nodes. As a mitigation, the SCADE compiler allows explicit inlining specifications, to abandon locally and on demand the modularity principle for the benefit of relaxed causality.

Some years later, the causality restriction was studied again, with modularity as a priority. Pouzet and Raymond [PR09] proposed a static analysis to determine how to split nodes into multiple functions and allow any valid feedback loop without inlining. Contrary to the initial multi-mode compilation, these functions do not represent different modes, but contain parts of the transition function. All of them must be called at each cycle.

Source-to-source compilation Various extensions (automata, modular reset, ESTEREL-like pure signals) have been proposed in LUCID SYNCHRONE [CP96], a language reminiscent of LUSTRE. Their compilation is based on *source-to-source* transformations. Programs are rewritten in the same language with simpler constructs, until they are expressed in a clocked data-flow core [CPP05]. This approach allows a precise and comprehensible formalization of the compilation process, which was decisive in the successful transfer of these ideas into SCADE.

The *source-to-source* approach is also used to bring the clocked data-flow core closer to an imperative form [Bie+08]. It is depicted with the `switch` node example in Figure 1.5. The normalization pass identifies memories by introducing fresh variables for the occurrences of `pre`, and separating the computation of the new value from access to the old one. The scheduling pass orders the equations in a sequence of *assignments* in an imperative language. The resulting program is ready for a straightforward translation into a simple imperative object-based language that is later compiled down to an imperative language such as C.

More recently, Bourke et al. [Bou+17] proposed VÉLUS, a LUSTRE compiler whose correctness is formalized and proved in the COQ proof assistant. Starting from a clocked data-flow core, the modular reset was then added by Bourke, Brun, and Pouzet [BBP20]. The introduction of automata is on its way.

A brief history of Scade Figure 1.7 glances at some synchronous language projects, centered on SCADE. The SCADE project arose from the merger of two projects: (i) the SAGA code generator from Schneider Electric, an industrial compiler for LUSTRE v.3[4] (written in C) and (ii) the SAO tool of Airbus for representing block diagrams. It aimed at providing a graphical development environment based on a well defined synchronous language, an asset compared to existing competitors such as Simulink [5]. In particular, it comprised a graphical editor and a code generator that targeted C.

In 2000, the above source-to-source compilation served as a basis for the RELUC compiler [Bie+08], written in OCAML. This was the first step toward a formalization of SCADE. At the

[4] LUSTRE and the SCADE differ on the handling of arrays, which were introduced in version 4 of LUSTRE.
[5] https://fr.mathworks.com/products/simulink.html

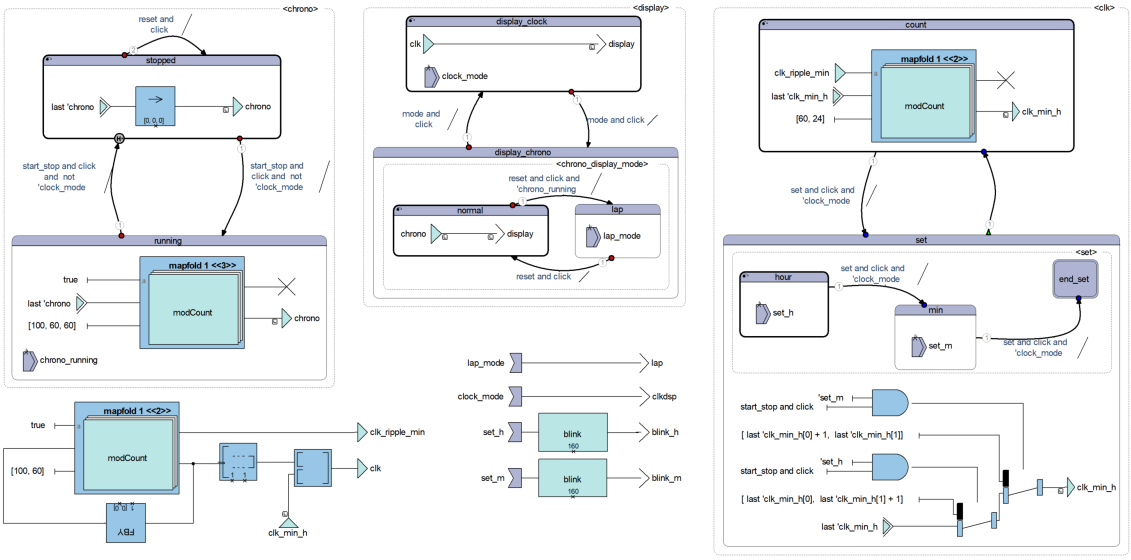


Figure 1.6: *The digital watch model in* SCADE.

same time, a model checker for SCADE was developed, based on the tools of Prover[6]. To comply with the certification requirements, a model coverage[7] tool was also released.

Following the ideas explored with LUCID SYNCHRONE [CP96], the static verifications (causality, initialization and clock correctness) were described as type systems [Col+04]. The SCADE 6 version [CPP17] of the language, released in 2008, introduced several extensions: (i) the support for automata [CPP05] was inspired by ESTEREL, in particular for their graphical representation, that came from SYNCCHARTS [And96a] and (ii) arrays were added to the language. Figure 1.6 gives an overview of a SCADE model for a digital watch. This is an enriched version of the chronometer example presented by Colaço, Pagano, and Pouzet [CPP05]. It features a mix of data-flow diagrams and nested automata.

The development of ZELUS resulted in a hybrid version of the language [Bou+15], and a multi-task code generator was also released [Col+18].

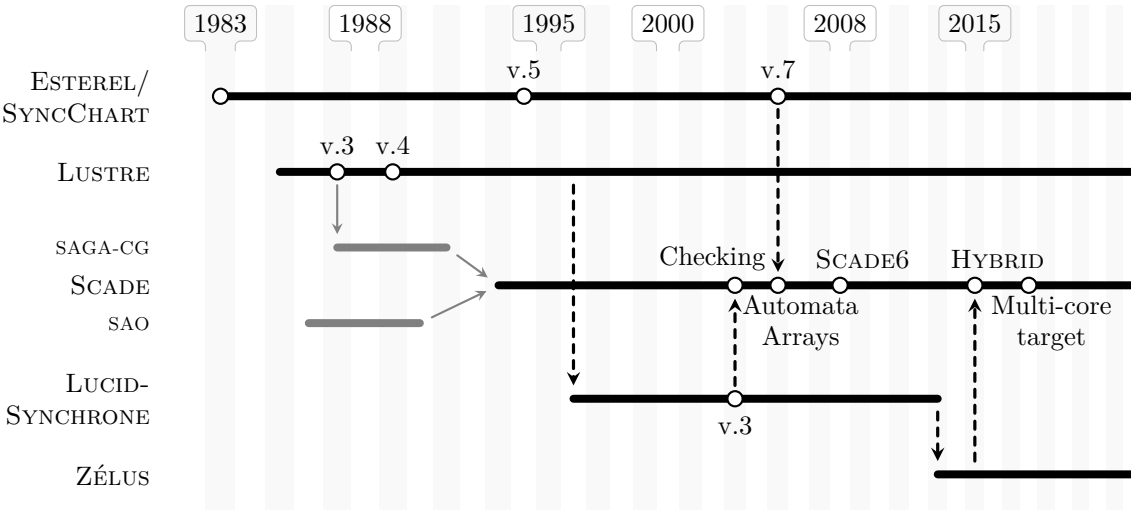


Figure 1.7: *The timeline and influences of* SCADE

[6] https://www.prover.com

[7] https://www.uni-bamberg.de/fileadmin/uni/fakultaeten/wiai_professuren/grundlagen_informatik/Synchron16/presentation-slides/ANSYS_synchron16.pdf

1.2 Arrays and their Correctness

Arrays are contiguous chunks of memory used to store homogeneous values. This allows to dynamically compute the addresses of elements. The i^{th} item is located at the address $bp + i \times vs$ where bp denotes the address of the array (the base pointer) and vs is the size of an element. Unlike lists, array accesses have complexity $\mathcal{O}(1)$, i.e., they require constant time independently of the accessed element.

1.2.1 Array Operations

Processors may not compute on arrays atomically,[8] hence each array operation must be decomposed into individual accesses and computation on their elements. Low-level languages represent arrays as an address in memory, alongside information on the size of elements to automatically compute offsets. The size of an array, i.e., the number of elements, is not attached to its representation, but must be maintained independently. For instance, adding two vectors point-wise could be implemented with the following `for` loop in C:

```
for (int i = 0; i < N; i++)                                    (C style)
  w[i] = u[i] + v[i];
```

Although simple, these two lines of code host various sources of bugs, that might not be caught by the static analyses of a C compiler. Some of them may be unearthed with the following questions:

- *Are the indexes within bounds?* Because indexes are computed dynamically, they might overflow their planned range. The risk is, at best, a stop of the execution if an invalid memory address is accessed or, at worst, a silent corruption of memory. The latter is a fruitful source of attacks (e.g., buffer overflows). In the C language, the size of arrays (i.e., `N` in the above example) is set apart from the arrays. Thus correctness of accesses emerges from a global consistency between declarations and access expressions, for which automated analyses are costly and incomplete.
- *Are no uninitialized values used?* This property is crucial to ensure determinism and it is non trivial in languages where allocation is separated from assignment. It may be safely approximated by requiring all the values to be fully initialized. However, there are still subtle points in the above example. For the `w` array to be fully initialized, its size must actually be `N` and the index expression `i` must take all the possible values between `0` and `N` excluded. While obvious in the present example, the second property is much less trivial if the write access were more complex, e.g., `w[N-i-1]`.
- *Are the arrays independent?* The result of this simple program depends on *aliasing*: if the array `w` overlaps with the end of one of `u` or `v`, the first iterations will override the elements that are needed after, hence computing an erroneous, or at least different, array. In this example, the result would be correct if the result aliases exactly `u` or `v`.

In C, these concerns are best practices rather than constraints. Programmers might bypass them if needed, e.g., the *flexible array member* idiom in C where the declared and real size of array differ. However, this flexibility has a price, even the simplest uses of arrays are unchecked.

Arrays as first-class objects Sufficiently high-level programming languages, from Java to Haskell, support safe arrays, which transfer the responsibility of array correctness to a compiler. They have in common that array values incorporate their size.

Then, it suffices to guard accesses so as to ban array errors. The compiler surrounds each access with a check that verifies that the requested element exists. Otherwise, a recovery behavior is triggered, e.g., throwing an exception or returning a default value. However, the costs are twofold, (i) it impacts performance drastically by introducing pervasive conditionals, hence losing the major asset of arrays, their low access complexity and (ii) it requires superfluous error handling code, which is dead code if accesses are actually correct.

A generic solution for the initialization issue exists as well, by imposing a full initialization when allocating arrays. The drawbacks are similar. A dummy initialization induces an extra execution cost and dead code if the elements are to be computed by other means after the initialization.

[8] In some processors, vector instructions are available. They apply to arrays of fixed sizes.

High-level operations on arrays As an alternative to the costly solutions sketched above, higher level languages provide complex operations over arrays as language constructs. By alleviating the need for explicit index manipulations, they reduce the above questions to simpler assumptions about array shapes.

To incorporate arrays in COMPEL, a single assignment language for parallel computing, Tesler and Enea [TE68] extended the notion of variable from a single value to sequences. This allows to express point-wise operations, i.e., parallel iteration, in a declarative way. The LUCID language went a step further: Wadge, Ashcroft, et al. [WA+85] proposed *temporal operators* **fby** and **next** to shift these streams in time, allowing to express accumulations such as the sum of elements, i.e., sequential iterations, in a data-flow style. The SISAL language [FCO90] takes the same approach. It distinguishes parallel iterations — *for-loop* construct – from sequential ones — the *for-initial*. The previous sum of vectors is written in SISAL:

```
w := for ui in u dot vi in v                                    (SISAL style)
    wi := ui + vi
  returns array of wi
  end for
```

The **dot** keyword allows to combine the so-called *range generators* — `ui in u` and `vi in v`. This kind of expression resembles *list comprehensions*, a construct that defines arrays out of an enumeration of its elements. Such a style, originated from HASKELL, is particularly used in PYTHON, where the sum of vectors (represented by lists) can be computed with:

```
w = [ ui + vi for ui, vi in zip (u, v) ]                        (PYTHON style)
```

The above styles are centered on elements. These constructs declare variables to traverse arrays and computations are expressed at the level of elements. As an alternative way of describing apparently atomic operations on arrays, higher-order languages such as OCAML provide second-order *iterators* [BW88]:

```
let w = map2 (+) u v                                            (OCAML style)
```

These operations are known as *intensional* [OW92; RW98], a term that comes from logic, by contrast with *extensional* operations, typically involving explicit index manipulation for arrays. The former handle compound objects as single values, taking advantage of their structure, while the latter are described in terms of computations over the elements, independently of their structure.

In extensional description, the overall meaning of the operations — the intension — hides behind implementation details that make it hard to retrieve. Here lies the advantage of intensional description, it allows to reason at the level of compound values, independently of their actual representation in memory.

Multi-dimensional arrays Some collections of values are naturally described with multiple indexes, e.g., a matrix. They can be represented as arrays whose elements are themselves arrays. However this is often more general than needed: unless strong typing conditions are enforced (see below), it does not account for the *regularity* property, i.e., that all the nested arrays have the same size. Non-regular arrays might be useful, e.g., to store a triangular matrix in an optimized way, but they introduce complex accesses that make analysis and compilation harder.

Because a wide range of computations (e.g., linear algebra, image processing, *etc*) are only based on regular arrays, also referred to as *hyperrectangular*, some languages such as APL [Ive62] have been designed to take advantage of this additional structure. While uni-dimensional arrays are described by their *size*, regular multi-dimensional ones are characterized by their *shape*, i.e., a uni-dimensional array that specifies the size of each dimension. The size of the shape, i.e., the number of dimensions is called the *rank*.

Genericity Most array algorithms make no assumptions on the size of arrays. In high-level programming languages, this is reflected by array operations that are generic in the size. The above OCAML expression — `map2 (+) u v` — would apply to any pair of arrays, as long as their sizes match: `map2` is a *size polymorphic* operator.

For multi-dimensional array languages, an additional level of genericity arises: some operations are independent of the number of dimensions, i.e., the rank. In the pioneering language APL,

introduced by Iverson [Ive62], operations are *rank polymorphic*: they are implicitly lifted to operate on arrays of arbitrary rank. Hence the point-wise sum of vectors is written simply:

```
w ← u + v                                                    (APL style)
```

The '+' operator is the *rank polymorphic* addition: it can be applied to any pair of arrays of consistent shapes. In particular, the sum of two matrices `a` and `b` would equivalently be written `a + b`. Similar liftings are found in mathematical environments such as MATLAB.

	Scalar (0D)	Vector (1D)	Matrix (2D)

C	`z = x + y;`	`for (int i = 0; i < N; i++)`
		`  w[i] = u[i] + v[i];`
			`for (int i = 0; i < N; i++)`
			`  for (int j = 0; j < P; j++)`
			`  c[i][j] = a[i][j] + b[i][j];`

SISAL	`z := x + y`	`w := for ui in u dot vi in v`
		`  returns array of ui + vi`
		`  end for`
			`c := for ai in a dot bi in b`
			`  cross aij in ai dot bij in bj`
			`  returns array of aij + bij`
			`  end for`

PYTHON	`z = x + y`	`w = [ ui + vi`
		`  for ui,vi in zip (u,v) ]`
			`c = [ [ aij + bij`
			`  for aij,bij in zip (ai,bi) ]`
			`  for ai,bi in zip (a,b) ]`

OCAML	`let z = (+) x y`	`let w = map2 (+) u v`	`let c = map2 (map2 (+)) a b`
APL	`z ← x + y`	`w ← u + v`	`c ← a + b`

Figure 1.8: *Point-wise sum in various dimensions and styles*

As a summary, Figure 1.8 gathers some possible implementations of a point-wise sum. In the imperative style (C), one loop is needed per dimension and the bounds must be specified. They are independent of the arrays. Moreover, some restrictions apply for the bidimensional version: the access `a[i][j]` is only valid if the compiler is able to deduce the size of the second dimension of `a` and likewise for `b` and `c`. The stream iteration (SISAL), list comprehension (PYTHON) and iterator (OCAML) styles are similar: sizes are implicit but expression structures depend on the number of dimensions. The former are centered on elements, i.e., it traverses arrays to compute on scalars, while the latter places the emphasis on arrays, i.e., it lifts the scalar function to operate on arrays. Lastly, in APL style, both sizes and dimensions are implicit: the operations are automatically lifted to apply to arguments of any rank.

1.2.2 Circumventions of the Bound Checking Issue

Several strategies have been studied to alleviate the need for guarded accesses while preserving array safety. We briefly review some of them.

Arrays as toruses The domain specific language ARRAYOL [Bou07] targets signal processing applications, in particular for radars. In this context, most array accesses are circular, i.e., the actual index is computed modulo the size of the array. Similar access schemes may be found to render textures. Like guarded accesses, this precludes illegal accesses at the cost of index computation complexity (modulo operations are costly if the modulus is not a power of 2). Moreover, this torus view of arrays is rather specialized and useless for linear algebra operations, for instance.

A similar approach was proposed for HEPTAGON [Gér+12], a LUSTRE-like language with automata and array extensions: accesses are clipped, i.e., the access `a[>i<]` denotes the element of array `a` at index $\min(\max(0, i), n-1)$, where n is the size of `a`.

Automatic dimensioning The loop constructs of SISAL [FCO90] iterate over *ranges*. The range generators may be combined with two operators: the **`cross`** and the **`dot`** product. The latter results in a range whose domain is the maximal valid domain, i.e., its size is the minimum of the

sizes of the two ranges. Uses of arrays are thus always correct, but the computed results might not be as large as expected.

Instead of computing sizes in a data-driven fashion, i.e., by deriving the size of results from the size of arguments, some domain specific languages for data processing proposed demand-driven computations: the size of intermediate computations is determined by the expected size of the result. For instance, the HALIDE language [Rag+13] describes image processing pipelines with local computations that use symbolic indexes. The ranges for these variables are deduced by the compiler. This approach takes advantages of just-in-time compilation, a common technique for GPU compilation: parts of the execution environment, in particular array sizes, are known at compile-time.

More recently, Sengul [Sen23] proposed lazily evaluated arrays for APL. The evaluation of intermediate arrays is driven by their use. The objective is efficiency. Laziness allows to compute only the parts that are needed. This is useful, for instance, to render a zoomed part of a larger scene, without altering the definition of the complete scene.

Size consistency Intensional operations greatly lower the required check, by turning the *bound checking* issue into a *size consistency* check. Programs are correct as long as some sizes are equal. On the execution side, instead of guarding all accesses, it suffices to check that the sizes match before executing intensional operations, e.g., iteration, with unguarded accesses. On the verification side, checking size *equalities* is simpler than ensuring index *inequalities*.

1.2.3 Static Checking of Sizes

Intensional operations are a first step towards verified arrays. Eliminating size inconsistencies statically is practicable on the condition that sizes can be described at compile-time. In the context of data-processing applications, Jay and Cockett [JC94] observed that the shapes of data-structures may influence the data, e.g., compute a mean, but rarely the converse. They coined the term *shapely* to describe operations where the sizes of the results depend only on the sizes of the arguments.

Sizes as static computations In order to check sizes statically, the expressions they depend on must not use values that are only available at runtime. This is precisely the goal of the *binding time analysis* proposed by Nielson and Nielson [NN88], although it was designed for compilation purposes rather than verification ones.

A similar separation was proposed by Jay and Sekanina [JS97] for the VEC language, a slightly extended λ-calculus. A family of types, the *datum-free* ones, represents the computations that depend only on statically known values. The authors define a *shape translation* that extracts from a given term t a shape term $\#t$ with the following property, if a shape error occurs when reducing t, then $\#t$ reveals a shape error too.

The FISH programming language [Jay98] draws the boundary between static sizes and dynamic terms in another way: shape and data expressions are taken from different languages. Both of them are given a dedicated reduction system, related by a similar result. The data expression reduces to a shape error if the shape expression does too.

The design of (scalable) regular circuits to implement computations such as sorting or multiplication have been studied from a functional point of view: Sheeran [She85] proposed the μFP language that builds on functional iterators to define regular circuits. This language, as well as its successors, LAVA [Bje+98] and WIRED [ACS05], extensively use arrays (represented as lists). Because of the compilation target, this structural part is independent of the runtime values (wire voltages) by construction and is evaluated at compile-time, during the definition of the resulting circuits.

This approach is generic. Arbitrary size computations may be evaluated. However, it is non-modular. Evaluation is only possible once size parameters are given concrete values. To remedy this modularity issue, more abstract descriptions of sizes have been studied using extensions of the Hindley and Milner type discipline [Hin69; Mil78]. They can be classified in two kinds.

Sizes as typing properties: (i) type families Such extensions supplement base types, e.g., integers, with an additional type index that distinguishes its various versions. In other words, base types are turned into a family of independent types. In this context, a typical type language looks like:

$$\tau ::= \alpha \mid \mathcal{B}\ \iota_1\ \ldots\ \iota_n \mid \tau \to \tau$$

where $\mathcal{B}$ is a *base type* such as `int` or `real` and $\iota_1 \ldots \iota_n$ are *type indexes*, taken from an index language. Types with the same base type but different indexes are considered incompatible, even though data might be represented in the same way. To describe generic functions, the index language contains variables that are generalized with polymorphism in a similar way to type variables.

The *dimension types* designed by Kennedy [Ken94] are an emblematic use case. The scalar type — `real` — is turned into a type family by supplementing it with (physical) dimensions. The index language is made of symbols (the base units) and an inner operation, the product. Scalar operations are given precise polymorphic types. For instance, with the above syntax, the type scheme for real addition and multiplication are:

$$\begin{array}{ll} \texttt{add}: & \forall\ d. \quad \texttt{real}\ d \to \texttt{real}\ d \to \texttt{real}\ d \\ \texttt{mul}: & \forall\ d_1\ d_2.\ \texttt{real}\ d_1 \to \texttt{real}\ d_2 \to \texttt{real}\ d_1 d_2 \end{array}$$

With very few modifications, a Hindley-Milner like type checker is able to statically ensure that computations are homogeneous.

Hughes, Pareto, and Sabry [HPS96] adapted this approach to *sized types*, that are recursive algebraic data-types whose height can be statically bounded. Their base types consist of type constructors such as streams and indexes are linear integer expressions completed with infinity.

Even closer to our subject, Zenger [Zen97] designed *indexed types*, a type system that describes the sizes of arrays. Its index language is made of complex multivariate polynomials. Whereas this is too general for sizes (we expect a size to be a positive integer), it suffices to check size consistency i.e., that sizes match.

For index languages that are only structural, i.e., index equality amounts to first order unification, the type family approach is similar to *phantom types*, a common programming trick introduced by Leijen and Meijer [LM99], where extra type variables, that are not used in the definition of the data variants, capture additional properties. Type families preserve a fundamental property of the Hindley-Milner type discipline, the semantics of such languages remains type-erasable, i.e., terms are independent of types.

Sizes as typing properties: (ii) refinement types Instead of duplicating base types in sibling versions, refinement types add some conditions to restrict the domains of types. This term was first introduced by Freeman and Pfenning [FP91] to denote *data-sort* refinements, i.e., the selection of some constructors of an algebraic data-type. Rushby, Owre, and Shankar [ROS98] later used the same term with its now widespread meaning of *predicate* refinement. They are usually represented with the syntax:

$$\tau = \alpha \mid \{\, x : \mathcal{B} \mid P(x) \,\} \mid \tau \to \tau$$

where $\mathcal{B}$ is a base type and P is a predicate that selects some of the values of $\mathcal{B}$. Contrary to type families, refinement types induce a natural notion of sub-typing: a value with refined type $\{\, x : \mathcal{B} \mid P(x) \,\}$ is also of type $\{\, x : \mathcal{B} \mid Q(x) \,\}$ if the implication $P \implies Q$ holds.

To express complex conditions, the predicates may use program variables. The idea of types that depend on values dates back to AUTOMATH [Bru70], a language designed to formalize and check mathematics. Its relevance for programming languages was studied by Martin-Löf [Mar84] and *dependent types* serve as a basis for several type systems and languages such as NUPRL [Con83], LF [HHP87] or the calculus of constructions [CH88]. In dependent type systems, the capture of values in types is introduced by the *dependent product* construct — $\Pi\, x : A.\ B(x)$ — where the variable x, of type A may appear in type B. This construct generalizes the simple type $A \to B$, which is a special case of $\Pi\, x : A.\ B(x)$ where x does not occur in B. Without restrictions, sub-typing depends on arbitrary predicate implications, which causes type checking to be undecidable.

The λ^H type system of Flanagan [Fla06] allows such general refinement types. It delegates the solving of implications to an external procedure (such as a theorem prover) that is allowed to fail. If so, the type system defers the unproved verifications to runtime checks. While such a system allows a best effort approach to static verification, the quality of the generated code (e.g., the number of checks that remain) depends on an external solver.

Xi and Pfenning [XP99] propose *dependent ML* (DML), a conservative extension of the ML type system with dependent types. To ensure array correctness, they delineated restrictions on the predicate language to allow static verification [XP98]. More recently, the Futhark [HE21] and Dex [Pas+21] languages propose partial verification of array sizes using a more restricted size language. We review these constraints below.

Dependent types are not limited to size checking. Trojahner and Grelck [TG09] studied the verification of rank generic programs with dependent types for a language inspired by *Single assignment C* [Sch03]. Rank generic functions require a two-level dependency. The type of an array depends on its shape, that is a vector whose length depends on the rank. A similar description of array shape was proposed by Slepak, Shivers, and Manolios [SSM14] for APL.

Relaxing the static assumption The static description of sizes is limited to shapely operations. This excludes, for instance, filtering an array on a condition, since the size of the result would depend on the runtime values of the elements. Some programming languages simply ban these operations, e.g., Scade, while others fall back on unsized, and thus dynamically checked, data structures, e.g., Futhark.

A midway solution exists: by using symbolic sizes, the type system may account for sizes that are not known at compile-time while still allowing to keep track of local *size relations*. In a dependently typed setting, such local abstract sizes are provided by *dependent sums* — $\Sigma\, x : A.\ B(x)$ — [XP99], that generalizes simply typed pairs: the type of the second component depends on the value of the first one.

A similar mechanism is possible in a type system without dependent types. In order to hide implementation details of a type in a module or a package, Mitchell and Plotkin [MP88] introduced existentially and universally quantified types. From this starting point, Jones [Jon97] proposed a type system where data-types may have quantified components. For type families, whose indexes are preferably generalized with polymorphism, these techniques may apply to represent dynamically indexed objects, e.g., an array whose size is existentially quantified. This extension is strongly related to *higher order polymorphism*, studied by Peyton Jones et al. [Pey+07], i.e., the ability to nest quantification in types.

1.2.4 Inference in Extended Type Systems

Explicitly typed languages are verbose and type families or refinement types would need even longer type annotations. Inference is mandatory to alleviate the annotation burden. To this end, the ML type system hit a sweet spot, complete inference is decidable. The situation is less favorable for extended type systems since it strongly depends on the expressive power of indexes or predicates.

Stratified inference Type reconstruction for extended type systems has been thoroughly studied. Most of the proposed algorithms follows a stratified approach that consists in:

1. Performing Hindley-Milner inference (first order unification) to build type skeletons with the usual soundness and completeness guarantees.
2. Extracting from the simply typed programs constraints over refinements or indexes. These constraints are respectively implications and equalities.
3. Solving the extracted constraints with dedicated tools (solver, theorem prover or dedicated algorithm).

With unrestricted refinements, Knowles and Flanagan [KF07] circumvented the undecidability obstacle for λ^H by defining type reconstruction as a *typeability-preserving* process, i.e., the existence of a valuation of the variables that makes the program well-typed is preserved. In essence, the reconstruction process propagates the postconditions along the data-flow graph so as to build the most precise predicate for each program point. No refinements are inferred per se, the existing ones are combined and quantified.

Rondon, Kawaguchi, and Jhala [RKJ08] introduced Liquid-Types, a general framework to handle refinement inference. Assuming decidability properties over a subset of the considered predicate language, they point out the places where refinements should be restricted so as to ensure inference decidability.

Most of the domain specific type systems, in particular for array checking, have explored several middle grounds between the expressive power and the inference and verification perspectives. They

all build on a total separation of the language of indexes or predicates. This allows to select the index language according to some relevant restrictions.

The choice of an index language To ensure decidability, several extended types systems are restricted to predicates that generate *linear* constraints (inequalities). Sized types [HPS96], DML [XP98] or the type system inspired from SaC by Trojahner and Grelck [TG09] are examples of type systems that rely on SMT solvers to treat the generated linear integer constraints. The context of dimension types [Ken94] is even more favorable. Because dimensions may only be combined by products, constraints amount to *linear equalities* on the exponents. Such systems may be solved efficiently.

The indexed types of Zenger [Zen97] describe sizes with *complex multivariate polynomials*. While surprising, this size language induces polynomial constraints whose implication is decidable, using for instance Gröbner bases. In a similar way to inference for λ^H [KF07], this type reconstruction does not infer sizes, but does check that the explicit sizes are consistent.

For the Futhark [HE21] and Dex [Pas+21] languages, the type system only handles *symbolic* sizes, i.e., either variables or constants. This rudimentary language is sufficient to describe size-preserving operations, e.g., `map`, that are the most used ones and avoids the need for linear solvers in the typing process. These type systems pursue best-effort checking. The operations that cannot be described precisely, e.g., array concatenation, are not checked statically.

1.3 Compilation of Data Processing Applications

Data processing applications tend to treat large amounts of data using comparatively small algorithms. In this context, the performance depends not only on the complexity of algorithms (e.g., a Fast Fourier Transform), but more importantly on their implementations, whose performance is mostly driven by two entwined aspects:

1. *Memory.* Because of their size, array data cannot be held in processor registers: it is stored in memory, transferred to the processor for computations and sent back to the memory for future use. The communication channel between the processor and the memory, named the *Von Neumann bottleneck* by Backus [Bac78], suffers two limitations: (i) the memory frequency is several orders of magnitude lower than the one of processors and (ii) the communication channels (buses) have limited bandwidth. Hence, if computation needs exceed available throughput, the processor must wait for data transfers to complete.

2. *Parallelism.* Data processing applications are described as *embarrassingly parallel*, a term first used in a book by Moler [Mol86]: they feature repetitive treatments whose parts are often independent. Hence they can be computed at the same time. This parallelism may operate at several levels. *Coarse grain* parallelism denotes algorithms that are split into large independent tasks while *fine grain* parallelism designates programs with locally parallel instructions. Parallel implementations need additional synchronizations to orchestrate the parallel computations.

Tuning parallelism without clever memory management has less chance of improving performanc. This remark is especially relevant for GPUs whose memory capabilities have been tailored for specific access schemes, known as *coalescing accesses*: throughput is maximal when the units that run in parallel read or write consecutive addresses.

Optimizing memory and parallelism We designate as *implementation optimizations* program transformations that respect the data dependencies. Multiple implementations perform the same operations. Their differences lie in the choice of computation order and the storage of intermediate values. They are compared according to the following criteria: (i) *limit storage* so that less communications with the memory are necessary and (ii) *increase data locality*, i.e., memory accesses that are close in time are also close in space, to take advantage of hardware capabilities such as caches. Here are some commonly used strategies:

- *Computation duplication.* Storing the results of relatively cheap calculations may have higher cost than recomputing them when needed. Duplication thus limits intermediate storage and favors coarse grain parallelism by making parts independent.

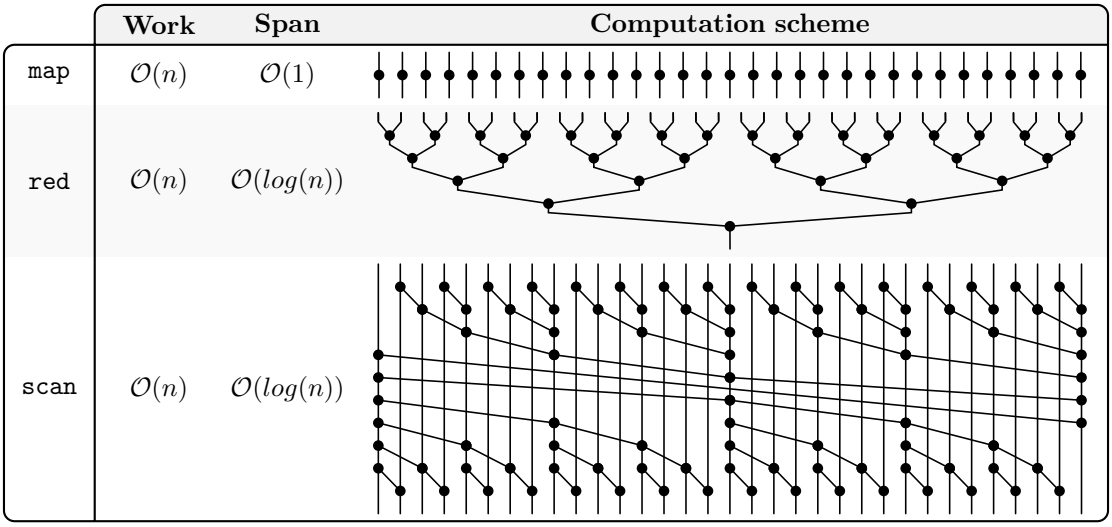


Figure 1.9: *Parallel implementations of second order array iterators (SOACs)*

- *Loop tiling.* The iteration space may be split into multiple dimensions. Together with loop reordering, this helps improve data locality. By accessing smaller chunks of data, it enhances cache reuses.
- *Loop fusion.* By aggregating multiple repeated computations into one, the storage of intermediate results becomes unnecessary. This limits communications with memory, at the cost of locality because fused operations may need more scattered data than do atomic ones.
- *Memory reuse.* Reordering computations may allow to write some result in place of older values that are no longer needed. When possible, data locality benefits from sharing.

By their nature, these memory optimizations are well described in imperative low-level languages where programs are expressed by a sequence of instructions that modify global storage. However, the algorithm and implementation details are intricate in these descriptions. For instance, Thomas, Mullin, and Świrydowicz [TMŚ21] recently improved the performances of cache-optimized matrix multiplication originally proposed by Van Zee and Van de Geijn [VV15]. This implementation consists in six nested loops instead of three in the naive algorithm. This low-level approach has two major drawbacks:

- *Verification.* Optimized programs are hard to understand and even more so to prove, since the structure of the algorithms is obscured by the implementation details.
- *Portability.* Each implementation is tailored for a specific architecture. To be used elsewhere, it may require at least tuning some parameters, e.g., tile sizes, and at worst rewriting some parts of the program.

Declarative languages for data processing In declarative languages, computation is driven by data dependencies. The program does not specify memory allocation and scheduling, and the compiler is in charge of finding an imperative implementation that is consistent with the dependencies. This flexibility is particularly valuable to extract parallelism. Contrary to imperative programming languages where instructions are connected by an invisible link — the memory, — all dependencies are explicit in a data-flow language.

In functional languages, second-order array combinators (SOACs), as named by Henriksen et al. [Hen+17], are typical examples of constructs that might be implemented in parallel. As an exception to the dependency preservation principle we stated above, parallel implementations of iterators (fold, reduce) rely on a few semantic equivalences, mostly the associativity of scalar operations and occasionally commutativity.

Such various parallel implementations of iteration schemes were first studied in the context of circuit design [She85] in which the various computations schemes are compared with several measurements. Among them, (i) the *work* gives the asymptotic number of computations, while the *span* designates the depth of the circuit, i.e., the length of the longest computation dependence

chain. In a sequential implementation, both measurements coincide. Figure 1.9 depicts the main iterators and one of their usual parallel implementations. In this representation of computation schemes, the horizontally aligned nodes depict computations that might run in parallel.

- `map` — The point-to-point application of a function is immediately parallelizable without further assumptions, because all the computations are independent.
- `red` — Reduction is a weaker version of the `fold` iterator where the order is not specified, hence the iterated function must be associative. This hypothesis allows to compute partial results in parallel before combining them.
- `scan` — This iterator computes all the partial reductions. With a similar associativity assumption, this operation might be implemented in parallel in multiple ways, as precisely reviewed by Hinze [Hin04]. We picked here the Brent-Kung circuit, that sacrifices the optimal span (although still being logarithmic) to use fewer operations and limit the *fan-out*, i.e., the number of concurrent accesses to a single value, another important criterium since these reads are performed by distinct computation units, which may have limited or costly accesses to shared memory.

For floating-point computations, associativity is often abusively assumed: rounding errors make scalar operations non associative. These errors may be inconsequential for some low-precision data-processing applications, but stability and robustness must be assessed for each specific process.

1.3.1 Specifying Memory in a Declarative Language

The management of memory is a tedious task. To help, programming languages provide various memory models. As a common base, most of them rely on a separation between a *stack* and a *heap*. The former stores values that are needed temporarily (from a function call point of view) and is statically managed by the compiler. The latter is used for data with longer life-time. In low-level languages such as C, it must be managed manually. Finding a correct orchestration of allocations and disallocations is error prone. Freeing data too early leads to *dangling pointers*, i.e. reading corrupted values, while forgetting to free data results in *memory leaks*, i.e. allocated space that cannot be used anymore.

The Rust language [KN18] proposes an allocation discipline built on a notion of *reference ownership*. It aims at expressing the common allocation schemes so that the compiler can check their correctness.

Higher level languages such as Java help the programmer further by alleviating the need of recycling memory: at runtime, a companion program, the garbage collector, is enmeshed with the user code to keep track of references and free memory appropriately. Instead of a uniform address space as in C, memory is represented as a set of objects, among which certain might be mutated, e.g., references in Standard ML or OCaml.

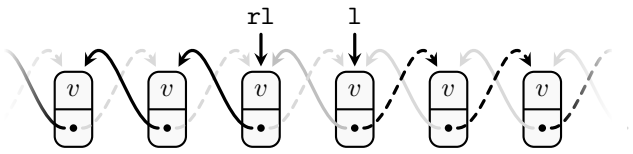
Declarative languages such as Haskell, Sisal or Scade even obviate the need for a memory model: programs describe computations on immutable values, independently of any memory storage. This abstraction is key for formal reasoning because the semantics of programs stems from definitions instead of incremental modifications of a global memory. These high-level programming languages carry the choice of memory management over to the compiler. For the best, because no memory errors can occur, and for the worst because the allocation is out of the programmer's control.

Two flavors of memory bloats... In declarative languages, memory allocation must ensure that computed values stay unaltered as long as they are needed. Two values that are needed in subsequent computations must be stored in different, more precisely *non-aliasing*, memory locations. The memory-concerned programmer must play with these principles and hope for optimizations to succeed so as to obtain effective implementations. Two causes of memory inefficiency are worth mentioning.

In imperative languages, large data structures such as arrays are commonly *mutated*. Moreover, some of these changes affect only a tiny part of the large structure. In a declarative setting, data-structures are persistent. Local changes are provided by *functional updates*, that implement a copy-and-modify operation. Compiled naively, its complexity is drastically worse, both in terms of memory and execution time. Instead of writing the modified bits, the whole data structure must be duplicated. More clever compilation strategies uses shallow bindings [Bak91] or restricted forms

```
let rec rev rl l = match l with
  |   []   -> rl
  | h :: t -> rev (h :: rl) t
let rev l = rev [] l
```

(a) Tail recursive implementation in OCaml



(b) Pointer flipping

Figure 1.10: *List reversal*

of persistence [CF08] if applications permit it. However, these approaches need dynamic memory management. In a statically allocated program, data-structures may be modified in place if the old values are known not to be used in subsequent computations. For instance, the Sisal compiler implements an update-in-place analysis [Fit93] for optimizing these cases.

In functional languages, algebraic data-types allow to represent complex data-structures such as lists or trees. They are implemented as heap allocated records that store information about the structure of the data. These descriptors are allocated when building new values of the given type. Upon their destruction, e.g., pattern matching, these constructors are freed, if no longer used. Lots of algorithms on algebraic data-types consist in traversing a structure and building an equivalently structured result with local modifications. Compiled unwisely, this results in freeing an old constructor and allocating a new one for each sub-structure of the data, relying on the garbage collector to clear the memory. In some cases, the descriptors could simply be *reused*, e.g., to build the resulting data. Figure 1.10 illustrates how a linked list may be reversed by making pointers to the next links (depicted with dashed lines) point on the previous ones (depicted with solid lines), instead of building new links.

... With a common remedy In order to turn these potential optimizations into constraints that the programmer may require, Wadler [Wad90] proposed *linear types*, a type discipline based on the linear logic of Girard [Gir87]. This type system ensures *no duplication* and *no discarding* properties, i.e., a linearly typed value cannot be dropped or used multiple times.

For data-structures, this linearity condition is overly restrictive. So as to ensure that the modified objects are not used elsewhere, a *semi-linearity* property suffices. Values may be read multiple times as long as they are consumed once, after all the pending reads. Semi-linear type systems have been used to compile functional updates into in-place modifications, both in the context of Lustre [Gér+12] and Futhark [Hen+17].

A general solution for the second issue has been proposed recently by Lorenzen, Leijen, Swierstra, et al. [LLS+23]. They delineate some linear typing annotations that allow destructive pattern matching, which may reuse the data that is matched.

An imperative oasis in a functional desert? The needs of imperative features in purely functional languages go beyond memory optimization, in particular communications with the outside world, e.g., the file system, are inherently imperative. Wadler [Wad95] proposed a game-changing approach to impure hooks: encapsulating the imperative parts in *monads*, that are abstract representations of sequential interpreters. They consist in two operations: `return` and `bind`. Intuitively, the former produces a result, while the latter chains computations. For arrays, the so called *state monad* gives in addition some functions to access and modify the internal state of the interpreter.

The monadic approach provides an elegant and relatively flexible way to alter the interpretation mechanisms of a purely functional program, e.g., by adding some imperative features. But this comes at a substantial price. All the parts that are computed within the monad are sequentialized hence loosing a major strength of the declarative style, its ability to expose parallelism for free.

Aggregation forecast While designing the Sisal language and its compiler, Gaudiot et al. [Gau+97] identified an additional source of memory inefficiency that is intrinsic to declarative languages. The data-flow style requires defining the parts of compound values before aggregating them. Without care, this latter phase amounts to moving the freshly computed parts to their final location, at the cost of memory footprint and execution time. As a mitigation, they designed a *build-in-place* optimization pass. It consists in separating the allocation of data from their definition, so that the sub-values might be computed at their final location, thus eliminating the need of copies to build the complete result.

Such an optimization is crucial for the manipulation of large data such as arrays. For instance, array padding, either on the left or on the right, is a common operation. To avoid extra costs, the to-be-padded array should be constructed inside a larger array, whose remaining parts are filled with the padding values. We will illustrate a similar operation in Section 1.5.

1.3.2 (Re)ordering Computations

Informally, the structure of data-intensive computations is characterized by a discrete iteration domain (the indexes for which computations occur) and dependencies between the iteration points that restrict the possible orders of computation. Declarative languages describe data dependencies, leaving implementation choices to compilers. The structure of computation is thus directly available. For imperative languages, this structure must be retrieved from an obfuscating sequential implementation with fragile analyses. This is a mandatory step for profound code transformations.

For such an abstract representation of algorithms, the selection of an efficient low-level implementation strongly depends on the target features (caches, parallelism, vectorized instructions, *etc*). Among the pioneering work, Lamport [Lam74] proposed the *wavefront* or *hyperplane* method, depicted in Figure 1.11: find in the multidimensional domain of computation a direction to iterate along that is aligned with the dependencies, that is, no dependencies must flow backward. Repeat the process in the hyperplanes defined by this direction that intersect the iteration domain.

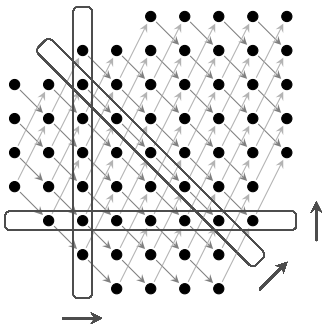


Figure 1.11: *The wavefront method: computations (dots) are partially ordered by dependencies (arrows). Rectangles and thick arrows depict possible hyperplanes and directions of iteration*

We now review some of the approaches to such iteration transformations, with various levels of abstraction and automation.

A case study: (almost) naive matrix multiplication Numerical linear algebra is arguably the most used set of array operations. They serve as basic blocks to build solutions to problems that have a linear representation. The famous LAPACK library (*Linear Algebra Package*) [Dem91] was originally developed for the FORTRAN programming language. It aims at providing a portable interface to highly efficient implementations of linear operations such as factorization and decomposition. It builds on BLAS (*Basic Linear Algebra Subprograms*), a specification of a set of low-level linear algebra operations. Among them, matrix multiplication is one of the most studied kernels. Its mathematical definition is:

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

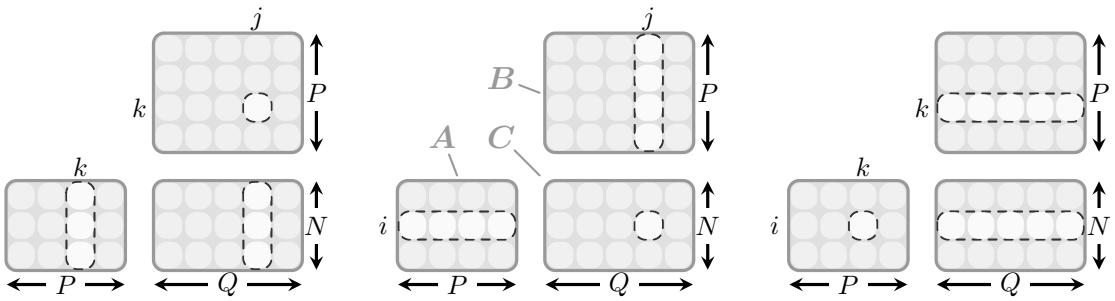
It turns into an immediate algorithm. Starting from an initial matrix of zeros, iterate over the three dimensions to sum each product to the appropriate element of the result. A naive implementation of a matrix product consists in three nested loops. Two of them iterate on the elements of the resulting matrix, while the other iterates on the common dimension to sum the products. Despite its $\mathcal{O}(n^3)$ complexity for square matrices, clever choices of computation order lead to implementations that largely outperform theoretically better algorithms[9] for relatively small matrices.

The most efficient algorithms strongly depend on hardware capabilities, so as to balance memory accesses and computations. Without getting into the art of optimizing cache usage, let us analyse the differences between the different iteration orders. Figure 1.12 illustrates the accesses that occur in the innermost loop for each of the three options.

The center one (Figure 1.12b) depicts iteration along the common dimension of the arguments. The inner-most loop computes the scalar product between the related row and column. At each iteration, it reads and writes the same element of the C matrix, hence the inner loop cannot be executed in parallel.

In the two other cases, the element that is accessed multiple times is only read. This allows the inner loop, that writes different elements at each step, to be implemented in parallel. Moreover,

[9] Complex algebraic properties allow smart factorizations, saving some products and leading to an $\mathcal{O}(n^\omega)$ complexity with best ω known smaller than 2.4. [Kar95]



(a) Iterate on $i < N$ *(b) Iterate on $k < P$* *(c) Iterate on $j < Q$*

Figure 1.12: *Accesses in innermost loop for computing $C = A \times B$*

depending on how matrices are stored, either in row-major or column-major order, one of the two options features accesses to adjacent memory locations. This kind of access, called *strided* in the context of GPUs, benefits from the highest memory bandwidth in GPUs and such contiguous data and computations are candidates for the vector instructions that are available on certain CPUs.

The polyhedral model The wavefront method sketched above laid the foundations of the *polyhedral model*, that arose notably with the ALPHA [QRW94] language for programming systolic arrays. It aimed at representing in a compact way massively parallel computation [Wil93]: polyhedra are subsets of the integer k-tuple ($\mathbb{Z}^k$) defined by a set of linear inequalities. Feautrier [Fea91] showed how such a polyhedral description can represent data-flow dependencies in array computations.

At the same time, loop optimization techniques were receiving much attention: Pugh [Pug91] proposed to used integer programming as a unified description of loop transformations (interchange, fusion, skewing, ...). The general skeleton of polyhedral model optimizing techniques is reviewed by Bastoul et al. [Bas+03]. It consists of:

1. Identifying the *static control parts* and their context, i.e., the parts of programs that can be represented with polyhedra. This mainly imposes the control code (loop bounds) to be statically computable and index relations to be linear.

2. Finding and optimizing a *schedule*, also called *scattering function*, i.e., a partial order on the iteration space so that data-dependencies are respected. Schedules are optimized with various transformations, such as splitting or fusing dimensions.

3. Scanning the final polyhedra with their scheduling to generate imperative or parallel code [Bas04]. This step is crucial for the performance of the resulting code, in order to avoid useless control structures such as conditionals in the loops.

Separating algorithms from schedules The polyhedral model helps to produce complex implementations from simple descriptions. However, the choice of implementation is not under the programmer's control and is subject to the strategies of the compiler. To accommodate a simple (mathematical) description of image processing algorithms with a precise specification of their implementation, Ragan-Kelley [Rag14] designed the HALIDE language, a domain specific language embedded in C++ (DSEL) that draws a complete separation between *algorithms* and *schedules*.

In HALIDE, arrays are described as functions from an index space whose dimensions are named. Algorithms are expressed by purely functional, extensional definitions of arrays, i.e., a mapping from named indexes to values. These names are crucial. Once the declarative algorithm is complete, they allow to specify the implementation with schedule directives. The directives determine for instance which dimensions should be traversed sequentially or in parallel, which dimensions should be split (tiling), which part of the intermediate results must be stored or recomputed, *etc.*

This high-level description of the implementation selects the transformations to apply to a direct translation of the algorithm, without having to enter the details of memory management, index computations or synchronization. The clear distinction between algorithms and schedules paves the way to several developments, notably:

- *Automatic schedule space exploration.* The schedule language provides a formal way to explore possible implementations. Because finding the best schedule is a tedious task, Mullapudi et al. [Mul+16] propose a cost model for auto-tuning the schedule. In the deep-learning context, Chen et al. [Che+18] propose TVM, a data-processing language with a similar algorithm/schedule separation, whose schedule exploration is driven by measurements of the performance of various implementations, to train a machine learning model of schedule costs.

- *Verified tensor operations.* The schedule language also identifies the atomic transformations to be performed on data-processing algorithms. They are assumed to preserve the semantics, so that correct algorithms may not lead to incorrect implementations. To give this claim a solid base, Clément [Clé22] proposed an automated validation method for such operations that relies on the generation of annotations to trace the transformations and prove their correctness.

Iterator fusion Among the structural transformations that the above techniques allow for, loop fusion consists in gathering the iterated tasks of consecutive loops into a single one. In purely functional programming languages, where array operations are primarily described with iterators, this transformation is crucial. For instance, it avoids building arrays for every single `map`.

The Futhark language, designed by [Hen+17], targets GPUs kernels from a purely functional description. Instead of relying on low-level, data-dependency-based loop transformations, optimizations such as loop fusion are performed by high-level rewrite rules. For instance, composition and point-to-point application commute:

$$(\texttt{map}\ g) \circ (\texttt{map}\ f) = \texttt{map}\ (g \circ f)$$

Along with dedicated rules for the other iterators (`reduce`, `scan`), the rewrite system allows to build large kernels and hence maximize coarse grain parallelism, out of sequences of parallel array operations, i.e., fine-grain parallelism.

Moreover, instead of describing compilation pipelines at the level of elements, the Futhark language provides a set of *streaming* iterators that process chunks of elements. They allow for tuned sequential implementations that the compiler is free to use taking into account the available parallelism, by tiling computations for instance.

Compared to the aforementioned techniques, iterator rewriting gives a more abstract level for transforming algorithms, whose atomic operations are easily verifiable. However, the growing family of iterators induces a combinatorial explosion of the set of rewrite rules. This is mitigated by using some generic, hence less precise, representations of transformations when no rewrite rules apply.

Bidirectional fusion In Obsidian, a DSEL embedded in Haskell proposed by Svensson, Sheeran, and Claessen [SSC08] for GPU programming, iterator fusion is achieved differently. Inspired by Lava [Bje+98], compilation is done by evaluating the Haskell program describing the kernel. To this end, arrays are represented with a dedicated data type, since array computations should be distributed. Putting aside the sizes, that are actually part of array data-types, and assuming an index type `idx` similar to integers, the original representation of arrays, later named *pull arrays*, that we denote with a $\leftarrow$ subscript, comprise functions from indexes to elements, i.e., values of type $\texttt{idx} \to \tau$. This representation allows an easy representation of the usual array operations and provides fusion for free, by only composing the functions:

$$\texttt{map}_{\leftarrow}\ f\ A : \quad \lambda i.\ f\ (A\ i)$$

$$\texttt{concat}_{\leftarrow}\ A\ B : \quad \lambda i.\ \texttt{if}\ i < n\ \texttt{then}\ A\ i\ \texttt{else}\ B\ (i - n)$$

GPU kernels are then made from the bodies of these array functions. At runtime, kernels are evaluated for each index, so as to build the resulting array in memory. In the $\texttt{concat}_{\leftarrow}$ example, where we assume n to be the size of A, the result is unsatisfying. It introduces a conditional inside the computation of each element, a performance-killer for GPUs.

To circumvent this weakness of the demand-driven representation, Claessen, Sheeran, and Svensson [CSS12] supplemented it with *push arrays*, denoted with a $\rightarrow$ subscript, a data-driven representation of arrays as functions from a continuation that expects an index and the associated element, i.e., values of type $(\texttt{idx} \to \tau \to \texttt{unit}) \to \texttt{unit}$. Intuitively, the continuation may perform

some additional computations before it ultimately writes the resulting element in memory. The above pull version of array operations are rewritten in the push form as:

$$\texttt{map}_{\rightarrow}\ f\ A: \qquad \lambda k.\ A\ (\lambda i.\ \lambda e.\ k\ i\ (f\ e))$$

$$\texttt{concat}_{\rightarrow}\ A\ B: \qquad \lambda k.\ A\ k\texttt{;}\ B\ (\lambda i.\ \lambda e.\ k\ (i+n)\ e)$$

The extraction of kernels amounts to applying a write continuation to the push arrays. The `;` operator used in the definition of $\texttt{concat}_{\rightarrow}$ denotes sequential composition (that might be parallel under conditions). The concatenation is implemented by passing an altered continuation to B. The index of the element is shifted.[10] The compilation perspectives are much better here. This form generates a list of kernels that compute separate parts of the result, i.e., A and B, which alleviates the need of conditionals in the kernel.

This brilliant idea conciliates simplicity and efficiency by acknowledging that some operations are intrinsically data-driven while others are demand-driven. Our memory-aware language largely draws from this dichotomy.

1.3.3 Arrays in Scade and their Compilation

Purely functional languages ease the verification task by shrinking the distance between programs and their mathematical specifications. The main value of the Scade language and the related tools (e.g., compiler, design verifier, coverage assistant) is to allow most of the certification process to be conducted at the model level and benefit from the resulting certification credits for the generated code. To this end, the compiler is certified to the highest security standards.

The predictability constraints of safety critical embedded software impose drastic limitations on the use of memory. Programs must run with a statically known bounded response time and memory footprint. In this context, dynamic allocation, either garbage collector assisted or manual, is prohibited because it is hard to formalize and verify. This results in a somehow paradoxical observation. Scade programs are guaranteed to run in statically bounded time and space, but the programmer has little way to control these bounds. A robustness issue arises. Small changes may lead to significantly different implementations with unrelated performance.

A vectorizable vector-matrix product To illustrate the lack of control on the generated code, let us implement the third version of matrix multiplication depicted in Figure 1.12c with the current version of the language and its compiler, KCG. For brevity, we drop the iteration on A rows, effectively implementing a vector-matrix multiplication. We recall the mathematical definition of the vector-matrix product in Figure 1.13a. The skeleton of a vectorizable implementation is given in Figure 1.13b. The consecutive iterations of the inner loop access contiguous elements of the arrays (both A and u), assuming a row-major storage, as Figure 1.13c illustrates.

This algorithm can be implemented in Scade using the classical array iterators `map` and `fold`, as shown in Figure 1.13d. Figure 1.13e renders a slightly pared down version of the generated code, which features the expected iteration order. However, the array in which accumulation is performed (v) is not modified in place: a local array has been allocated (1) and the previous value of v is copied in the temporary value (2) at each outer iteration. The expected gain of a possibly vectorizable implementation is spoiled by the extra local array and the associated copy.

The weaknesses of copy elimination The insufficiently optimized code for the vectorizable vector-matrix product is a consequence of the source-to-source compilation process. As sketched in Section 1.1.3, programs are normalized and scheduled in a declarative representation — thus independently of memory — so as to obtain a form that can be straightforwardly translated into a simple object-based language. Memory optimizations operate on this imperative form, aiming to eliminate unnecessary variables and reuse memory.

Whereas this is simple for scalar values, i.e., check that they are not used in the following instructions, the task is much more involved for arrays, because they can be partly read or written. To eliminate an intermediate array, the read and write accesses must combine nicely so that no elements of the first array are needed after the corresponding element of the second one is written. The following table illustrates both situations:

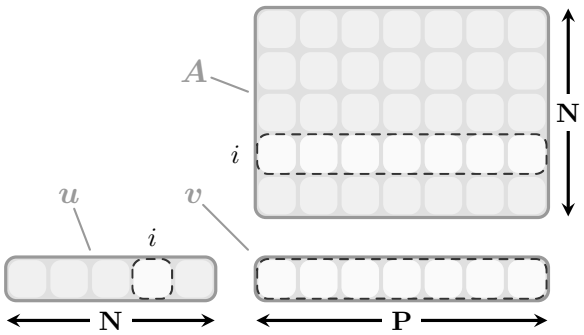
[10] The η-expanded version of the continuation emphasises its type: a function of an index and an element.

$$v_j = \sum_{i=1}^{N} u_i A_{ij}$$

(a) Formal definition.

```
for (int j = 0; j < P; j++)
  v[j] = 0;
for (int i = 0; i < N; i++)
  for (int j = 0; j < P; j++)
    v[j] += u[i] * A[i][j];
```

(b) Expected imperative pseudo-code.



(c) Access scheme for the inner loop ($j < P$).

```
-- The accumulation operation
function sum_prod (s, a, b: int32)
returns (t: int32)
  t = s + a * b;

-- Point-wise application of the accumulation operation
function vec_mat_i «P» (v: int32^P; ui: int32; Ai: int32^P)
returns (w: int32^P)
  w = (map sum_prod «P») (v, ui^P, Ai);

-- Accumulation starting with an array of 0
function vec_mat «N,P» (u: int32^N; A: int32^P^N)
returns (v: int32^P)
  v = (fold (vec_mat_i «P») «N») (0^P, u, A);

-- Main function, with concrete sizes for code generation
function main (u: int32^4; A: int32^5^4)
returns (v: int32^5)
  v = (vec_mat «4,5») (u,A);
```

(d) A SCADE implementation. Size parameters — <<N>> — are pretty-printed «N».

```
void main (inC_main *inC, outC_main *outC) {
  array_int32_5 acc;                        // Local array: extra memory footprint    (1)
  kcg_size i;
  kcg_size j;

  for (i = 0; i < 5; i++)
    outC->v[i] = kcg_lit_int32(0);

  for (i = 0; i < 4; i++) {
    kcg_copy_array_int32_5 (&acc, &outC->v); // Array copy: extra execution time      (2)
    for (j = 0; j < 5; j++)
      outC->v[j] = acc[j] + inC->u[i] * inC->A[i][j];
  }
}
```

(e) Structure of the generated C code.

Figure 1.13: *Vectorizable vector-matrix multiplication. Consecutive accesses of the inner loop access contiguous elements. So as to obtain the desired nesting of loops, the accumulator in the SCADE program is an array. Despite the inlining of iterated nodes, the conservative optimizations of the code generator (KCG) do not eliminate the temporary accumulator.*

	Increment	Reverse
Declarative form (SCADE)	`B = (map inc «N») (A);`	`B = reverse (A);`
Imperative form (C)	`for (int i = 0; i < N; i++)` `  B[i] = inc (A[i]);`	`for (int i = 0; i < N; i++)` `  B[i] = A[N-1-i];`

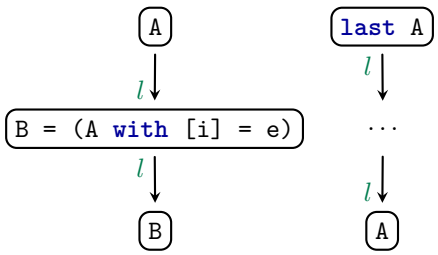
For the increment case, the same element is read and written at each iteration. Thus, `A` and `B` may be allocated at the same memory location. By contrast, such sharing is impossible for the reverse case: the first half of the iterations would overwrite the elements that are needed subsequently. With an imperative form, deciding whether sharing is possible or not requires a thorough analysis of indexes that KCG optimizations do not support, mainly for safety reasons.

These missed optimization opportunities arises mainly in three situations, where potentially superfluous copies are inserted and cannot be eliminated afterwards:

- *Array updates.* Local modifications of arrays are expressed in SCADE with functional updates — `A with [i] = e` — that can be implemented as an in-place update if the old array is not needed afterward.
- *Accumulators.* The `fold` iterator must store the produced values of accumulators at the location of the previous ones. The general compilation scheme introduces new variables to hold the previous values of accumulators, unless optimizations determine that they are useless. This is the reason for the extra copy in the vector-matrix multiplication.
- *Registers.* The normalization pass replaces expressions `pre e` by a fresh variable `lx`, and introduces two equations — `lx = pre x;` and `x = e;` — where `x` is fresh too. In a similar way as for accumulators, the first one introduces a copy of the old value into a fresh location, independent of the state, represented by `x`.

The compilation process would benefit from (i) improving the robustness of optimizations and (ii) allowing some control of the generated code from the model. Our work focuses on the second direction. For SCADE, the above situations suggest a possible form of implementation specification. The memory concerned programmer could tag the constructs (array updates, registers, *etc*) that should not introduce copies. The compiler would then check that these directives are consistent with the data-dependencies and generate code that complies with them.

A closer look at the above situations exposes two kinds of memory sharing constraints. (i) For array updates, the constraint follows the data-flow, i.e., the location of the result is the location of an argument, as depicted by the data-flow graph on the left. The constraint is *local*. (ii) Accumulators or registers introduce *global* constraints, i.e., they relate parts of the program that are not necessarily connected by data-flow dependencies, as depicted on the right.



While (semi-)linear type systems describe local constraints well, their use for the global ones is more contrived. This observation is one of the starting points of a more precise description of memory locations and constraints. Furthermore, the similarity between accumulators and registers is a compelling reason for the design of a new iteration construct in which accumulation leans on the state: a no-copy state will allow a no-copy accumulation.

1.4 Contributions

We propose a *Memory-Aware Declarative Language*, abbreviated MADL, to bridge the gap between a data-flow description of stream functions and their imperative implementations. Although it would serve as an intermediate language for the compilation of SCADE, it has been designed as an independent language. It does not rely on invariants that would be inherited from a higher level form. Instead, we provide static checks to establish the correctness of programs.

The compilation process of SCADE favors scheduling over allocation. The compiler introduces memory after having chosen the order of computations using the data-flow dependencies. Hence

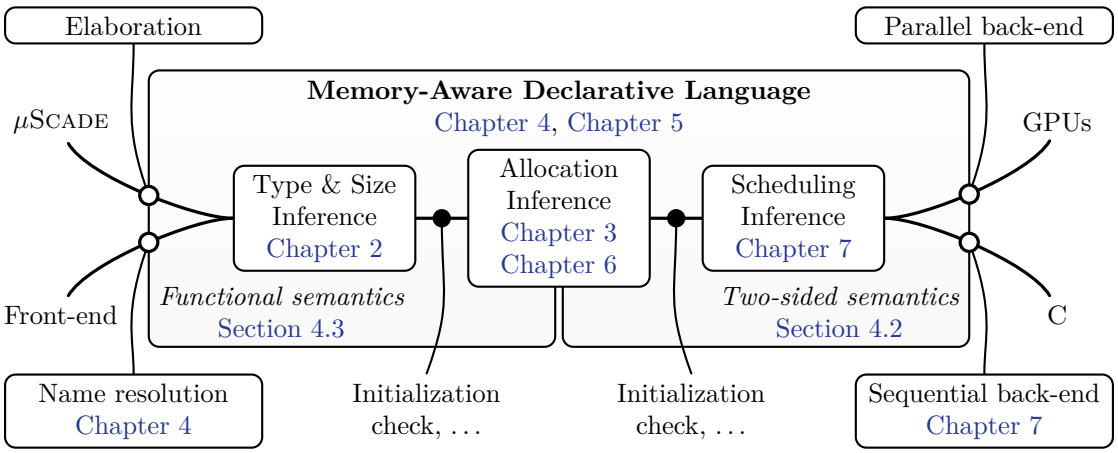
allocation concerns are handled in an imperative form. To specify memory from the declarative program, our proposal does the converse. Data-flow dependencies and allocation coexist as two dimensions of a single object — the *fully located* MADL program — before scheduling.

Our proposal focuses on language design rather than compilation. MADL programs provide a declarative description of algorithms and a precise specification of their memory implementation. Hence, the role of the compiler mainly amounts to extracting the indirect implementation and checking that it is consistent with the declarative specification. With MADL, the compilation process is structured in two phases:

1. *Elaboration.* Memory locations are indirectly specified in MADL. As for types, an elaboration pass turns a latent structure of the MADL program into an explicit specification of allocation. The implicitly allocated language must give enough control to specify the allocation.
2. *Projection.* A fully allocated program contains both the description of an algorithm and its memory implementation. Both aspects can be extracted in a straightforward manner, to obtain a purely declarative description or an imperative implementation. In particular, compilation — projection onto the imperative world — does not need any transformations.

As a result, the MADL compiler is mainly made of static analyses and inference passes. In particular, we have not designed any optimizations, not only because of lack of time, but also because optimizations can be conducted on the source. The MADL language should allow to describe all the sequential implementations of a given algorithm.

The Holy Grail The MADL language aims at targeting both sequential and parallel execution. Although it could be used as a stand-alone language, it has been designed with a view to allow a relatively simple elaboration from a subset of Scade. Its compilation is depicted below:



The language is primarily defined by a *two-sided* semantics that accounts for its declarative and imperative nature. It defines how programs are evaluated in an operational way by writing to a global memory, while ensuring that the evaluation is consistent with data-flow dependencies. The two-sided semantics exists under conditions on programs that are checked statically. However, this semantics and the related verifications apply to fully located programs. We also pursue a memory-less definition of MADL with a *functional* semantics for a couple of reasons.

- The functional semantics describes MADL at a more abstract level, closer to the usual semantics of synchronous languages. It gives a base to state an equivalence between the two-sided view and the purely functional view.
- It allows to define correctness criteria that apply to unallocated programs. In particular, checks that do not use memory locations report more helpful errors than their memory-aware counterparts, in particular, if Scade were the input language.

Some aspects of the semantics and the associated static checks are independent of memory concerns, for instance, typing. By contrast, others depend on the memory discipline, in particular initialization. For the memory dependent aspects, verifications are more easily conducted on fully located programs, i.e., after allocation inference. However, we provide coarser checks that do not rely on memory information. They help for the definition of the functional semantics. These checks conciliate unachievable soundness and completeness objectives. They should reject as many programs as possible without restraining too much the expressiveness power of the language.

The shaky reality Only some of the various dimensions sketched above have been studied. On the compilation side, neither the μSCADE elaboration nor the parallel back-end have been studied. We focus on a compilation chain from the dedicated front-end to sequential code. The scheduling aspects have been rapidly studied and prototyped, but are not part of the compilation process for now. On the formalization side, we described the type system and the allocation discipline. We also outlined the main principles of the two-sided semantics, but we do not support the full language. The functional semantics is even less developed, although we sketch some of its principles.

Moreover, our language is rather restricted and some important features are lacking. There are no clock operators, conditionals or functional updates for arrays. Some are partially supported, e.g., modular checks and code generation. However, we think that the language contains the main ingredients to define a declarative language with a precise control of memory. To do this, we explored several topics that we review here briefly.

A sized typed system We propose a Hindley-Milner-like type system where sizes are multivariate polynomials. Its objectives are twofold: to allow checking and inferring sizes in a modular way and to provide a compile-time description of sizes that is at the heart of our description of memory allocation.

Our proposal takes advantage of the context of SCADE. All sizes are ultimately known at compile-time, which allows complete but non modular verification of array accesses. Hence our type system is complete. It ensures size consistency, but leaves some size inequality constraints to a non modular verification, once the final values of sizes are known. The size language is expressive enough to describe a large class of array operations, but this comes at the cost of inference completeness. To guide inference in the failing cases, we propose a mechanism based on singleton types.

Our approach allows to track size relations. It is applicable in a larger context than our first order language, so we present it as an extension of an ML type system, that strongly resembles the indexed types of Zenger [Zen97]. However, we give sizes a semantic role here, e.g., they may determine the size of arrays and iteration bounds.

A memory location discipline The imperative side of MADL relies on a size and type polymorphic memory model that aims at describing complex accesses to structured data (tuples, arrays). This description is based on two restrictions: (i) the memory contains pure data *chunks*, i.e., without any indirections, and (ii) the chunks are not aliased. Complex accesses to chunks are described by *projection functors* that are static mapping of projections. For example, viewing a matrix chunk in a transposed way is described with a functor that swaps accesses. As for sizes, the projection functor language is again composed of multivariate polynomials. This is the result of a compromise between its expressiveness and formal manipulation possibilities, notably for composition and comparison. In essence, our projection functors surprisingly resemble the *memory moves* proposed by Iannetta, Gonnord, and Radanne [IGR21] to implement in-place pattern matching, although the described objects are different, since the authors aim at reusing the constructors of algebraic data types.

Structural operations such as matrix transposition are represented as location relations between their arguments and results instead of computations which produce new values. As a result, these operations are translated into complex accesses without the help of any optimizations. The description of array operations as location relations is largely inspired by the pull and push array distinction proposed by Claessen, Sheeran, and Svensson [CSS12] for OBSIDIAN. In MADL, the separation between demand-driven and data-driven array operation guides allocation inference.

A declarative iteration construct Instead of the usual `map` and `fold` iterators of functional languages, that are second-order operators, we propose a first-order **`forward`** construct to iterate over arrays. Hence, it does not need to distinguish some operator expressions as currently done in SCADE. It is inspired by the `initial-for` loop of SISAL [FCO90]. This construct benefits from the sequential nature of MADL in which arrays are handled as (finite) sequences. Contrary to functional iterators, it easily describes multidimensional iterations. Section 1.5.1 illustrates this construct.

We propose a novel feature for this kind of iteration by allowing recursive definitions of arrays. The computation of an element may use already computed ones. In SCADE, expressing such computation schemes is contrived, e.g., mimicking imperative algorithms, and leads to unsatisfactory

generated code. This extension preserves the declarative style of MADL. It also avoids useless initializations and array update operations that may introduce copies.

GPUs programming Among our initial objectives, we planned to explore how SCADE could be compiled for GPUs. It rapidly appeared that a more precise control of memory would be necessary so as to obtain reasonable parallel code. Although we did not address the question of a GPU back-end, this objective has driven the design of MADL. In particular, the algorithm - schedule separation proposed by Ragan-Kelley et al. [Rag+13] inspired our proposal of a clear separation between memory allocation and data-flow variables.

Organization Chapter 2 present our sized type system using an ML-like language that is independent of streams and synchronous features. Chapter 3 introduces *projection functors* and their manipulations, independently of their use for describing memory locations. We introduce MADL without iteration in Chapter 4 alongside its two-sided semantics and some hints about a functional one. Chapter 5 is devoted to the `forward` construct. The location discipline and allocation inference are presented in Chapter 6. Last, we sketch scheduling and code generation in Chapter 7.

Vocabulary and conventions We call functions on streams *operators*. The term *node* is dedicated to a special kind of operators, the stateful ones. From a transition function point of view, operators are *instantiated*: each occurrence must be given its own state, i.e., its own history, if any. The term *operations* is used to denote atomic computations, e.g., scalar ones, that are blindly forwarded to the target language (C). Moreover, the *sequential* features, as opposed to the *combinational* ones, refer to operations that depend on the stream nature of values, e.g., delays.

MADL builds on a fundamental distinction between *structural* and *computational* expressions. The first category refers to the parts of programs that are eliminated statically, i.e., that are transformed into complex accesses, while the latter denotes the parts of programs that compute something.

In the following, we use SCADE as a representative for the languages reminiscent of LUSTRE such as LUCID SYNCHRONE [CP96], HEPTAGON or SCADE itself. Hence, some of the features of 'SCADE' we point out also exist or have existed in some of these other languages.

In the presentation of MADL, we use the *imperative* qualifier to designate the memory aspects of imperative paradigms, without including their *sequential* side, to be understood here as the order of execution. Hence we claim MADL to be both declarative and imperative although it imposes only a partial order on computations.

1.5 Overview

Before entering the detailed presentation of our work, we propose a rapid tour of our *Memory-Aware Declarative Language*. This preliminary section builds on various examples to illustrate the main principles of the language. It gives an overview of the contributions. It should guide the reader through the formalizations. This section also presents the current status of our prototype and its limitations.

1.5.1 Streams and Iterations

As a pretext to visit the different features of MADL, let us implement one of the simplest linear algebra operations: the scalar product — $u \cdot v$ — between vectors u and v of length n. It is defined as the sum of the point-to-point product:

$$u \cdot v = \sum_{i=1}^{n} u_i v_i$$

In MADL, vectors are represented with arrays. The above formula is implemented by the following `dot` operator. This program is rendered with the *abstract syntax* of MADL. Unless explicitly specified, all the code snippets we give in abstract syntax are correct.

```
let dot (u, v) returns (s):
  forward ([ui] = u) ([vi] = v) returns (s = {si}):
    si init 0 = last si + ui * vi;
```

This program declares an *operator* — `dot` — that expects two streams of arrays u, v and produces one stream of scalars s. Its body is made of a single *expression* — `forward` ... — that interprets the elements of arrays u and v as *faster* flows ui and vi. We elaborate on this interface below. The body — si `init` 0 `=` `last` $si + ui * vi$ — defines a flow si that sums the products $ui * vi$. Its value at the end of the iteration defines s.

Depending on the purpose of examples, some are reproduced with the *concrete syntax*. In these cases, the various parts are automatically generated using the output of our compiler prototype. This allows to show some additional information: the inferred type scheme, errors, or generated code. The concrete syntax for the scalar product example is similar:

```
1 // Scalar product
2 let dot (u, v) returns (s):
3   forward ([ui] = u) ([vi] = v) returns (s = {si}):
4     si init 0 = last si + ui * vi
```

```
val dot: size i. [i]int, [i]int → int
```

Type system The language is implicitly but strongly typed: although no type annotations are given in the definition of `dot`, the compiler infers the type scheme `size` i. `[`i`]int, [`i`]int → int`. The types of MADL contain sizes. Hence the signature of `dot` reflects the expected condition on array sizes, i.e., that the `u` and `v` arguments are two integer arrays — `[`i`]int` — of the same size. The type is polymorphic with respect to the size variable i.

To make sizes and types explicit, we declare a size parameter — `<<n>>` — in the interface of the operator. Morever, the expressions are decorated with type annotations, that are introduced with the · `of` t syntax. Last, iteration length is also specified:

```
1 // Scalar product, explicitly typed
2 let dot <<n>> (u of [n]int, v of [n]int) returns (s of int):
3   forward <<n>> ([ui] = u) ([vi] = v) returns (s = {si}):
4     si init 0 = last si + ui * vi
```

```
val dot: size i. «i» → [i]int, [i]int → int
```

The inferred signature is slightly different. It contains an additional *size parameter* — «i» — that represents a size that can be specified at instantiation. Sizes are checked alongside types and the compiler rejects programs whose sizes are inconsistent:

```
1 let main () returns (s):
2   s = dot ([| 1,1,1 |], [| 1,1 |])
```

```
Error: This expression has type [3]int
but an expression of type [2]int was expected
```

```
Note: Size 3 is not compatible with size 2
```

The syntax `[|` ... `|]` denotes immediate arrays. In this example, the size parameter is omitted, but it could be specified as well: `dot «3» ([| 1, 1, 1 |], [| 1, 1, 1 |])`.

Streams and iterations MADL is a stream language. The `last` construct delays a stream, i.e., it allows to access its previous value. It corresponds to the `pre` temporal operator of SCADE. Because accesses to the past are undefined at the first instant, some mechanisms are needed to ensure outputs do not depend on them. SCADE provides the `->` binary operator that builds an initialized flow by defining its value at the first instant. It is supplemented with a type system that ensures a consistent use of `pre` and `->`. In MADL, the `init` construct sets the initial value of flows used with `last`, hence the accesses to the past are always initialized. This form is less expressive, but it enjoys better compilation properties. In the operator `dot`, the initial value of the flow si — si `init` 0 — specifies the value of `last` si at the first instant.

The `forward` construct iterates over arrays by viewing them as streams. In dot, the argument clauses — `[`ui`] =` u and `[`vi`] =` v — define *current elements* ui and vi as flows whose values are made of the elements of arrays u and v. These flows are *faster*: each value of the flows of arrays define n values of the flows of current elements. The result clause — s `= {`si`}` — specifies that the flow s — the result of the iteration — is the last value of si. This allows to accumulate. As for accesses to the past (`last`), the returned flow has to be initialized, so that it has a value even if the number of iterations is null.

So that our definition actually computes the scalar product between the arrays of flows u and v, the si flow must be reset at each synchronous cycle. This is the default behavior of the `forward` construct, but it may be explicitly specified with the `restart` *reset condition*. It is sometimes necessary to keep the state of the iterated stream definitions instead of resetting it at each synchronous cycles. This allows for instance to compute a sum of array elements across multiple synchronous cycles. It is specified with the `resume` reset condition. To illustrate these reset conditions, we depict the semantics of two operators `sum` and `integr` that implement a restarted and resumed sum respectively. In the chronogram, the flows of ui and vi are faster. The black stars represent reset, i.e., the writing of 0 for the si flow. Reset happens at each cycle for the restarted iteration and only once at the beginning for the resumed one.

```
let sum (u) returns (s):
  forward ([ui] = u) returns (s = {si}):
    si init 0 = last si + ui;
  restart
```

```
let integr (u) returns (s):
  forward ([ui] = u) returns (s = {si}):
    si init 0 = last si + ui;
  resume
```

	Cycle 1			*Cycle 2*			*Cycle 3*			*Cycle 4*			···
u	[| 1, 2, 3 |]			[| 1, 2, 3 |]			[| 1, 2, 3 |]			[| 1, 2, 3 |]			···
`sum (u)`		6			6			6			6		
ui	1	2	3	1	2	3	1	2	3	1	2	3	···
si	⋆ 1	3	6	⋆ 1	3	6	⋆ 1	3	6	⋆ 1	3	6	⋆
`integr (u)`		6			12			18			24		
ui	1	2	3	1	2	3	1	2	3	1	2	3	···
si	⋆ 1	3	6	7	9	12	13	15	18	19	21	24	

Code generation For both versions of the scalar product (with or without the size parameter), our compiler generates the following C code. Indeed, the size parameters are only considered for typing purposes.

```
1 // Scalar product
2 let dot (u, v) returns (s):
3   forward ([ui] = u) ([vi] = v) returns (s = {si}):
4     si init 0 = last si + ui * vi
```

```
val dot: size i. [i]int, [i]int → int
```

```
1 void dot (size_t I, int *s, int *u, int *v) {
2   *s = 0;
3   for (int idx = 0; idx < I; idx++)
4     *s += u[idx] * v[idx];
5 }
```

The compiler generates *size polymorphic* code (type polymorphic code is not yet supported). All the returned values, even of scalar types, are passed by pointer. This point is among the possible but non-critical future improvements.

Multidimensional iteration The `forward` construct allows for multi-dimensional iteration. In its clauses, the dimensions are separated by commas. It is the equivalent of the `cross` operator of Sisal [FCO90] We illustrate this feature in Figure 1.14 with a matrix multiplication in an (almost) naive way. Line 3 initializes the result matrix with 0 by iterating on two dimensions. The second `forward` iterates on three dimensions to sum the element products in the appropriate element of the result. The `[i]`, `[j]` and `[k]` constructs capture the current index of iteration. They are unused here, but they give hints for the compiler to name variables in the generated code.

Let us review the various iteration clauses:

- Line 3 — *c0* `=` `[[`*c0ij*`]]` — This clause builds a matrix *c0*, i.e., a bidimensional array, by aggregating the values of *c0ij*. The rank of the result is defined by the number of nested aggregation constructs — `[` · `]` —, two here.
- Lines 5,6 — `[[`*aij*`]]` `=` *a*, `[[`*bkj*`]]` `=` *b* — These clauses traverse matrices by naming current elements using the preceding dimensions. Here as well, the number of dimensions is defined by the nesting depth of accumulations. In particular, the traversal of *b* only use the two

```
1 // Matrix product, vectorizable
2 let mat_mat (a, b) returns (c):
3   forward [i], [j] returns (c0 = [[c0ij]]): c0ij = 0 end
4   forward <<5>> [i],
5           <<7>> [k] ([[aik]] = a),
6           <<3>> [j] ([[bkj]] = b)
7   returns (c = [{[cij]}] init c0):
8     cij = last cij +  aik * bkj;
```

```
val mat_mat: [5][7]int, [7][3]int → [5][3]int
```

```
1 void mat_mat (int a[5][7], int b[7][3], int c[5][3]) {
2   for (int i = 0; i < 5; i++)
3     for (int j = 0; j < 3; j++)
4       c[i][j] = 0;
5   for (int i = 0; i < 5; i++)
6     for (int k = 0; k < 7; k++)
7       for (int j = 0; j < 3; j++)
8         c[i][j] += a[i][k] * b[k][j];
9 }
```

Figure 1.14: *Matrix multiplication in MADL and the generated code*

inner dimensions. The names of current elements are consistent with the index variables of the dimensions they traverse.

- Line 7 — `c = [{[`*cij*`]}] init` *c0* — This clause defines the resulting matrix *c* from the inner variable *cij* by aggregating along the first and third dimensions and accumulating along the second one, starting from the *c0* matrix. The semantics of `last` *cij* is subtle. Intuitively, it represents the previous value of the memory location of *cij*, i.e., the matrix element at index `[i,j]`. As the generated code shows, `last` *cij* is not the value of *cij* at previous iteration of the inner-most loop, i.e., `c[i][j-1]`, but the value of the element of `c[i]` at the previous iteration of the middle loop. The underlying principles are presented in Chapter 5.

This definition forces immediate values for the iteration sizes and hence fixes the matrix dimensions. This only matters for the readability of the generated code. Indeed, our compiler does not use the variable-length arrays of C. If dimensions are unknown, it generates code where multidimensional arrays are flattened into a single array. For instance, the element `[i,j]` of a matrix `m` of size `(n,p)` is accessed with `m[i*n+j]`. Forcing immediate sizes allows to generate nested array projections: `m[i][j]`.

1.5.2 The Pervasive Spectrum of Memory

MADL aims at giving control on memory allocation. However, memory is implicit in the programs. Instead of defining where values are stored, the language specifies which values can be copied. To do this, the operations that compute, i.e., that write a new value somewhere in memory, are distinguished from the *structural* operations that introduce relations between memory locations. In particular, no copies are implicit, even for scalars. MADL is a declarative language, programs are in single assignment form and they make explicit all data dependencies. However, the underlying memory concerns explain some of the unusual features of the language.

First order array combinators To illustrate how this *no implicit copy* principle constrains allocation in the generated code, we first introduce two built-in array operators, `concat` and `window`. The former is also written as an infix operator `++`. The latter expects a vector u (a unidimensional array) and builds a matrix m (a bidimensional array) whose rows contain the slices of u. We show their uses and type schemes by η-expanding the predefined operators:

```
1 let concat (a,b) returns (c): c = a ++ b
2 and window <<k>> (a) returns (m): m = window <<k>> (a)
```

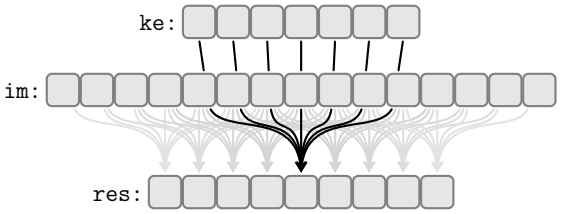
```
val concat: size i j. type a. [-i+j]a, [i]a → [j]a
val window: size i j. type a. «i» → [-1+j+i]a → [j][i]a
```

The type scheme of `window` universally quantifies over two variables, representing the dimensions of the resulting matrix. The size parameter «*i*» allows to specify the length of the slices. We illustrate its definition on a concrete example:

```
window «3» ([| 1, 2, 3, 4, 5 |]) = [| [| 1, 2, 3 |] ,
                                       [| 2, 3, 4 |] ,
                                       [| 3, 4, 5 |] |]
```

These two operators are *structural*: they do not build their result in a new memory location. In the case of `window`, the result is made of the *same* data, i.e., it is the same memory location but it is accessed differently. For concatenation, allocation must ensure that the arguments are computed in the left and right parts of a unique array, that thus contains the result.

Convolution A unidimensional discrete convolution amounts to computing a scalar product for each slice of a given array. This operation is depicted below. Because each element is computed using its neighbors, the resulting array is smaller that the input.



The operation is a typical job for the `window` operator. This convolution can be written:

```
// Unidimensional convolution with kernel [ke]
let convol (ke, im) returns (res):
  forward [i] ([im_i] = window <<_>> (im)) returns (res = [res_i]):
    forward [j] ([ke_j] = ke) ([im_ij] = im_i) returns (res_i = {s}):
      s init 0 = last s + ke_j * im_ij
```

```
val convol: size i j. [i]int, [-1+j+i]int → [j]int
```

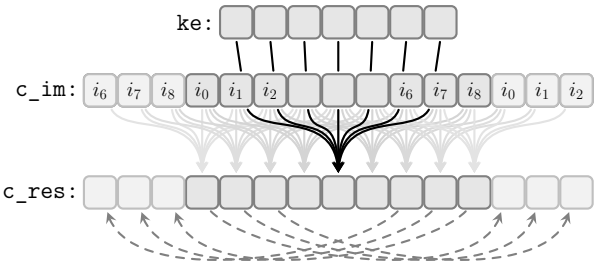
```
void convol (size_t J, size_t I, int *im, int *ke, int *res) {
  for (int i = 0; i < I; i++) {
    res[i] = 0;
    for (int j = 0; j < J; j++)
      res[i] += ke[j] * im[i+j];
  }
}
```

The placeholder — `<<_>>` — stands for a free size variable that the compiler will infer. The inner `forward` iterator is exactly the scalar product defined previously. It is inlined for two reasons: (i) in our prototype, code generation does not support instantiation for now and (ii) inlining nodes, in particular in iterations, relaxes some constraints on the underlying allocation (see Section 7.2.2).

Unsurprisingly, the type scheme of `convol` illustrates a size relation between the input vector and the output one that is similar to the one of `window`. The convolution cannot be defined on the sides of the array, and `res` array in the above illustration is smaller than the `im` array.

Apart from the order of sizes and parameters that are swapped around, the MADL definition and the generated code have comparable structures but an element disappeared in the latter. The matrix of slices, that is, the result of `window «_» (`*im*`)`, is never computed. Instead, the input vector is directly accessed with a computed index: `i+j`. *No optimizations are involved here.* Indeed, the `window` operator is *structural*, it does not compute anything, but rather introduces a mapping between its result and the source array. This mapping is turned into the complex projections used to access the result of `window «_» (`*im*`)` through the *im_ij* variable.

Circular convolution Some algorithms may need to preserve the size of arrays used in convolutions. Array padding and circular indexing (toroidal for multidimensional convolutions) are two classical strategies to compensate the unavoidable size shrinking. We propose to implement the second one in a slightly unusual way. For now, MADL cannot describe circular accesses (i.e., computing indexes with a modulo). Instead, we consider arrays whose sides are replicated so that all the elements may be computed as above. Convolution must duplicate the two ends in order to maintain this invariant. This last step is depicted below by the dashed arrows:



The MADL implementation follows. We force the kernel to have an odd size — $2 * 'k + 1$ — so that each array end shrinks by $'k$ elements. This size $'k$ is an undeclared size variable, like the $'a$ type variables in OCaml. The computation of the scalar product is exactly the same as before, apart from the size parameter of **window** that is now specified (line 5). The **end** keywords limit the extent of the **forward** iterator. Otherwise, it would extend to the end of the declaration.

```
1  // Unidimensional circular convolution with kernel [ke] on redundant array [im]
2  let c_convol (ke, c_im) returns (c_res):
3
4    // Compute the inner elements
5    forward [i] ([im_i] = window <<2*'k+1>> (c_im)) returns (res = [res_i]):
6      forward [j] ([ke_j] = ke) ([im_ij] = im_i) returns (res_i = {s}):
7        s init 0 = last s + ke_j * im_ij
8      end
9    end
10
11   // Extract left and right parts of the result
12   l ++ _ = res;
13   _ ++ r = res;
14
15   // Copy the side parts. Type annotations are needed to specify sizes.
16   dup_l of ['k]_ = !r;
17   dup_r of ['k]_ = !l;
18
19   // Build the redundant result
20   c_res = dup_l ++ res ++ dup_r
```

```
val c_convol: size i j. [1-2*i+2*j]int, [-2*i+3*j]int → [-2*i+3*j]int
```

```
1 void c_convol (size_t J, size_t I, int *c_im, int *c_res, int *ke) {
2   for (int i = 0; i < J; i++) {
3     c_res[-1*I+J+i] = 0;
4     for (int j = 0; j < -2 * I + 2 * J + 1; j++)
5       c_res[-1*I+J+i] += ke[j] * c_im[i+j];
6   }
7   memcpy (&c_res[0], &c_res[J], (-1 * I + J) * sizeof (int));
8   memcpy (&c_res[-1*I+2*J], &c_res[-1*I+J], (-1 * I + J) * sizeof (int));
9 }
```

After computing *res* (line 5), the left and right parts are extracted by using a concatenation on the left-hand side of equations (lines 12 and 13). How does the compiler know where to cut? This sizes are deduced from the uses of the variables l and r. Indeed, their copies (lines 16 and 17) are annotated with the type `['k]_` that specifies the sizes of these arrays. Without annotations, these sizes would stay unconstrained and generalized. Finally, the redundant array is built by concatenating the convolution result and the duplicated parts (line 20).

The interface is more complicated than it should. We did not yet try to simplify type schemes. Indeed, by the change of variable $k = -i + j$, $n = j$ we obtain the much more readable form:

```
val c_convol: size k n. [1+2*k]int, [2*k+n]int → [2*k+n]int
```

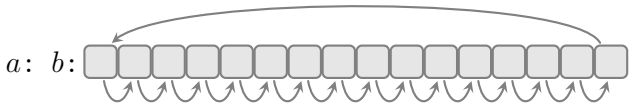
Compared to the MADL definition, the generated code grows modestly. The first nested loops are similar to those of the simple convolution, except that results are written directly in `c_res`, the final result, starting at index `J-I`, i.e., k, instead of 0 as in the previous implementation. The replications of both ends copy some elements of the computed part of `c_res` into the remaining parts of the result.

In short, all the variables of the program refer to some part of arrays *ke*, *im* or *c_res*. This is not a coincidence. In this program, the only expressions that write values in memory are (i) the scalar computation (line 7) and the copies (line 17 and 18). The compiler must find a memory

allocation that fulfills the constraints originating from the structural expressions, otherwise the program is rejected.

Memory constraints and copy sprinkling The structural constructs of MADL, such as `concat` and `window`, do not introduce any copies. This is also the case for equations. If we write b = a, both variables are represented in the generated code by the same memory location. However, it is sometimes necessary to duplicate values, notably if we aim to overwrite them. This is the role of the explicit copy construct — `!`· — in the definition of `c_convol` (line 18).

To illustrate the need of explicit copies, let us define an in-place array rotation, whose semantics are depicted below. Elements are shifted to the right in a circular way: the right-most element is moved to the left end. This operation produces its result at the same memory location as the input array.



We first define an operator that accesses the rightmost element of an array. It has no body and it does not contain any computational operations. Like `concat` or `window`, this user-defined operator is structural.

```
1 let right_elem (_ ++ [|a|]) = (a)
```

```
val right_elem: size i. type a. [1+i]a → a
```

The sign of sizes *Remark*

Its signature might be misleading as it seems to require an array of size at least 1, a constraint we might expect. However, sizes are integers — elements of $\mathbb{Z}$ — and the compiler cannot check in a modular way, during typing, that instantiated sizes are indeed positive. The type system only ensures size consistency, i.e., that sizes match.

The array rotation algorithm is simple. It stores one element in a temporary memory, initialized with the last element and traverses the array by swapping the current element and the temporary memory. Here is a decomposed implementation in MADL.

```
1 let rotate_right (a at 'l) returns (b at 'l):
2
3   a of [_]int;  // Monomorphisation
4
5   forward ([ai] = a) returns (b = [bi]):
6     reg init ! right_elem (a);
7     tmp = ! last reg;          //
8     reg = ! ai;                // Swapping [last reg] and [ai]
9     bi  = ! tmp;               //
```

```
val rotate_right: size i. [i]int → [i]int
```

```
1 void rotate_right (size_t I, int *a) {
2   int reg;
3   reg = a[I-1];
4   for (int idx = 0; idx < I; idx++) {
5     int tmp;
6     tmp = reg;
7     reg = a[idx];
8     a[idx] = tmp;
9   }
10 }
```

The type annotation (line 3) enforces a non-polymorphic type. This allows to generate the code, since type polymorphism is not supported in the back-end.

The · `at` $'l$ annotations (line 1) are location constraints. They use an undeclared location variable $'l$ that is the location counterpart of size and type variables. These annotations require that the argument a and the result b are stored at the same location. Since MADL is declarative, we cannot modify a. Indeed, the generated code expects a single array.

Inside the body of the `forward` iterator, we separate the initialization of the register (line 6) and a step-by-step swap (lines 6-8). Recall that *ai* and *bi* are stored at the same location and likewise for `last` *reg* and *reg*, because reading or defining the state is a copy-less operation.

The four copies are crucial here. The register *reg* must be independent from the array, hence the copy for the initial value. The swapping of two values requires a temporary storage *tmp*. In the generated code, these two locations have different scopes. The register *reg* is part of the state of the iterated body. It is defined in the enclosing scope (line 2) while the temporary value *tmp* is part of the iterated scope (line 5).

Memory scheduling With such sharing constraints between values, the scheduling of operations must fulfill a strong property: no value is overwritten if it is needed by a following computation. Here lies a substantial missing part in our compilation chain: the memory-aware scheduling has been prototyped on an older version of the language but it has not been ported yet. The order of instructions is currently retrieved from the syntactic order. We identified two steps for future work:

1. Build a dependency analysis that supports memory locations. A first prototype of memory-aware scheduler has been designed for a simpler version of the language, but it has not been ported to the version we present here. We sketch its mechanism in Chapter 7.

2. Build an extended causality analysis based on program variables that allows to detect and report scheduling issues in an intelligible form, i.e., without memory locations. Our experimentation with a dependency analysis shows an immediate limitation: scheduling errors — cycles in a graph made of data dependencies and anti-dependencies — are almost incomprehensible to programmers, in particular because anti-dependencies arise from memory locations, that are implicit in the programs.

To emphasize the data-flow nature of program variables, the above array rotation operator may be defined with a one line iteration (we drop monomorphisation here). The four copies are still necessary. Because we specify operations instead of memory, the two-copy pattern — `!!`$\cdot$ — actually introduces an independent memory location, that relaxes scheduling constraints. This location is the one of the *tmp* variable above. This definition also illustrates how the `last` construct may apply on any initialized expressions, even without introducing a variable such as *reg* variable in the first definition.

```
let rotate_right «n» (a at 'l) returns (b at 'l):
  forward «n» ([ai] = a) returns (b = [bi]):
    bi = !!last (!ai init !(right_elem (a)))
```

As a last example, here is the definition of a schedulable operator that produces an alternating sequence of zeros and ones, by swapping the values contained in two unnamed registers. Its value at the first instant is the initial value of the argument of the outermost `last`: 1. The definition of *a* is equivalent to the SCADE expression `a = 1 -> pre (0 -> pre a)`.

```
let oscillate () returns (a):
  a = !last (!last (!a init 0) init 1)
```

A two-level reading grid MADL programs should only be understood from their declarative side, i.e., following the data dependencies. However, bearing in mind the strict memory constraints underlying the various constructs helps to understand them. In particular:

- MADL left-hand side expressions, called *patterns*, may contain complex constructs, in particular operator instantiations. However, a crucial restriction applies: patterns must be *structural*. We can insert a concatenation on the left of an equation (lines 12 and 13 of the circular convolution) because there are no computations involved here. The concatenation defines relations between the memory locations of its arguments and results that must be fulfilled by the allocation. These relations may be used in the opposite direction. Hence, writing the concatenated array defines at the same time its left and right parts.

- Initialization is indirect. In the matrix multiplication example, the accumulation matrix is initialized — `[{[`*cij*`]}] init` *c0* — (line 7) whereas the last values of the current element is

accessed — `last` *cij* — (line 8). How does the initial matrix *c0* define the initial value[11] of *cij*? Here as well, a memory-aware look shall clarify the situation. `init` and `last` constructs apply in essence to memory locations, the state. Hence the initial value of flows, in particular variables, may be defined by initialization of flows they are part of. The practical consequence is similar for patterns: uses of `last` are well-initialized if they are separated from some `init` constructs by *structural* expressions only.

1.5.3 Recursive Array Comprehension

Some algorithms, such as pivot-based ones, define array elements using the previously computed ones. Among them, a satisfactory description of the Cholesky decomposition was rightly identified by Jean-Louis Colaço as a motivating objective. Indeed, this use-case challenges most of the topics we explored: typing, expressiveness and scheduling. These algorithms are easily expressed in imperative languages where allocation is separated from assignments: first allocate an array of the right size, then fill it step-by-step, accessing the elements that are needed.

Such algorithms may be defined with recursive array comprehensions as proposed by Sinkarovs et al. [Sin+17]. In general, lazily evaluated arrays provide a solution, but they require a dedicated runtime and might lead to dead-locks. We follow an alternative path, whose expressiveness is more limited, but that guarantees a simple structure in the generated code. To illustrate this, we propose to compute an array of the powers of two using the formula:

$$2^{n+1} = 1 + \sum_{k=0}^{n} k^n$$

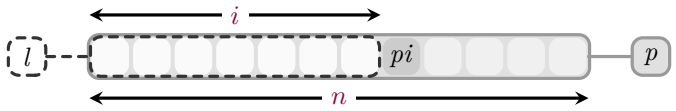
At each iteration of this (rather inefficient) definition, we must compute the sum of the already computed elements. In our language, this is translated into the following operator:

```
1 // Powers of 2, using:  p[i] = 1 + sum (p[0:i])
2 let pow2 () returns (p):
3   forward [i] returns (p = [pi] as l):
4     forward [k] ([pk] = l) returns (pi = {s}):
5       s init 1 = last s + pk
```

```
val pow2: size i. [i]int
```

```
1 void pow2 (size_t I, int *p) {
2   for (int i = 0; i < I; i++) {
3     p[i] = 1;
4     for (int k = 0; k < i; k++)
5       p[i] += p[k];
6   }
7 }
```

As above, the indexes (`i` and `k`) only serve as name hints for code generation. The alias — · `as` *l* — introduced in the outer `forward` iteration is special: the variable *l* represents the array of already constructed values. As depicted below, its size is equal to the index of the iteration. This is confirmed by the generated code where the bound of the inner loop is the index of the outer one.



[11] Or values? A more advanced mechanism is at work here: *distant* `last` (see Section 5.1.4).

2 Polymorphic Types with Polynomial Sizes

Introduction

MADL is a statically typed language. Its type system describes the sizes of arrays. It aims at expressing as many relations between sizes as possible while remaining decidable and largely implicit. This type system combines two widely studied features: (i) a restricted form of *refinement types*, as defined in Xi and Pfenning [XP99], and (ii) the ubiquitous *let-polymorphism* of Milner [Mil78] extended to sizes.

Polynomial sizes lead to constraints that cannot be resolved symbolically. Nonetheless, the context of SCADE ensures that all sizes are ultimately known statically: for the monomorphic main node, size checking is trivial but late and non modular. We pursue an earlier and modular size checking, that relies on instantiation as a fallback mechanism only. The typing concerns are orthogonal to the stream nature of the language. The stream constructs are not considered in this chapter.

The interest of a static description of sizes go beyond correctness concerns. Knowing the sizes of arrays is a prerequisite for a static description of memory allocation. This formal representation of sizes is essential for a modular compilation, that generates code for operators that are generic in types and sizes.

We present our type system with a core ML-like language $\mathcal{L}^\eta$ that contains the minimal constructs required to introduce sizes into types. In particular, it has no primitive notion of arrays: they are considered as functions on a bounded domain. This allows to concentrate on the specificities of our type system independently of the memory and expressiveness concerns of MADL. Moreover, this type system is applicable in a wider context than the one of MADL. In particular, it is not restricted to a first-order language. Hence we formalize our type system in this wider situation.

Most of the material in this chapter is taken from a published article [CPP23], of which a preliminary version is also available (in French) [CPP22]. We added the details of some proofs and we discuss how size inference benefits from abstract size variables in Section 2.4.4. This last point is notably used for recursive array aggregation.

2.1 Overview

This section gives an informal insight into $\mathcal{L}^\eta$, its type system and type inference through simple examples. $\mathcal{L}^\eta$ is equipped with a separate language for sizes, that is sizes and expressions are distinct syntactic classes. The size language is made of multivariate polynomials. Sizes (ranged over by $\eta, \ldots$) and their variables (ranged over by $\iota, \kappa, \delta, \ldots$) are handled in a similar way to types ($\tau, \ldots$) and type variables ($\alpha, \beta, \gamma, \ldots$).

Intensional Arrays and Size Consistency In many programming languages intended for scientific computations such as SISAL [FCO90], explicit index manipulations are replaced by operators acting on arrays called *combinators* [JS97; Hen+17]. This style benefits functional and data-flow programming languages by allowing to write array definitions with single expressions. Array combinators provide predefined access patterns that are always correct, avoiding the need for runtime checks. However, some of these primitives still require array sizes to coincide. To enforce such properties by type checking, sizes need to be expressed in types. The point-wise function

application (`map`), its binary variant (`map2`) and the array reduction (`fold`), three operators that are available in SCADE, are given the following type schemes in the proposed language $\mathcal{L}^{\eta}$:

```
val map  : ∀ι·α·β. (α → β) → <ι> → [ι]α → [ι]β
val fold : ∀ι·α·β. (α → β → α) → <ι> → α → [ι]β → α
val map2 : ∀ι·α·β·γ. (α → β → γ) → <ι> → [ι]α → [ι]β → [ι]γ
```

Given a polynomial size η, the type `<η>` (read *value of size* η) denotes a *refinement* of the integer base type, i.e., $\{x : \texttt{int} \mid x = \eta\}$. Actually, this is a singleton type. Here, the second argument of each function thus allows to constrain the value of the size variable ι. Using the notation of FUTHARK [Hen+17], `[η]τ` is the type of arrays with length η and elements of type τ. Used as an expression, the syntax `<η>` also designates the only value of type `<η>`, thus the partial application — `map` f `<9>` — can only be given an array of length 9. These signatures highlight polymorphism that acts both on type variables (α,β,γ) and size variables (ι). Using them, the scalar product is expressed as:

```
let dot = λu. λv. fold (+) <_> 0 (map2 (*) <_> u v)
                                  [ ∀ι. [ι]int → [ι]int → int ]
```

The second line renders the inferred type scheme. In the definition of `dot`, the size values — `<_>` — are omitted for both iterators: they are inferred. The inferred type scheme forces the sizes of u and v to coincide. The size variable ι cannot be directly constrained as no arguments have type `<ι>`. Thus, ι will be deduced from the size of u and v. Let us assume the definition of a primitive `window` defining a sliding window of size κ with step 1:

```
val window : ∀ι·κ·α. <κ> → [ι+κ-1]α → [ι][κ]α
```

This function builds a matrix whose rows are slices of length κ of the input array, starting at the element given by the row index. For example — `window <3>` $[0,1,2,3,4]$ — produces the matrix $[[0,1,2],[1,2,3],[2,3,4]]$. The size $\iota+\kappa-1$ encodes the relation between input and output array sizes so that the former is fully read. Filtering data with a kernel K of size κ is a common signal processing operation. It is expressed with a discrete convolution: $(K*I)_i = \sum_{j=0}^{\kappa-1} K_j \cdot I_{i+j}$. This uni-dimensional filter may be defined as:

```
let convolution = λk. λi. map (dot k) <_> (window <_> i)
                                  [ ∀ι·κ. [κ]int → [ι+κ-1]int → [ι]int ]
```

Here as well, inference is able to determine the missing sizes (and types), making the kernel size coincide with the slice length. Inference derives the above type scheme for this declaration. Note that, by a change of variables, it is equivalent to $\forall\iota{\cdot}\kappa.$ `[κ]int → [ι]int → [ι-κ+1]int`.

Extensional Arrays and Bounds Propagation Arrays are not primitive constructs. The type `[η]τ` is a shortcut for `[η]` $\to \tau$ where `[η]` (read *index of size* η) is a second refinement of type `int` denoting non-negative integers strictly less than η: $\{x : \texttt{int} \mid 0 \leq x < \eta\}$. Although not realistic for compilation, this simplifies the formalism. Using this refinement, the `map2` iterator is expressible in $\mathcal{L}^{\eta}$:

```
let map2 = λf. λn:<'ι>. λu. λv. λi:['ι]. f (u i) (v i)
                  [ ∀ι·α·β·γ. (α → β → γ) → <ι> → [ι]α → [ι]β → [ι]γ ]
```

It defines an 'array', i.e., a function with a bounded domain, whose content is obtained by applying f to u and v elements. Array accesses are denoted by the applications — $(u\ i)$ `and` $(v\ i)$ — that respect bounds by construction. The second argument — n — of `map2` is unused, but the type annotations `<'ι>` and `['ι]`, where `'ι` is an *undeclared* size variable alike OCAML's ones, force n to be the size of index i. This argument may then be used to constrain the size of arrays.

The index type only allows to propagate known bounds. Notably, no arithmetic operations are defined for indexes. Values of type `[η]` are obtained by calling functions with a bounded codomain, e.g., the modulo, whose type scheme is $\forall\iota.$ `int → <ι> → [ι]`. Although elementary, this refined type allows to separate bound checking from array accesses.

The Ghost Size Issue In the previous examples, all unspecified sizes were deducible from the types. Not all programs can be treated so simply. With the annotation `[_]`, the `cst` function

below defines a constant array with an arbitrary size.

```
let cst = λc. λi:[_]. c                                  [ ∀ι·α. α → [ι]α ]
let even = fold (+) <_> 0 (cst 2)                        (Error: Unconstrained size)
```

In the declaration of `even`, summing the elements of `cst` 2 without specifying `fold`'s size leads to an ambiguous value since the size is unconstrained. Such definition must be rejected.

Contrary to types in ML-like languages whose semantics is type-erasable,[1] sizes have a *computational content*. They may determine the semantics of expressions. Inference must thus ensure that the semantics of the reconstructed term was already specified in the source. We formalize this property in Section 2.3.5. When type inference succeeds, all well typed annotated versions of the source program evaluate to the same result. Our `even` example becomes unambiguous by adding an argument:

```
let even = λn. fold (+) n 0 (cst 2)                      [ ∀ι. <ι> → int ]
```

2.2 Size Polymorphism in a Functional Language

To focus on the specificities of a sized type system, the tightest possible language $\mathcal{L}^\eta$ is used: a core ML (λ-calculus with let bindings) augmented with a few constructs. In our opinion, there are three main ingredients:

- *A language of sizes.* Although sizes are integer expressions, they deserve a dedicated language, that is tailored for the needs of the type system. For instance, the comparison of sizes would profit being richer that syntactic identity, e.g., $\kappa * (\nu + \mu) = \kappa * \mu + \kappa * \nu$.

- *Some size-dependent types.* Sizes are intended to make types more precise. Some of the type constructs should be parametrized by sizes. For instance, unidimensional arrays should be described by a size and a type.

- *Some size-dependent values.* The semantics of expressions may depend on the value of some sizes. For instance, we would like to use constant arrays without requiring a value to represent their size.

The last point is open to debate. Allowing for values to be constructed from sizes necessarily leads to a *non-size-erasable* semantics. This consequence must be considered according to the objectives of the type system. In our opinion, it serves not only as a verification mechanism, but also as some structural information the user might rely on to program. Similar non type erasable languages have been considered for a long time, for instance, by introducing type classes in Haskell. [Hal+96]

2.2.1 Syntax and Semantics

The syntax of $\mathcal{L}^\eta$ is defined in Figure 2.1. It is explicitly typed, i.e., type annotations are part of expressions. From now on, $n \in \mathbb{Z}$ denotes an integer. To emphasize their similarities, sizes, types and their variables are designated by Greek letters whereas Latin ones will be dedicated to terms and program variables.

Name-spaces and Free Variables Because sizes, types and expressions are syntactically distinct, their variables are taken in distinct name-spaces, respectively denoted $\mathcal{V}_\eta$, $\mathcal{V}_\tau$ and $\mathcal{V}_e$, allowing for name reuse without masking. In the formalization, these sets are considered disjoint. Given a syntactic object o, the set of *free variables* — $\mathcal{FV}(\ o\ )$ — is a subset of $\mathcal{V}_\eta \cup \mathcal{V}_\tau \cup \mathcal{V}_e$ that contains the variables that are not bound by the rules of Figure 2.2: (i) abstraction — $\lambda x\!:\!\tau.\ e$ — binds x in e; (ii) local binding — $\texttt{let}\ x_{\boldsymbol{V}}\!:\!\tau = e\ \texttt{in}\ e'$ — binds x in e' and variables $\boldsymbol{V}$ in τ and e. *Closed* objects are the ones with no free variables.

[1] Unless advanced features, such as type classes, are considered.

η	::=		***Sizes***	$\boldsymbol{V}$	::=	$\varepsilon \mid \iota \cdot \boldsymbol{V} \mid \alpha \cdot \boldsymbol{V}$	***Generalization***
	$\mid$	$\iota,\ \kappa,\ \delta$	variable	$\boldsymbol{S}$	::=	$\varepsilon \mid \eta \cdot \boldsymbol{S} \mid \tau \cdot \boldsymbol{S}$	***Instantiation***
	$\mid$	ν	constant				
	$\mid$	$\eta + \eta$	sum	e	::=		***Expressions***
	$\mid$	$\eta * \eta$	product		$\mid$	${}^{\boldsymbol{S}}x$	variable
					$\mid$	$e\ e$	application
τ	::=		***Types***		$\mid$	$\lambda x{:}\tau.\ e$	abstraction
	$\mid$	$\alpha,\ \beta,\ \gamma$	variable		$\mid$	$\texttt{let}\ d\ \texttt{in}\ e$	local binding
	$\mid$	$\tau \to \tau$	function		$\mid$	$e \triangleright \tau$	coercion
	$\mid$	$\texttt{int}$	integer		$\mid$	$\texttt{<}\eta\texttt{>}$	size value
	$\mid$	$\texttt{<}\eta\texttt{>}$	singleton		$\mid$	n	integer
	$\mid$	$\texttt{[}\eta\texttt{]}$	interval	d	::=	$x_{\boldsymbol{V}} : \tau = e$	***Declarations***

Figure 2.1: *Syntax of* $\mathcal{L}^\eta$

$$\mathcal{FV}(\ \lambda x{:}\tau.\ e\ ) = \mathcal{FV}(\ \tau\ ) \cup \overline{\mathcal{FV}(\ e\ )}^{\,x}$$
$$\mathcal{FV}(\ \texttt{let}\ x_{\boldsymbol{V}} : \tau = e_1\ \texttt{in}\ e_2\ ) = \overline{\mathcal{FV}(\ \tau\ ) \cup \mathcal{FV}(\ e_1\ )}^{\,\boldsymbol{V}} \cup \overline{\mathcal{FV}(\ e_2\ )}^{\,x}$$

Figure 2.2: *Binding rules of* $\mathcal{L}^\eta$. $\overline{\mathcal{FV}(\ o\ )}^{\,x}$ *denotes the set* $\mathcal{FV}(\ o\ ) \setminus \{x\}$

Sizes and Types The size language is made of multivariate polynomials with integer coefficients, $\mathbb{Z}[\mathcal{V}_\eta]$. The main benefit of this restricted class of arithmetic expressions lies in the existence of a normal form: a weighted sum of products of variables. This allows for symbolic comparison of sizes that are structurally different (e.g., $(\iota - 1)^2 - 1 \ = \ \iota * (\iota - 2)$).

Besides functions, the type language contains a single constructor $\texttt{int}$, along with two refinements, as defined by Xi and Pfenning [XP99]: (i) the type $\texttt{<}\eta\texttt{>}$ (read *value of size* η) denotes the singleton $\{\eta\}$ and (ii) $\texttt{[}\eta\texttt{]}$ (read *index of size* η) represents the interval $[\![0, \overline{\eta} - 1]\!]$, where $\overline{\eta}$ is the value of η, depending on the valuation of size variables. In the syntax of refinement types, they are respectively expressed as $\{x : \texttt{int} \mid x = \eta\}$ and $\{x : \texttt{int} \mid 0 \leq x < \eta\}$.

Polymorphism Types, including polymorphism, are explicit in $\mathcal{L}^\eta$. Variables — ${}^{\boldsymbol{S}}x$ — are instantiated with a list $\boldsymbol{S}$ of sizes and types while local bindings — $\texttt{let}\ x_{\boldsymbol{V}} : \tau = e\ \texttt{in}\ e'$ — declare the list[2] $\boldsymbol{V}$ of size and type variables that are generalized. We shift away from the standard notation — $\texttt{let}\ x : \forall \boldsymbol{V}.\ \tau = e\ \texttt{in}\ e$ — to emphasize generalized variables' scopes, which are bound in both type τ and expression e.

Expressions Integers occur in two ways: — n — denotes immediate values while — $\texttt{<}\eta\texttt{>}$ — stands for the only value of type $\texttt{<}\eta\texttt{>}$, that is defined by size variable valuation. This is the only size-dependent value.

Lastly, the coercion — $e \triangleright \tau$ — represents an explicit type constraint. Because of refined types, it plays a central role in the definition of the semantics (see Section 2.2.2).

Arrays $\mathcal{L}^\eta$ has no support for array manipulation, neither in types nor in expressions. For typing purposes, arrays are essentially functions on a bounded domain, expressed by index refinement. To make examples more intuitive, we will use the notation of Futhark [Hen+17] — $\texttt{[}\eta\texttt{]}\tau$ — as a shorthand for the type $\texttt{[}\eta\texttt{]} \to \tau$.

Handling arrays as functions may look irrelevant, in particular when sub-typing is considered. However, the typing issues that arise do not emerge from this simplification. They exist as soon an index type exists, which is useful for primitives such as `mapi`.

[2] The use of lists simplifies the association of generalized variables with their instantiations.

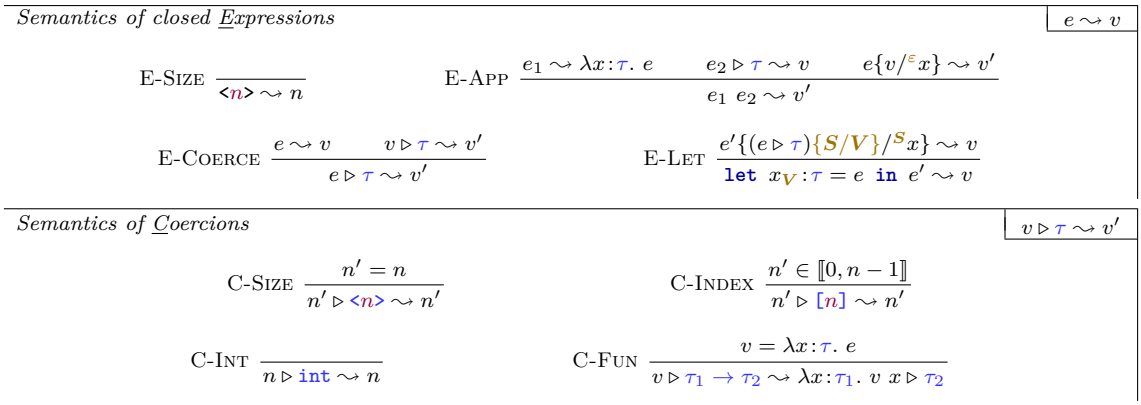


Figure 2.3: *Semantics of $\mathcal{L}^{\eta}$.*

2.2.2 Semantics

As discussed above, the semantics of $\mathcal{L}^{\eta}$ depends on types. We propose a big-step semantics — $e \rightsquigarrow v$ — that associates some closed expressions *values*. As defined at right, values are either integers or abstractions. The deduction rules are detailed in Figure 2.3. They are syntax-directed, thus the $\mathcal{L}^{\eta}$ semantics is deterministic.

			Values
v	::=		
	|	n	integer
	|	$\lambda x\!:\!\tau.\ e$	abstraction

Figure 2.4: *$\mathcal{L}^{\eta}$ values*

Substitutions Substitutions (ranged over by ρ, ...) are defined for each syntactic class and variable/element pairs. Their application is written $e\{\rho\}$. Explicit ones are uniformly denoted $\cdot/\cdot$. Thus, $e\{\eta/\iota\}$ is the substitution in expression e of the occurrences of the size variable ι by the size η, including in sizes and types contained in e. This notation is naturally extended to generalization and instantiation lists, assuming they are compatible, i.e., that each size variable is substituted with a size and likewise for types.

When evaluating let bindings (rule E-Let), each occurrence of the defined variable — ^{S}x — is substituted with its coerced definition in which generalized size and type variables have been instantiated — $(e \triangleright \tau)\{S/V\}$.

Refinements and Coercions The semantics of $\mathcal{L}^{\eta}$ is not type-erasable. This obviously transpires in the rule E-Size that extracts a value from a size. Moreover, refinements discriminate between values of the same *shape* (or base type) and they must be checked in several places. For instance, the term — $(\lambda x : [4].\ e)\ 8$ — should not be reduced further since the argument — 8 — is not a value of the expected type $[4]$. More generally, for any substitution of a term variable, the substituting value must fulfill the substituted variable refinement. Therefore, the E-App and E-Let rules insert coercions; hence the need for an explicit coercion construction in expressions.

For integer refinements, coercions check that the refinement is fulfilled (rules C-Int, C-Size and C-Index). Similarly to λ^{H} [Fla06], function coercions reduce into delayed coercions on argument and result (rule C-Fun), that will be evaluated upon application. The coercion on argument is actually inserted by the evaluation of the introduced application: $v\ x$.

Type Independence Although $\mathcal{L}^{\eta}$ semantics is not type-erasable, only sizes have computational content, i.e., the observational semantics of an expression does not depend on the valuation of its type variables. This observation is crucial for the properties of the inference. Instantiating the types variables of a polymorphic declaration with types containing different refinements leads to terms with identical observational semantics on the intersection of their domains (if the declaration is a function).

Definition 2.1 (Observational equivalence)**.** Two closed expressions e_1 and e_2, are *observationally equivalent* — $e_1 \equiv e_2$ — if and only if, for any closed expressions $a_1, \ldots, a_k$ and integers n_1, n_2:

$$\left\{ \begin{array}{l} e_1\ a_1\ \ldots\ a_k \rightsquigarrow n_1 \\ e_2\ a_1\ \ldots\ a_k \rightsquigarrow n_2 \end{array} \right. \implies n_1 = n_2$$

Observational equivalence states that two expressions that may be applied to the same arguments to produce integer values will coincide. Expressions for which no such common evaluation environment exists are considered equivalent. Used along with typing assumptions to rule out silly cases (e.g., expressions have different types), it allows to compare functions whose domain are disjoint due to refinements, e.g., types $\texttt{<2>} \to \texttt{int}$ and $\texttt{<3>} \to \texttt{int}$.

Definition 2.2 (Equality modulo types)**.** Two expressions e_1 and e_2 are *equal modulo types* — $e_1 \approx_\tau e_2$ — if and only if it exists an expression e, free type variables $\vec{\alpha}$ of e and types $\vec{\tau_1}$, $\vec{\tau_2}$ such that the following syntactic equalities hold:

$$e_1 = e\,\{\vec{\tau_1}/\vec{\alpha}\} \quad \wedge \quad e_2 = e\,\{\vec{\tau_2}/\vec{\alpha}\}$$

Expressions are equal modulo types if they agree on all the sizes used as values, with the expressions $\texttt{<}\eta\texttt{>}$. However, the sizes that appear in type annotations, i.e., in abstractions and instantiations of declarations, may differ: $\lambda x\!:\![4]\tau.\ e \approx_\tau \lambda x\!:\![2]\tau.\ e$ since both terms are obtained by substituting α with $[4]\tau$ and $[2]\tau$ respectively in $\lambda x\!:\!\alpha.\ e$. Type equivalence is preserved by substitution.

Lemma 2.3. *Given expressions e_1, e_2, term variable x and expressions e'_1, e'_2, then*

$$\begin{cases} e_1 \approx_\tau e_2 \\ e'_1 \approx_\tau e'_2 \end{cases} \implies e_1\{e'_1/x\} \approx_\tau e_2\{e'_2/x\}$$

Proof. Some care is required to avoid unwanted captures. Let e_1, e_2, e'_1, e'_2 expressions such that $e_1 \approx_\tau e_2$ and $e'_1 \approx_\tau e'_2$. Then there exist expressions e and e', type variables $\vec{\alpha}$ and $\vec{\alpha'}$, types $\vec{\tau_1}$, $\vec{\tau_2}$, $\vec{\tau'_1}$ and $\vec{\tau'_2}$ such that:

$$\begin{cases} e_1 = e\,\{\vec{\tau_1}/\vec{\alpha}\} \\ e_2 = e\,\{\vec{\tau_2}/\vec{\alpha}\} \end{cases} \quad \wedge \quad \begin{cases} e'_1 = e'\,\{\vec{\tau'_1}/\vec{\alpha'}\} \\ e'_2 = e'\,\{\vec{\tau'_2}/\vec{\alpha'}\} \end{cases}$$

Let $\vec{\beta}$ and $\vec{\beta'}$ be fresh type variables, i.e., that do not appear in any of the types or expressions. These intermediate variables allow to make substitutions commute:

$$\begin{aligned} e_1\,\{e'_1/x\} &= e\,\{\vec{\tau_1}/\vec{\alpha}\}\,\{e'\,\{\vec{\tau'_1}/\vec{\alpha'}\}/x\} \\ &= e\,\{\vec{\beta}/\vec{\alpha}\}\,\{\vec{\tau_1}/\vec{\beta}\}\,\{e'\,\{\vec{\beta'}/\vec{\alpha'}\}\,\{\vec{\tau'_1}/\vec{\beta'}\}/x\} \\ &= e\,\{\vec{\beta}/\vec{\alpha}\}\,\{\vec{\tau_1}/\vec{\beta}\}\,\{e'\,\{\vec{\beta'}/\vec{\alpha'}\}/x\}\,\{\vec{\tau'_1}/\vec{\beta'}\} \\ &= e\,\{\vec{\beta}/\vec{\alpha}\}\,\{e'\,\{\vec{\beta'}/\vec{\alpha'}\}/x\}\,\{\vec{\tau_1}/\vec{\beta}\}\,\{\vec{\tau'_1}/\vec{\beta'}\} \\ &= e\,\{\vec{\beta}/\vec{\alpha}\}\,\{e'\,\{\vec{\beta'}/\vec{\alpha'}\}/x\}\,\{\vec{\tau_1}, \vec{\tau'_1}/\vec{\beta}, \vec{\beta'}\} \end{aligned}$$

Similarly, we obtain:

$$e_2\,\{e'_2/x\} = e\,\{\vec{\beta}/\vec{\alpha}\}\,\{e'\,\{\vec{\beta'}/\vec{\alpha'}\}/x\}\,\{\vec{\tau_2}, \vec{\tau'_2}/\vec{\beta}, \vec{\beta'}\}$$

Hence we have $e_1\{e'_1/x\} \approx_\tau e_2\{e'_2/x\}$. □

This lemma allows to show the preservation of equivalence modulo type by the semantics:

Theorem 2.4 (Type independence)**.** *Given two closed terms e_1, e_2 such that $e_1 \approx_\tau e_2$, then*

$$\forall\ v_1\ v_2,\ \begin{cases} e_1 \rightsquigarrow v_1 \\ e_2 \rightsquigarrow v_2 \end{cases} \implies v_1 \approx_\tau v_2$$

Proof. Consider e_1, e_2 and v_1, v_2 such that $e_1 \approx_\tau e_2$, $e_1 \rightsquigarrow v_1$ and $e_2 \rightsquigarrow v_2$, and examine the rules of expression semantics to check that the above invariant holds.

E-Size Since only types may differ, $e_1 \approx_\tau e_2 \implies e_1 = \texttt{<}n\texttt{>} = e_2 \implies v_1 = n = v_2 \implies v_1 \approx_\tau v_2$

E-App Applying the induction hypothesis on the two first premises, and the substitution lemma for the last one gives the equality modulo type of resulting values.

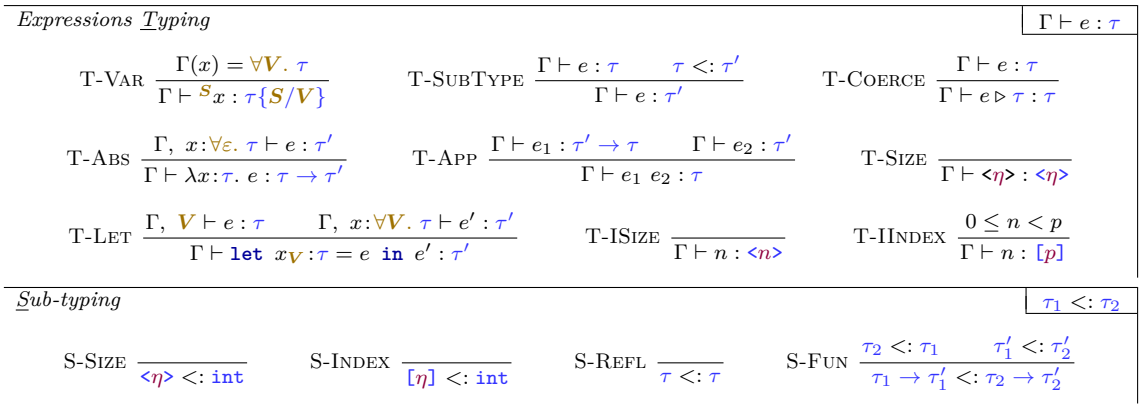


Figure 2.5: *Type system for $\mathcal{L}^{\eta}$, non syntax directed*

E-Coerce *idem*

E-Let We have $e_1 = \texttt{let}\ x_{\boldsymbol{V}_1} : \tau_1 = e_1\ \texttt{in}\ e_1'$ and $e_2 = \texttt{let}\ x_{\boldsymbol{V}_2} : \tau_2 = e_2\ \texttt{in}\ e_2'$. It follows that $e_1' \approx_{\tau} e_2'$ and $e_1 \triangleright \tau_1 \approx_{\tau} e_2 \triangleright \tau_2$. By the substitution lemma, the premise yield equal modulo type values.

It remains to prove that coercions of equal modulo types values and possibly different types yields equal modulo types values. This is established by examining the value's shape:

n For both coercions, one of C-Size, C-Index or C-Int applies. Each of them produces the same result: the original value. Thus results are equal modulo types.

$\lambda x{:}\tau.\ e$ Coercions are reduced with the C-Fun rule, thus both types have the form $\cdot \to \cdot$. The produced abstractions are immediately equal modulo types. □

Corollary 2.4.1 (Observable type independence)**.** *Expressions that are equal modulo types are observationally equivalent:*

$$\forall\ e_1\ e_2,\ e_1 \approx_{\tau} e_2 \implies e_1 \equiv e_2$$

Proof. Given e_1, e_2 such that $e_1 \approx_{\tau} e_2$, and closed expressions $a_1, \ldots, a_k$, we immediately have $e_1\ a_1\ \ldots\ a_k \approx_{\tau} e_2\ a_1\ \ldots\ a_k$. Observational equivalence follows because equality modulo types for integers implies equality: if both applications reduce, and one of the results is an integer, then the other is identical. □

This result ensures that the semantics depends only on size variables. It does not depends on type variables, even if they can be instantiated with refined types that contain sizes. This property matters for the inference because the semantics should not depend on the reconstruction process.

2.2.3 Typing

A type discipline, based on the one of Hindley and Milner [Hin69], filters the terms for which a semantics exists, i.e., expressions that may be reduced to a value or diverge.

Environment Expressions are typed in an environment Γ defined as a pair (Γ_v, Γ_e) where $\Gamma_v \subset \mathcal{V}_\eta \cup \mathcal{V}_\tau$ is the set of bound-size and type variables and Γ_e is a partial map from term variables to type schemes —$\sigma := \forall \boldsymbol{V}.\ \tau$—. In the following, terms are supposed to be named so that no clashes occur. The environment is thus unordered and insertions —$\Gamma,\ x{:}\forall \boldsymbol{V}.\ \tau,\ \boldsymbol{V}$ — assume that added variables —x and $\boldsymbol{V}$— are unbound in Γ.

Judgments The typing judgment —$\Gamma \vdash e : \tau$— reads 'in the environment Γ, the expression e has type τ'. This relation implicitly assumes that τ and e are *well-formed*, i.e., that their free variables are bound in Γ. It is defined alongside the sub-typing relation —$\tau_1 <: \tau_2$— in Figure 2.5.

It is worth mentioning that type equality, used in the S-Refl rule among others, requires a syntactic identity between the sizes appearing in refinements. For instance, types $[\iota]$ and $[2 - \iota]$ are considered different even though they yield equal types when instantiated with $\iota = 1$.

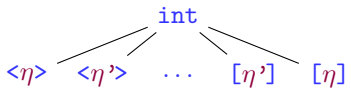
```
let dot = λu. λv.
    fold (+) <_> 0 (map2 (*) <_> u v)                    [ ∀ι. [ι]int → [ι]int → int ]

let mat_vec = λa. λv.
    map (dot u) <_> (transpose a)                        [ ∀ι·κ. [ι][κ]int → [ι]int → [κ]int ]

let mat_mat = λa. λb.
    map (mat_vec b) <_> a                                [ ∀ι·κ·δ. [ι][κ]int → [κ][δ]int → [ι][δ]int ]
```

Figure 2.6: *Usual linear algebra primitives defined with iterators*

Refinements and Sub-typing General dependent type systems such as DML [XP99] provide rich sub-typing relations based on refinement implication, at the cost of static checking undecidability. For that reason as well as inference perspectives, sub-typing is restricted to inserting or dropping refinements (with respect to the variance). Thus, the relations $[\eta] <: \texttt{int}$ and $\texttt{int} \to \alpha <: [\eta]\,\alpha$ are valid, whereas the semantically correct relation $[\iota] <: [\iota + 1]$ is invalid. This flat order between refined types, illustrated bellow, is the key restriction to keep type checking decidable: correction only relies on size equalities, instead of general inequalities on polynomials.



The Types of Constants The only immediate values we consider here are integers. The rules T-ISize and T-IIndex allow any semantically correct refined type whose size is an immediate value. This is needed to state *type preservation*. Otherwise the expression $\texttt{<}n\texttt{>}$ of type $\texttt{<}n\texttt{>}$ would reduce to n, that could not be given the same type. In λ^H, Flanagan [Fla06] faces a similar situation: constants are given singleton types, so that any possible refinements may be derived using sub-typing. Because our sub-typing lattice is not closed by intersection, we directly provide all the possible types.

Integers primitives are handled as external functions over non refined integers only. For instance, addition has the following signature:

```
val (+) : int → int → int
```

Preservation and Soundness This type system enjoys both preservation and soundness. Types are preserved by reduction and well-typed terms have a semantics. Formally:

Theorem 2.5 (Preservation and soundness)**.** *Given an expression e and a type τ such that $\vdash e : \tau$.*
Then there exists a value v such that $e \rightsquigarrow v$. *(Soundness)*
Moreover, $\vdash v : \tau$. *(Preservation)*

Proof. The generic construction for big step semantics set up by Dagnino et al. [Dag+20] allows to establish these results from three local properties on the type system and the semantics (see Appendix 1.1): local preservation, ∃-progress and ∀-progress. To establish them, a key step is the normalization of typing derivations, that confines the instances of the T-SubType rule to specific premises. □

2.3 Inference

Although type annotations are helpful as documentation (e.g., in interfaces), they tend to obfuscate programs as size expressions become larger. They should be inferred. However, pursuing a full and complete type inference as the HM type discipline enjoys [Hin69] is vain: the size language, that allows non-linear arithmetic expressions, will surely cause unsolvable constraints. Despite this, the size relations that occurs in data-intensive applications are often simple, giving the opportunity to omit most of them. Figure 2.6 gives implicitly typed definitions of simple linear algebra operations and their inferred types.

```
let slope = λf:[_] → _. λi: _ . λj: _ .
    (f i - f j) / (i - j)                    [ ∀ι. ([ι] → int) → [ι] → [ι] → int ]

let slope = λf: _ → _. λi:[ι]. λj:[ι].
    (f i - f j) / (i - j)                    [ ∀ι. ([ι] → int) → [ι] → [ι] → int ]

let slope = λf: _ → _. λi:[_]. λj:[_].
    (f i - f j) / (i - j)                    [ ∀ι·κ. (int → int) → [ι] → [κ] → int ]
```

Figure 2.7: *Different type annotations in the* `slope` *example and the inferred type schemes.*

One point must be carefully handled: $\mathcal{L}^\eta$ semantic is specified over *closed typed* terms. Inference must ensure that the semantics of reconstructed terms is fully defined by implicitly typed ones. The *ghost size* issue sketched in Section 2.1 is crucial here. Unconstrained size variables should not get defined during reconstruction. Otherwise, inference could have chosen the semantics. To this aim, inference must ensure that no unnecessary size relations are introduced.

2.3.1 Implicitly Typed $\mathcal{L}^\eta$

As an implicitly typed language, a slight variation of $\mathcal{L}^\eta$ is used: generalization and instantiation places, i.e., $\boldsymbol{V}$ and $\boldsymbol{S}$ in $\mathcal{L}^\eta$ syntax, are omitted. Unlike polymorphism markers, type annotations are still present in implicitly typed terms, but they might contain size and type variables that are unbound. Given a term e, the inference builds a *completion* of e by providing definitions for polymorphism places, i.e., a list of size and type variables that are generalized or used for instantiation, alongside a substitution of unbound size and type variables. In examples, placeholders (_) stand for fresh size or type variables.

2.3.2 Algorithm

Size equality constraints amount to vanishing polynomials. Unlike types, whose unification is structural, these constraints cannot be solved easily. For that reason, instead of building a substitution on the fly as done by Algorithm $\mathcal{W}$ [Mil78], sub-typing constraints and unbound size and type variables are collected by a term traversal and the resulting system is solved at generalization point (i.e., `let`), in the hope of using the simplest constraints to simplify the complex ones. In the context of sub-typing, similar algorithms were proposed, e.g., by Aiken and Wimmers [AW93], where constraints are simplified at generalization points. The constraint collecting algorithm is explained in detail in Appendix 1.2.

Generalization is introduced at let bindings: once the definition has been traversed, sub-typing constraints are solved. As for simple ML, the remaining type and size variables that do not appear in the environment are generalized. Moreover, two extra checks are performed on the generalized variables:

1. They should not appear in remaining constraints.
2. They must appear in the declaration's type.

The former allows to consider unconstrained polymorphism, while the latter detects unconstrained variables. This last check is crucial since a term's semantics may depend on sizes. Section 2.1 gives an example of such an ambiguous term:

```
let even = fold (+) <_> 0 (cst 2)        (Error: Unconstrained size)
```

2.3.3 Principal Typing

Before presenting the constraint resolution strategy, let us focus on a thorn in our side. This type system does not enjoy principal types, i.e., some declarations do not have a most general type scheme. The comparison of type schemes is defined by the *subsumption* relation presented by Peyton Jones et al. [Pey+07]. Informally $\sigma_1 \preccurlyeq \sigma_2$ if and only if any instance of σ_2 may be obtained by instantiating σ_1 and using sub-typing. σ_1 is then *more general* than σ_2. It naturally defines

a notion of equivalence, that amounts for simple ML types (without sizes), to a renaming of type variables. Because size equality is not structural, this relation is larger here. The uni-dimensional convolution defined in Section 2.1 may be given the following type schemes:

$$\texttt{val convolution} : \forall \iota \cdot \kappa.\ [\kappa]\texttt{int} \to [\iota]\texttt{int} \to [\iota - \kappa + 1]\texttt{int}$$
$$\texttt{val convolution} : \forall \iota \cdot \kappa.\ [\kappa]\texttt{int} \to [\iota + \kappa - 1]\texttt{int} \to [\iota]\texttt{int}$$

Any instance of the first is an instance of the second, and reciprocally. More importantly, some terms may be given multiple type schemes that have no common generalization and this must be carefully handled during inference. There are two reasons for this.

1. Polynomial Sizes Allowing more than linear expressions for sizes surely engenders constraints with multiple solutions. Given a function `split`, declared below, that transforms a 1-dimensional array into a 2-dimensional one, its application to an 'array' of size 4 raises several possible types, corresponding to different semantics.

$$\texttt{val split} : \forall \iota \cdot \kappa \cdot \alpha.\ [\iota * \kappa]\alpha \to [\iota][\kappa]\alpha$$
$$\texttt{let mat} = \texttt{split}\ (\lambda i{:}[4].\ 0) \qquad \begin{cases} [1][4]\texttt{int} \\ [2][2]\texttt{int} \\ [4][1]\texttt{int} \end{cases}$$

In such a situation, the underlying constraint ($\iota * \kappa - 4 = 0$) will not be solved (see Section 2.3.4), and inference will fail, asking for more annotations.

2. Sub-typing and Unconstrained Polymorphism Because polymorphism is unconstrained, all refinements are selected at definition. The `slope` function below computes the ratio of images' difference over arguments' difference, assuming suitable arithmetic operators defined over integers.

$$\texttt{let slope} = \lambda f.\ \lambda i.\ \lambda j.\ (f\ i\ \texttt{-}\ f\ j)\ \texttt{/}\ (i\ \texttt{-}\ j)$$

The subtlety comes from the simultaneous applications of f to i and j: should f's domain and both arguments share the same refinement? Indeed, possible type schemes include:

$$\begin{array}{lll} \forall \iota. & (\texttt{<}\iota\texttt{>} \to \texttt{int}) \to \texttt{<}\iota\texttt{>} \to \texttt{<}\iota\texttt{>} \to \texttt{int} & (\sigma^s_s) \\ \forall \iota \cdot \kappa. & (\texttt{int} \to \texttt{int}) \to \texttt{<}\iota\texttt{>} \to \texttt{<}\kappa\texttt{>} \to \texttt{int} & (\sigma^b_s) \\ & (\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int} & (\sigma^b_b) \\ \forall \iota \cdot \kappa. & (\texttt{int} \to \texttt{int}) \to [\iota] \to [\kappa] \to \texttt{int} & (\sigma^b_i) \\ \forall \iota. & ([\iota] \to \texttt{int}) \to [\iota] \to [\iota] \to \texttt{int} & (\sigma^i_i) \end{array}$$

Among them, $\sigma^b_b \preccurlyeq \sigma^s_b$ and $\sigma^b_b \preccurlyeq \sigma^i_b$. Others are incompatible pair-wise (denoted $\not\approx$) for multiple reasons: refinements...

1. are incompatible: $\sigma^s_s \not\approx \sigma^i_i$; $\sigma^b_s \not\approx \sigma^b_i$; $\sigma^b_s \not\approx \sigma^i_i$; $\sigma^s_s \not\approx \sigma^b_i$
2. appear covariant and contravariant: $\sigma^s_s \not\approx \sigma^b_b$; $\sigma^i_i \not\approx \sigma^b_b$
3. impose extra size constraints: $\sigma^s_s \not\approx \sigma^b_s$; $\sigma^i_i \not\approx \sigma^b_i$.

Constrained Polymorphism Coupling sub-typing and unconstrained polymorphism is unusual. The general theory proposed by Aiken and Wimmers [AW93] provides constrained types schemes. Shrinking the constraint set at generalization points is then the key to avoiding an exponential blow-up of constraints [Pot01]. Such systems enjoy the principal types property. In this context, the function `slope` would be given the type scheme:

$$\forall \alpha \cdot \beta \cdot \gamma \mid \alpha <: \texttt{int} \wedge \beta <: \alpha \wedge \gamma <: \alpha.\ (\alpha \to \texttt{int}) \to \beta \to \gamma \to \texttt{int}$$

However, modularity would be sacrificed here, by deferring size constraints resolution to monomorphic instantiations. Coupled with the loss of readability of such types, this is the main reason for keeping unconstrained polymorphism.

Inference and Semantics This issue about principal types is all the more crucial because our semantics is not type erasable. Since sizes have computational content in our language. Inference should ensure that no sizes have been arbitrarily defined. We formalize this in Section 2.3.5.

2.3.4 Constraint Solving

Solving the constraint system aims at extracting from the set of sub-typing constraints a *most general unifier*, i.e., a necessary substitution of the free variables. This is achieved gradually: (i) types (without refinements) are inferred using structural unification; (ii) the necessary refinements of type `int` are selected; (iii) size constraints are solved; (iv) refinements are propagated. Similar stratification has been previously used for inference in extended type systems [XP98; KF07; RKJ08]. However these steps are entangled in our proposal: instead of separating phases across multiple passes, types, refinements and sizes get partially defined at each solving point (`let`), allowing an easier handling of polymorphism than would be possible with disconnected inference passes.

To illustrate our overview of the solving process, three slightly modified versions of the `slope` example used previously are defined in Figure 2.7, where some annotations are added, constraining the refinements at different places.

(i) Types. To begin with, refinements are ignored to build *simple types* that will be made precise in the subsequent phases. By replacing every refinement with `int`, sub-typing relations are turned into equalities. They are solved using structural unification, failing in the usual modalities (e.g., top-level type constructor inequality, cyclic types). This phase generates a *most general unifier* [Mil78]. At that point, Each declaration of `slope` has type $(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$.

(ii) Refinements. The integer types previously derived may now be refined. Each occurrence of `int` in the substitution is replaced by a fresh type variable. Sub-typing constraints are distributed with the S-Fun rule (the usual variance rule), leading to simple constraints containing variables and refined types. The ones of the form $\alpha <: \texttt{[\_]}$, $\alpha <: \texttt{<\_>}$ and $\texttt{int} <: \alpha$ define variable α's refinement while unsolvable constraints such as $\texttt{int} <: \texttt{[\_]}$ lead to errors that are reported to the programmer. Refinements are not propagated further but rather postponed until after size resolution, since adding refinements may introduce constraints between sizes that would otherwise be unrelated. Unconstrained variables are thus replaced by `int`.

In our example, the first version of `slope` gets type $(\texttt{[\_]} \to \texttt{int}) \to \texttt{[\_]} \to \texttt{[\_]} \to \texttt{int}$ while the two others have type $(\texttt{int} \to \texttt{int}) \to \texttt{[\_]} \to \texttt{[\_]} \to \texttt{int}$.

(iii) Sizes. Sub-typing constraints are now distributed again, extracting size equalities, i.e., vanishing polynomials $\eta_1 - \eta_2 = 0$ for constraints of the form $\texttt{<}\eta_1\texttt{>} <: \texttt{<}\eta_2\texttt{>}$ or $\texttt{[}\eta_1\texttt{]} <: \texttt{[}\eta_2\texttt{]}$. The resulting polynomial system C^η is solved, by deriving a *most general substitution*.

Definition 2.6 (Most general substitution)**.** Given a constraint set C, a substitution ρ is *most general* if and only if for any substitution ρ', such that $\vdash C\{\rho'\}$, then there exists a substitution ρ'' such that $\rho' = \rho'' \circ \rho$.

For now, we have implemented a simple resolution strategy that eliminates *isolated variables*, i.e., substituting η for ι when a constraint $\iota - \eta = 0$ exists. The resulting substitution is immediately a most general one. This elementary strategy, complemented with a strategy to decompose size constraints using abstract variables, which we present in Section 2.4.4, works for most of the size constraints we encountered. Adding some annotations helps for the remaining cases. At this point, the three versions of `slope` have respectively the types:

1. $(\texttt{[}\iota\texttt{]} \to \texttt{int}) \to \texttt{[}\iota\texttt{]} \to \texttt{[}\iota\texttt{]} \to \texttt{int}$
2. $(\texttt{int} \to \texttt{int}) \to \texttt{[}\iota\texttt{]} \to \texttt{[}\iota\texttt{]} \to \texttt{int}$
3. $(\texttt{int} \to \texttt{int}) \to \texttt{[}\iota\texttt{]} \to \texttt{[}\kappa\texttt{]} \to \texttt{int}$

In particular, ι and κ sizes are not unified in the last declaration, because sub-typing occurs at each application.

(iv) Propagation. Lastly, refinements are propagated. The aim is to make types more accurate. Among the type variables introduced during refinement inference, those with a unique lower bound — $\texttt{[}\eta\texttt{]} <: \alpha$ or $\texttt{<}\eta\texttt{>} <: \alpha$ — are defined accordingly, i.e., $\alpha := \texttt{[}\eta\texttt{]}$ or $\alpha := \texttt{<}\eta\texttt{>}$.

In the type of the second version, refinements are now equal. They are propagated, yielding the type $(\texttt{[}\iota\texttt{]} \to \texttt{int}) \to \texttt{[}\iota\texttt{]} \to \texttt{[}\iota\texttt{]} \to \texttt{int}$. Conversely, they differ in the third version where f's domain cannot be refined.

Solver-based size resolution The resolution of polynomial size constraints could be delegated to an external solver. However, this would create a dependency on a possibly unpredictable system, not only for type checking but also for the rest of the compilation process, that would heavily depend on sizes.

Moreover, the use of external tools is impractical in the context of safety-critical software, for certification purposes. In particular, the usual solution checking approach, where the certified or proven program asserts the correctness of the solution of an oracle, fails here since the compiler must prove that the provided solution is the most general. This is not as trivial as checking that the given substitution is indeed a solution.

2.3.5 Inference Properties

The expressiveness of the size language allows no hope for complete inference. Nevertheless, it is sound and we expect inference to be *non-specializing*, i.e., that it rejects any term with ambiguous semantics.

Theorem 2.7 (Inference soundness)**.** *Given an implicitly typed expression, if inference succeeds, the reconstructed term is well-typed.*

Proof sketch. The detailed proof is available in Appendix 1.2, alongside a formalization of the inference algorithm. It established the following invariant: for each sub-term and a substitution that solves the constraints gathered by the algorithm, there exists a type derivation for the substituted term, in the same environment. □

Conjecture 2.8 (Inference non-specialization)**.** *Given an expression e, if inference succeeds and produces a closed term e', then for any possible well-typed completion e'' of e, $e' \equiv e''$.*

This proof has not been fully conducted yet. The main difficulty lies in the handling of non-principal types across let bindings, for example the relation between the possible type schemes of our `slope` example is delicate. The intuition is that the type scheme of the second version (where sizes match and the domain of the function is refined) is obtained from the inferred type scheme (with distinct sizes) by adding refinements in place of some `int`s and using instantiation and sub-typing. Hence, all possible type schemes contain at least as many size relations as the inferred one.

We define a *size subsumption* relation in Appendix 1.2 that makes the above intuition precise. To support let bindings, it is necessary to relate the size constraints generated in environments where declarations have type schemes related by this size subsumption relation. However, they introduce different constraint sets whose relations have not yet been described in a satisfactory manner.

2.4 Extensions for a Realistic Array Language

Our simplistic language $\mathcal{L}^\eta$ provides the basis for an ML-like language where sizes are handled in the same way as types, i.e., with polymorphism. This section gives an insight into some extensions that are necessary for a more realistic array language.

2.4.1 Polymorphic Recursion

If recursion is to be supported, it must be polymorphic. Indeed recursive algorithms on arrays usually call themselves on sub-arrays, whose sizes are smaller. Because sizes are fixed by types, a monomorphic recursion would not allow to write them. The Fast Fourier Transform is an example of algorithms where the sizes of arrays vary at each call. Polymorphic recursion has been extensively studied [Mee83; Myc84].

Semantics While adding recursive declarations requires few changes to the implementation, it deeply impacts the formalization we proposed. Diverging terms would now exist and they must be distinguished from blocked ones in the $\mathcal{L}^\eta$ big-step semantics. Fortunately, the Dagnino et al. [Dag+20] formalization was designed for non-deterministic semantics. By giving a non-deterministic evaluation of fix-points (either stopping with an error value or reducing further), the preservation and soundness results (Section 2.2.3) extend to a language with recursion.

Inference As shown by Henglein [Hen93], polymorphic recursion turns inference into an undecidable problem. We follow the classical approach, by considering fix-points monomorphic unless explicitly generalized at declarations.

An extra check is required to ensure that the actual type of the declaration is indeed as polymorphic as the specified one. This amounts to checking the subsumption relation that we defined in Section 2.3.3. This validates *a posteriori* that recursive occurrences of the recursive variables have been correctly instantiated.

2.4.2 Explicit Coercions

The Fast Fourier Transform raises a second issue. It requires decomposing the size of the treated array as a product of sizes. However, apart from fixed sizes, such decompositions cannot be expressed with our size language. A less general form could require a size that is a power of 2, but this is also not expressible with polynomials.

The solution is twofold: (i) allow for *existentially quantified sizes*, that we review in Section 2.4.4 and (ii) provide some *explicit size coercions* that insert casts between sizes. Contrary to the implicit coercions used for semantic purposes, these explicit coercions would be partially checked by the type system. They impose an identical type skeleton, but allow for size mismatches. Explicit coercions identify the size properties that must be verified after typing by other means.

As mentioned by Jay and Sekanina [JS97], coercions may be checked in various ways: at run-time with defensive code, or using alternative formal verification tools. In the context of static sizes, Nielson and Nielson [NN88] proposed a *binding time analysis*, to ensure that coercions (and local existential sizes) are computable at compile-time.

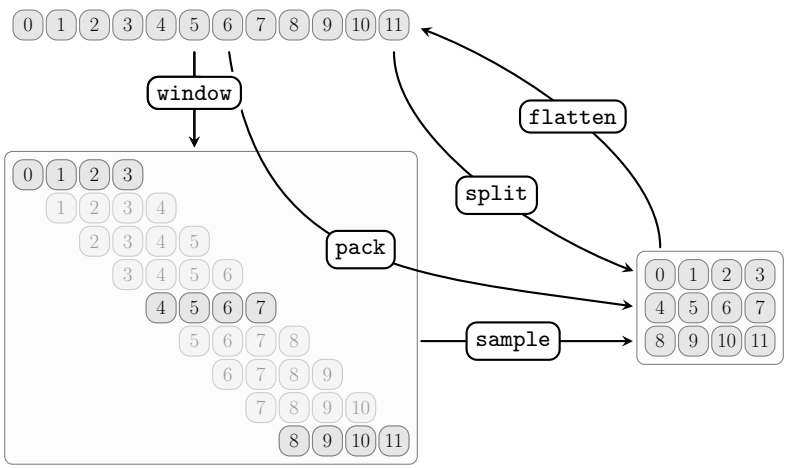
2.4.3 Index Computations and Array Combinators

Numeric operations are defined on unrefined integer values, i.e., of type `int`. Whereas it is straightforward to extend addition and multiplication to singleton types — $\langle\eta\rangle$ —, the task is less simple for indexes, because of the bounds. Instead of overloading scalar operations or providing dedicated ones, we introduce index computations with array operators, i.e., a set of built-in functions that transform arrays. These primitive functions are called *first-order array combinators* (FOACs), a term coined by Henriksen et al. [Hen+17] to emphasize the similarities with iterators, named *second-order array combinators* (SOACs). Indeed, the shift from index computations to FOACs is similar to the shift from explicit indexing and SOACs: combinators describe index computations in an *intensional* way, i.e., at the level of arrays instead of scalars.

In Scade, the following first order array combinators are available:

```
val transpose : ∀ι·κ·α. [ι][κ]α → [κ][ι]α
val reverse   : ∀ι·α. [ι]α → [ι]α
val concat    : ∀ι·κ·α. [ι]α → [κ]α → [κ+ι]α
```

To efficiently describe algorithms such as convolution, we propose to extend this family with the few additional operators depicted below with their type schemes. As illustrated, the `pack` operator is the composition of the `window` and `sample` ones.



```
val window : ∀ι·κ·α. <κ> → [ι+κ-1]α → [ι][κ]α
val sample : ∀ι·κ·α. <κ> → [ι*κ-κ+1]α → [ι]α
val split  : ∀ι·κ·α. <κ> → [ι*κ]α → [ι][κ]α
val flatten: ∀ι·κ·α. <κ> → [ι][κ]α → [ι*κ]α

let pack = λs. λx.
      sample s (window <_> x)          [ ∀ι·κ·δ·α. <δ> → [ι*δ-δ+κ]α → [ι][κ]α ]
```

The `window` function (see Section 2.1) builds a matrix whose rows are slices of the input array. The `sample` function extracts one element out of every κ, selecting both ends of the array. The size of the input array must thus be a multiple of κ plus 1. Composing these functions defines a general sampling operator `pack`. It selects ι slices of size κ that are uniformly distributed and cover both ends of the input array, whose size is obtained by considering a sampling step δ. Note that the size arguments of `sample` and `pack` are necessary since the associated variable might not be deduced from the array size (because of non linear sizes). Although redundant, the `split` primitive carries extra information by defining a bijection between arrays. To illustrate the general sample operator `pack` in a less specific case, here is an example application that leads to overlapping slices (the coercion forces the result size, hence defining the ι and κ generalized sizes):

$$\texttt{pack <2> } (\lambda i\texttt{:[7]}.\ i) \triangleright \texttt{[3][3]int} \rightsquigarrow \begin{bmatrix} 0 & 1 & 2 \\ 2 & 3 & 4 \\ 4 & 5 & 6 \end{bmatrix}$$

The proposed first-order array combinator may be related to explicit index computations. The `window` operator encodes an index addition and the `sample` operator introduces a multiplication of an index by a size. We did not think of any applications of index multiplication, that would introduce irregularly spaced accesses.

2.4.4 Abstract Sizes

To ensure generalization, we may want to impose that some size and type variables be unconstrained. This is provided by abstract sizes and types. For our type system, we found two use cases, that we introduce in Figure 2.8.

e	::=	...	***Expressions***
	|	$\lambda \boldsymbol{V}:\boldsymbol{S}.\ e$	local abstraction
	|	$\texttt{let size}\ \iota = e\ \texttt{in}\ e$	existential quantification

Expression Typing (extension) $\quad \Gamma \vdash e : \tau$

$$\text{T-Abstract}\ \frac{\Gamma,\ \boldsymbol{V} \vdash e : \tau}{\Gamma \vdash \lambda \boldsymbol{V}:\boldsymbol{S}.\ e : \tau\{\boldsymbol{S}/\boldsymbol{V}\}} \qquad \text{T-Exist}\ \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma,\ \iota \vdash e_2 : \tau}{\Gamma \vdash \texttt{let size}\ \iota = e_1\ \texttt{in}\ e_2 : \tau}$$

Figure 2.8: *Existential quantification and local abstraction.*

The local abstract types and sizes construct — $\lambda \boldsymbol{S}:\boldsymbol{V}.\ e$ — borrows the syntax of application. It binds in e unbound size and type variables $\boldsymbol{S}$. As the typing rule T-Abstract shows, these size and type variables are substituted with some sizes and types $\boldsymbol{V}$. Inside the scope of local abstraction, the size and type variables are incompatible with the size and type they are substituted with. In OCaml, such local abstract types are notably used to help inference with GADTs [GR13]. For our simple type system without such advanced features, local abstract sizes are mainly useful to help type inference.

Existential quantification — $\texttt{let size}\ \iota = e_1\ \texttt{in}\ e_2$ — allows to introduce a size whose value is determined by the result of an integer expression. This size variable is considered abstract in e_2 and it cannot be unified with any other size. Its use is restricted. So that the rule T-Exist applies, the type of e_2, τ must be well-formed in environment Γ, that does not contain the size ι. Hence, this size must not appear in τ_2. Existential quantification has no equivalent for types since we do not allow for post-typing defined types. For sizes, it gives a backdoor to the limited expressiveness of our polynomial language. Moreover, we use them intensively in the iteration construct of MADL: the value of the iteration index is represented with an existentially quantified size.

Inference Both constructs induce a notion of scope for size and type variables. Inference must be adjusted to reflect this and to ensure that variables do not escape their scope. The constraint collecting algorithm builds a tree of scopes[3] where each scope contains the abstract (either local or existentially quantified) sizes and types alongside the typing constraints of the scope.

For types, whose constraints are resolved with first-order unification, detecting escaping abstract types amount to checking that the free variables of a given scope are not substituted with sizes or types that contains abstract variables from an inner scope. In OCaml, this leads to the infamous error *type constructor would escape its scope*:

```
let neq (type t) x (y: t) =                                   (* OCaml *)
  x <> y                                                  [ ∀α. α → α → bool ]

let neq x (type t) (y: t) =
  x <> y                    (Error: The type constructor 'a' would escape its scope)
```

The situation is less favorable for sizes. To highlight this, consider the following constraint:

$$7 * (\iota - \kappa_i) - \delta_o = 0$$

where ι is a abstract size variable of the inner scope, κ_i is a free variable of the inner scope and δ_o is a free variable of the outer scope. Using our resolution strategy, the only possible substitution is $\delta_o := 7 * (\iota - \kappa_i)$. Does this mean that the abstract size ι escapes its scope? This will depend of the value of κ_i. If κ_i is substituted with $\iota - 6$, the former substitution just sets $\delta_o := 42$.

Instead of finding sufficient conditions for abstract sizes to escape their scope, we use abstract sizes to decompose polynomial constraints. The constraint $\eta_1 * \iota + \eta_2 = 0$ is simplified into the conjunction of constraints $\eta_1 = 0 \ \wedge \ \eta_2 = 0$ on the condition that ι may not be captured by η_2. This simplification preserves the set of solutions. Indeed, given a size substitution ρ such that $(\eta_1 * i + \eta_2)\{\rho\} = 0$ and $i \notin \mathcal{FV}(\ \eta_2\{\rho\}\ )$, we have (1) that allows to deduce (2):

$$0 = (\eta_1 * i + \eta_2)\{\rho\}\{0/i\} = \eta_2\{\rho\} \quad (1)$$
$$0 = (\eta_1 * i + \eta_2)\{\rho\}\{1/i\} = \eta_1\{\rho\} \quad (2)$$

This simplification may be generalized to higher order polynomial constraints in the abstract variable, either directly or by repeating this process on the constraint $\eta_1 = 0$. This exact form of constraints arises from recursive array aggregation.

Undeclared variables OCaml provides a very practical mechanism to relate type annotations at different places of the code: undeclared type variables, whose syntax is $'a$. The scope of these undeclared size variables is that of the top-level declaration. Hence the following version of our `neq` function is rejected in OCaml.

```
let neq (type t) (x: 'a) (y: t) =                             (* OCaml *)
  x <> y                    (Error: The type constructor a would escape its scope)
```

For MADL, where abstract sizes are frequently used and size annotations may be required to help type inference solve constraints, we provide similar undeclared sizes and types, but we adopt another convention: they are part of the inner-most scope that contains all their occurrences in the top-level declaration. Hence we allow captures of abstract sizes or types in undeclared variables as long as they appear in the scope of these abstract variables.

Conclusion

The type system we propose emphasizes polymorphism: sizes are treated in a similar way to types. The analogy between sizes and types goes beyond the basic generalization and instantiation process: it allows to incorporate more advanced features such as local abstract sizes.

Sizes are introduced in types with the help of two refinements of the integer type. A singleton type `<η>` and an interval one `[η]`. The decidability of our type system is based on a fundamental restriction, our hierarchy of types is flat. Distinct refined types are pairwise incompatible. Because of this, only size equalities are considered, instead of inequalities. This kind of constraint is easily

[3] Sibling scopes may actually be merged, allowing to only consider a list of scopes.

checked with multivariate polynomials since they have a normal form. In our opinion, the language of sizes, multivariate polynomials, gives a relevant compromise between expressiveness and formal manipulation. It can describe most array transformations and it allows for size inference, although it is incomplete.

We impose a second restriction: polymorphism is simple, i.e., unconstrained. As a consequence, our type system does not have principal types. There are two reasons for this. The polynomial size constraints and the choice of refinements. While this may seem like a high cost for keeping polymorphism simple, this restriction is fundamental. By selecting the refinements, it ensures that size resolution is modular, i.e., type scheme contains no delayed size constraints. This full knowledge of size is a basis for our memory-aware compilation. Moreover, using a dedicated array type instead of functions with bounded domains limits the second source of non-principal types (select refinements).

As a distinctive feature of our proposal, we allow expressions to depend on sizes (the <η> construct). The goal is to allow size parameters of SCADE to be omitted, without having to distinguish between the mandatory and optional ones. With such a size expression <η>, the semantics is no longer type erasable. As a consequence, care must be taken to detect all the possibly unconstrained variables, because they could lead to ambiguous programs.

Our type system does not target full verification of arrays. It shrinks the remaining checks to some size inequalities. We have not yet formalized this part. More importantly, this type system gives a formal description of the sizes of arrays which is central information for the subsequent compilation process. It is used in particular to describe memory locations.

3 Projection Functors

Introduction

Array safety benefits from *intensional* operations, like `map`, `transpose`, that obviate the need for explicit indexing. This declarative style favors the emergence of intermediate data such as transposed matrices or slices. These values should not be constructed in the generated code. On the other hand, the target language (C) gives a lower level view of memory where compound data are manipulated *extensionally*, namely with explicit indexing. The following question arises: *how should the catalog of intensional operations over compound data be lowered to extensional manipulations?*

A black-box approach would consist in handling these operations as built-in libraries that are either linked to actual implementations or handled in dedicated ways by the back-end. This is convenient for later compilation passes — new constructs are essentially described by their types and the intermediate representations may remain fully intentional — but it gives poor compilation perspectives: the invariants and properties that are derived on earlier, and hence more abstract, representations are hardly usable in the back-end. Moreover, some compilation stages, like scheduling, strongly limit the transformations that are possible in the subsequent steps.

For these reasons, we prefer to convert intensional operations to explicit indexing early in the compilation pipeline. The intermediate representations are therefore slightly weaker, allowing for some erroneous programs, but the available information is much more precise, in particular for memory management. Moreover, compiler complexity is not sacrificed here since explicit indexing offers a unified representation for both array and tuple operations.

The central element of this representation, named *projection functor*, describes the embedding of one data-structure in another, in particular for arrays. Projection functors serve two purposes that have guided their definition by imposing some operations and properties. We review them briefly.

Views Computations over arrays are expressed with *iterators*, that provide predefined access schemes. For advanced operations, data has to be *presented* in an appropriate form, e.g., by transposing, reversing or slicing a given array. These intermediate results contain the same data organized differently. They amount to a *static* shuffling of the original data. Instead of relying on fragile optimizations to eliminate these cumbersome temporary values, our language uses projection functors to define *views*, i.e., values of different forms that share the same data, eliminating the need for allocation and copies.

For safety reasons, views are not transparently available in the language. Rather, they are the internal representation of some *correct* primitives that give a somewhat indirect but safe way to introduce valid index computations, both for reading and writing arrays. This approach is only possible because the following operations on projection functors are available:

- *Composition.* Complex views are defined by composing simple primitives. This builds on a similar composition at the level of projection functors. In particular, this operation allows simplifications like `transpose` $\circ$ `transpose` $=$ `id`.
- *Classes of projection functors.* The data-flow aspects of our language rely on the *injectivity* and *surjectivity* properties of projection functors. The former ensures uniqueness of definitions while the latter guarantees completeness. *Invertibility*, a slightly stronger property than the conjunction of the above ones, also plays an important role for memory allocation inference.

Interferences Distinct data that are stored at the same location may not be alive at the same time. This tension between memory allocation and scheduling is at the heart of our memory-aware declarative language: memory specification is turned into scheduling constraints. It introduces location sharing with dedicated annotations and specific constructs, such as the aforementioned views. Memory reuse is also a prerequisite for expressing *in-place updates*, which allows to mutate parts of a compound data without having to first copy it.

Because compilation will fail if scheduling is impossible, the finer the description of dependencies, the more advanced the expressible memory allocations. Dependencies amount to testing whether memory locations, described by projection functors, intersect. It will be conservatively approximated by testing *inclusion* and *disjunction*.

3.1 Overview

To guide the reader through the complete formalization, we give an intuitive introduction to projection functors and their manipulations. Although we use an OCaml-like syntax, the following expressions are not functions of our programming language. They represent rather objects that are manipulated by the compiler. Figure 3.1 illustrates an example of a functor that relates a pair of arrays to their concatenation. We progressively present the various elements of the diagram below.

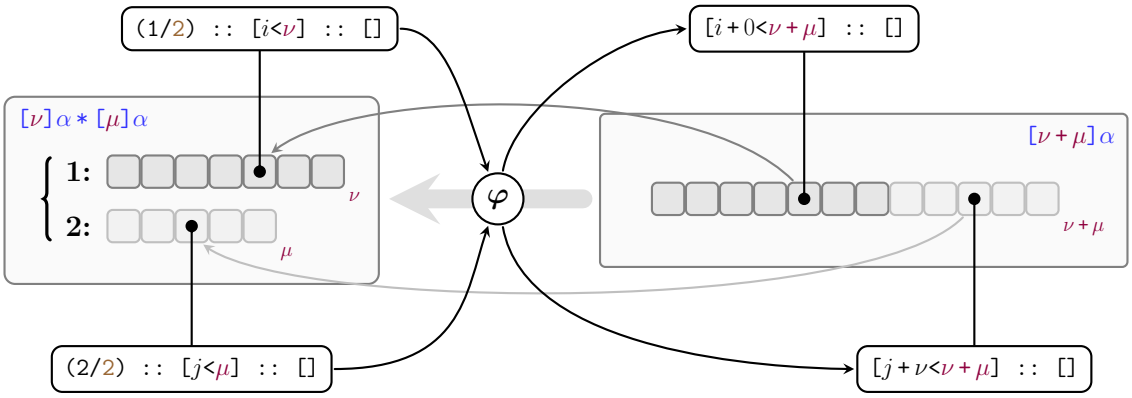


Figure 3.1: *The* $\varphi = \texttt{cut}\ \nu\ \mu\ \alpha$ *projection functor*

Compound Data Functors aim at expressing static manipulations of *compound data*, also called *data-structures*, which are values built from a collection of smaller possibly compound data. In this work, we consider two kinds of data-structures: tuples and arrays. The former are characterized by their *arity*, i.e., the number of components, while the latter are described by their *size*. In Figure 3.1, the box on the left — $[\nu]\alpha * [\mu]\alpha$ — represents a pair, e.g., a tuple with arity 2, whose components are arrays of size ν and μ, while the box on the right — $[\nu+\mu]\alpha$ — depicts a single array, of size $\nu+\mu$. This discussion is independent of how such data would be stored. In this chapter, we only focus on mapping between the structure of compound data.

Projections Compound data are accessed with *projection lists*. Some of them are represented in the rectangular tags in Figure 3.1. Each projection addresses an element of a data structure and contains some structural information about the compound data: `(1/2)` denotes the first component of a tuple of arity 2 and $[i<\nu]$ denotes the ith element of an array of size ν, assuming that i is a valid index, i.e., $0 \leq i < \nu$. With a more usual syntax, the top-left projection — `(1/2) :: [i<ν] :: []` — corresponds to the access `p.1[i]`, where `p` names a value of type $[\nu]\tau * [\mu]\tau$. Annotating projections with structural information allows to manipulate them independently of data while keeping track of types. Projections can be viewed as functions from the type of the compound data to the type of its elements, as depicted in Figure 3.2. These projections can be composed.

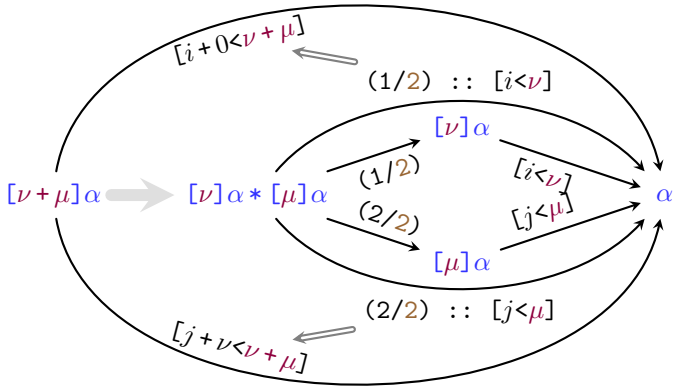


Figure 3.2: *A typing look at the* `cut` *projection functor*

3.1.1 Projection Functors

Projection functors map projections that are applicable on one compound data-structre to projections on another data-structure, independently of the actual data. In the type-based representation of Figure 3.2, they are depicted by the double arrows. In this overview, projection functors are described with a restrictive form of pattern matching, in an OCaml-like syntax that maps lists of projections. The functor of Figure 3.1, represented by φ, is defined by:

```
let cut ν μ α = function
  | (1/2) :: [i<ν] :: l_α → [i+0<ν+μ] :: l_α
  | (2/2) :: [i<μ] :: l_α → [i+ν<ν+μ] :: l_α
```

The `cut` functor is generic in sizes ν, μ and type α. It expects a pair projection — `(k/2)` — followed by an array projection of size ν — $[i<\nu]$ — (resp. μ — $[i<\mu]$ —) for the first (resp. second) component. In both branches, it produces an array projection of size $\nu+\mu$ with different index expressions and transmits the subsequent projections — l_α — unaltered. This l_α is a 'pattern' that matches any valid projection on type α. The index expressions are multivariate polynomials.

This projection functor actually maps projections of type $[\nu]\alpha * [\mu]\alpha$ to projections on data of type $[\nu+\mu]\alpha$: this allows to access a single array of size $\nu+\mu$ as a pair of its left and right parts. Indeed, the index of the second component — the `(2/2)` branch — is shifted by the size ν of the first array: $[i+\nu<\nu+\mu]$.

Views Projection functors allow to build *views* on data. A projection functor φ that maps projections from a type τ_v to a type τ_d transforms a data of type τ_d into a data of type τ_v by mapping accesses to the object of type τ_v (the result) to accesses to the argument (of type τ_d). The action of projection functors is depicted in Figure 3.1 by the right-to-left arrows. The elements of the view (on the left) are defined by the elements of the data (on the right) at the projection given by the functor. This reciprocity between projection functors and views will be used throughout this chapter. Unless explicitly specified, we adopt a data-centeric point of view for naming, typing and composition order. Hence the name of the above `cut` functor that is given the type $[\nu+\mu]\alpha \to [\nu]\alpha * [\mu]\alpha$ even though it maps projections of type $[\nu]\alpha * [\mu]\alpha$ to type $[\nu+\mu]\alpha$. The inverse transformation, that would represent a concatenation, cannot be expressed as a projection functor because indexes are not handled uniformly. Some of them are mapped to the first component of the pair and the others to the second component.

Views are weaker than data. Their compound sub-parts cannot be referred to because projection functors map the *maximal projections* of a type, i.e., the projections that access atomic components, whose types are either abstract or scalar. For instance, applying the `cut` functor to the projection list `(1/2) :: []` cannot be reduced further.

Correctness Two restrictions apply to our 'pattern matching'. Tuple projections are matched with *immediate integers* (`[1<2]`) while array projections are matched with *index variables* ($[i<\eta]$). In order to compare projection functors, index expressions are limited to integer polynomials over size and index variables. Matched and produced projections contain structural information: the size of arrays or the arity of tuples. To be valid, additional properties are required:

- *Correctness.* Array projections (e.g., `[`$i+\nu$`<`$\nu+\mu$`]` in the `cut` functor) must be valid: the index expressions ($i+\nu$) must evaluate to a positive integer strictly less than the sizes (ν), regardless of the values of the index variables (i). This cannot be checked in general.
- *Exhaustiveness.* The projections over tuples are matched separately. The projection functor must cover all cases, which can be checked easily.
- *Consistency.* The type of both matched and produced projections must be compatible in each branch. It includes the type of transmitted projections — l_α — and the structural information of projections.

Hence, the following projection functors are respectively incomplete (the case for the third component is missing) and inconsistent (the sizes of array projections disagree):

```
let incomplete = function
  | (1/3) :: l_int → [0<3] :: l_int
  | (2/3) :: l_int → [1<3] :: l_int
```

```
let inconsistent = function
  | (1/3) :: l_int → [0<1] :: l_int
  | (2/3) :: l_int → [1<2] :: l_int
  | (3/3) :: l_int → [2<3] :: l_int
```

Disclaimer: the limits of our `function`-like syntax *Warning*

This OCaml inspired syntax is misleading on several points, although we think it gives an intuitive insight into projection functors:

- Functor arguments are *sizes* and *types* rather than values. A closer OCaml-ish syntax for the `cut` ν μ α functor would use local abstract types and sizes: `let cut (size` ν μ`) (type` α`) = function`.
- Projections are not homogeneous lists. Every projection conveys some structural information (size, arity). For tuples, the type of the tail projection list depends on the type of the head projection, like `[`ν`]`α and `[`μ`]`α in the `cut` example.

Examples Let us take a tour of simple array projection functors. The two following ones, `single` and `extract`, respectively insert or remove dimensions.

```
let single α = function
  | [i<1] :: l_α → l_α
```

```
let extract α = function
  | l_α → [0<1] :: l_α
```

They are mutually inverse, although this property is not directly readable from the projection functors because indexes are matched with variables. It relies on the set of possible values of type `[1]` which is the singleton $\{0\}$. Transposition is an example of an auto-inverse functor:

```
let transpose ν μ α = function
  | [j<μ] :: [i<ν] :: l_α → [i<ν] :: [j<μ] :: l_α
```

The next projection functor gives a matrix view of a vector, i.e., a unidimensional array. As for the `cut` functor, the index expression — $i*\kappa+j$ — captures a size parameter:

```
let split ν κ α:  = function
  | [i<ν] :: [j<κ] :: l_α → [i*κ+j<ν*κ] :: l_α
```

Here as well, the `split` cannot be inverted for a different reason. In this case, the inverse `flatten` operation, that builds a vector from a matrix, would associate to an array projection `[`$i<\nu*\kappa$`]` the projections `[`$i/\kappa<\nu$`]` and `[`$i\%\kappa<\kappa$`]`. Contrary to `concat`, accesses are still regular since their expressions are uniform with respect to the index value. However, the integer division ($/$) and modulo ($\%$) operations are not allowed by our index language, which requires multivariate polynomials.

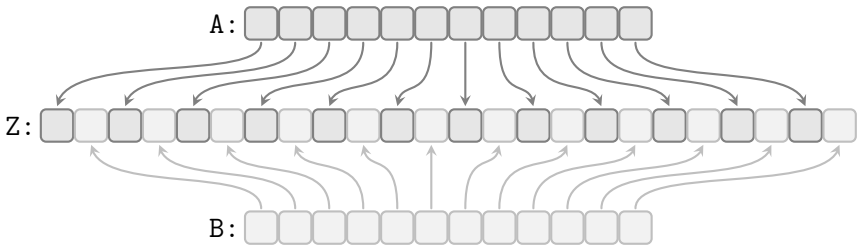
The index language *Remark*

The choice of a restricted index language is grounded in the following observations. (i) To define allocation, the projection functors must be comparable, even in the presence of formal variables. (ii) Projection functors represent access computations that might be duplicated, they should be efficient to calculate.

3.1.2 Composition

Under typical typing assumptions, correct projection functors can be chained together to build new valid mappings. This is at the heart of our memory description. The language provides a few built-in correct views that are composed to define new ones. This alleviates having to ensure their correctness, which depends on hardly provable polynomial inequalities.

We illustrate this process with an *array interleaving* projection functor, that zips two equally sized arrays `A` and `B` into an array `Z` that is twice as long. In the illustration below, we represent an array as a set of data that is continuously addressed by a range of integers, independently of any concrete memory mapping.



Used appropriately, the array operations we have already discussed allow to define array interleaving as a composition of functors. Figure 3.3 illustrates the various steps.

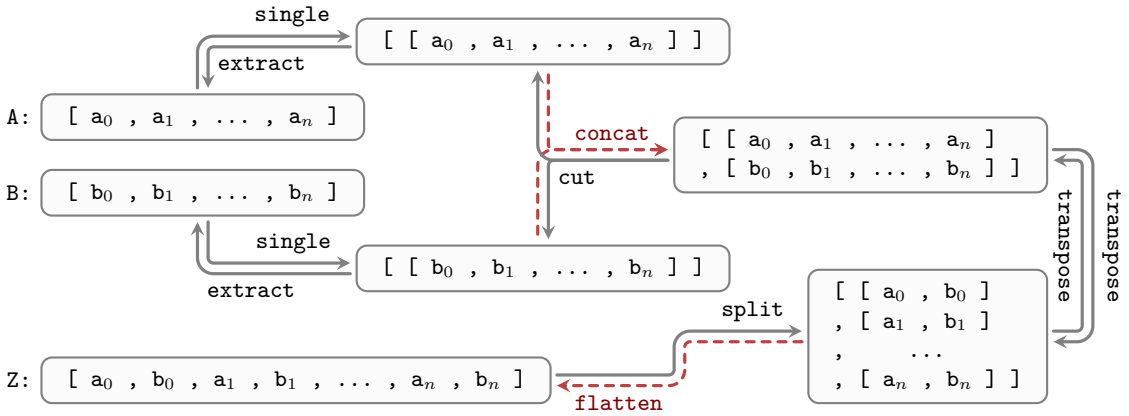


Figure 3.3: *Decomposition of array (un)zipping. ($\nu = n+1$)*

The dashed arrows (`concat` and `flatten`) depict array operations that cannot be translated into projection functors. In this example, both impossible functors flow in the same direction, from inputs to outputs. Hence, the reverse operation (`unzip`) is defined by the following composition (in a data-centeric order):

$$\texttt{unzip}\ \nu\ \alpha\ :=\ \texttt{split}\ \nu\ 2\ \alpha \gg \texttt{transpose}\ 2\ \nu\ \alpha \gg \texttt{cut}\ 1\ 1\ \alpha \underset{(1/2)}{\gg} \texttt{extract}\ \alpha \underset{(2/2)}{\gg} \texttt{extract}\ \alpha$$

From a projection point of view, these functors must be applied the other way round. For instance, the first composition —$\texttt{c}_1\ \nu\ \alpha\ :=\ \texttt{split}\ \nu\ 2\ \alpha \gg \texttt{transpose}\ \nu\ 2\ \alpha$— is obtained by β-reducing the matched projections of the left argument with the produced projections of the right one. The substitution is trivial in this case:

$$\begin{aligned}
\texttt{transpose}\ 2\ \nu\ \alpha = \texttt{function}\ [j{<}2] :: [i{<}\nu] :: l_\alpha &\to [i{<}\nu] :: [j{<}2] :: l_\alpha \\
\texttt{split}\ \nu\ 2\ \alpha = \texttt{function}\ [i{<}\nu] :: [j{<}2] :: l_\alpha &\to [2*i+j{<}2*\nu] :: l_\alpha \\
\texttt{c}_1\ \nu\ \alpha = \texttt{function}\ [j{<}2] :: [i{<}\nu] :: l_\alpha &\to [2*i+j{<}2*\nu] :: l_\alpha
\end{aligned}$$

Composing further with the `cut` $1\ 1\ \alpha$ functor, we obtain the following projection functor of type $[2*\nu]\alpha \to [1][\nu]\alpha * [1][\nu]\alpha$.

$$\begin{aligned}
&\texttt{function} \\
&\mid (1/2) :: [j{<}1] :: [i{<}\nu] :: l_\alpha \to [2*i+j{<}2*\nu] :: l_\alpha \\
&\mid (2/2) :: [j{<}1] :: [i{<}\nu] :: l_\alpha \to [2*i+j+1{<}2*\nu] :: l_\alpha
\end{aligned}$$

The extra dimensions of each component of the pair — `[`j`<1]` — are dropped with *local compositions*, denoted $\ldots \underset{(1/2)}{\gg}$ `extract` $\alpha \underset{(2/2)}{\gg}$ `extract` α, that apply the projection functor `extract` α to the projections occurring after `(1/2)` (resp. `(2/2)`):

```
let unzip ν α = function
  | (1/2) :: [i<ν] :: l_α → [2*i+0<2*ν] :: l_α
  | (2/2) :: [i<ν] :: l_α → [2*i+1<2*ν] :: l_α
```

3.1.3 Non-representable Transformations

One might argue that our zipping example has been carefully chosen to feature this fortunate alignment of expressible functors. Indeed less favorable situations exist. Although this expressiveness limitation does not matter in the present chapter, it is fundamental for our description of memory allocation. To illustrate this, let us introduce a `reverse` projection functor:

```
let reverse ν α = function
  | [i<ν] :: l_α → [ν-i-1<ν] :: l_α
```

As depicted in Figure 3.4, the array reversal operation, that is represented by the above projection functor, allows to define a matrix rotation using the `flatten` and `split` operations. Non-representable transformations (`flatten`) appear here in both directions.

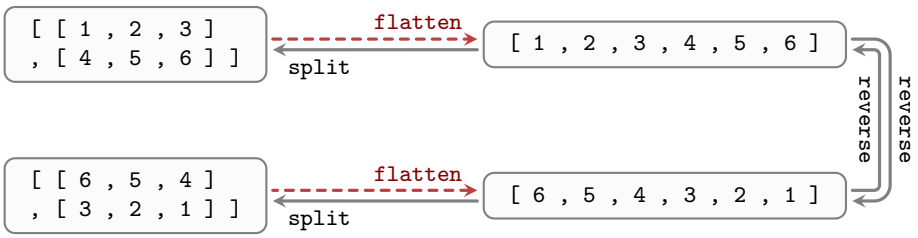


Figure 3.4: *Matrix rotation*

Defined in this way, this transformation cannot be represented by composing projection functors in one direction or the other because `flatten` is not representable. However, both input and output matrices can be described as views over one of the intermediate vectors. For the allocation, this amounts to introduce a memory storage for one of these vectors.

A more clever matrix rotation *Remark*

For this simple example, a better definition uses only invertible operations:

`reverse` ν `[`μ`]`$\alpha \gg$ `transpose` ν μ $\alpha \gg$ `reverse` μ `[`ν`]`$\alpha \gg$ `transpose` μ ν α

The reverse situation is more problematic. It is illustrated in Figure 3.5 that shuffles a vector. If the non-representable transformations flow from intermediate result to the arguments and results, introducing a memory storage does not help.

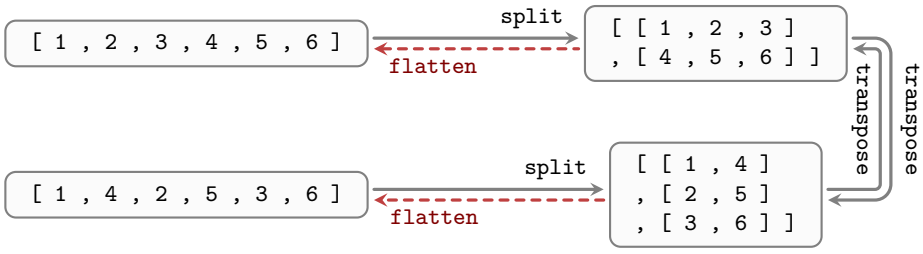


Figure 3.5: *Vector shuffling*

For the allocation, such situation may lead to an error. A solution lies in adding a *computation*, i.e., a copy, between the two matrices. The constraints between the direction of representable transformations, memory and computations drive memory allocation inference. We elaborate on the underlying mechanisms in Chapter 6.

3.2 Compound Data

Before diving into the formalization of projection functors, we first describe how compound data, namely arrays and tuples, and their associated projections are represented. As in Chapter 2, array sizes — $\eta \in \mathcal{S}$ — are described by multivariate polynomials, whose variables are ranged over by ν, μ. The two kinds of compound data are handled differently.

- Arrays gather *homogeneous* data, i.e., their elements are of the same type. This restriction allows to access arrays dynamically, since elements can be handled in a uniform way. Arrays are structurally described by their size.
- Tuples are built out of *heterogeneous* data. They can only be statically accessed. Tuples are described by their arity, i.e., the number of components.

			Sizes
η	::=		
	|	$\nu,\ \mu,\ \ldots$	variable
	|	$0,\ 1,\ \ldots$	constant
	|	$\eta + \eta$	sum
	|	$\eta * \eta$	product

Figure 3.6: *The sizes of arrays*

3.2.1 Projections

Compound data aggregates values that are accessed using *single projections*. In order to keep track of the structure, they are strongly typed. Beside the sub-data they refer to, single projections convey some information about the compound data to which they may apply, namely size and arity.

Single projections Both kinds of compound data — arrays and tuples — are accessed with dedicated *single projections*. Their properties and notations are summarized in Figure 3.7.

	Arrays	**Tuples**
Structural information	Size η (number of elements)	Arity $\mathtt{m}$ (number of components)
Sub-parts	Homogeneous	Heterogeneous
Projections	Dynamic	Static
Syntax	$[e<\eta]$	$(\mathtt{k}/\mathtt{m})$
Domain	$[\eta]$ $\{\ e \in \mathcal{S} \mid \forall \Gamma,\ \mathcal{C} \implies 0 \leq e < \eta\ \}$	$(\mathtt{m})$ $\{\ \mathtt{k} \in \mathbb{N} \mid 1 \leq \mathtt{k} \leq \mathtt{m}\ \}$

Figure 3.7: *Summary of the compound data, single projections and notations*

- Arrays are dynamically addressed with *index expressions*, ranged over by $e \in \mathcal{S}$. These index expressions resemble sizes. They are integer terms that will be formally manipulated, e.g., compared). Moreover, they might depend on sizes. A natural candidate for this language is the set of multivariate integer polynomials, where variables are either size or index ones. The free variables of index expressions (and sizes) are gathered in an environment Γ coupled with a set of polynomial constraints $\mathcal{C}$. The set of correct η-sized array projections, denoted $[\eta]$, contains the polynomials that evaluate to positive integers strictly less than η under $\mathcal{C}$ constraints, i.e., $\{\ e \in \mathcal{S} \mid \forall \Gamma,\ \mathcal{C} \implies 0 \leq e < \eta\ \}$. For $e \in [\eta]$ the array projection $[e<\eta]$ designates the element at index e of an array of size η.
- Tuples are statically addressed with *immediate integers*, ranged over by $\mathtt{k} \in \mathbb{N}$. The set of correct $\mathtt{m}$-ary tuple projections, denoted $(\mathtt{m})$, contains the integers from 1 to $\mathtt{m}$, i.e., $\{\ \mathtt{k} \in \mathbb{N} \mid 1 \leq \mathtt{k} \leq \mathtt{m}\ \}$. For $\mathtt{k} \in (\mathtt{m})$, the tuple projection $(\mathtt{k}/\mathtt{m})$ designates the $\mathtt{k}^{\text{th}}$ component of an $\mathtt{m}$-ary tuple.

Projections Nested compound values are addressed by chaining single projections. A general *projections* is naturally represented by a list of single projections.

Definition 3.1 (Projection space)**.** The set of *single projections* $\mathcal{P}_u$ gathers all the valid single projections over arrays and tuples. The *projection space*, denoted $\mathcal{P}$, is the set of finite lists of single projections:

$$\begin{aligned}\mathcal{P}_u &:= \bigcup \begin{array}{l}\{\ \texttt{[}e\texttt{<}\eta\texttt{]}\ \mid\ \eta \in \mathcal{S},\ e \in \texttt{[}\eta\texttt{]}\ \} \\ \{\ \texttt{(k/m)}\ \mid\ \texttt{m} \in \mathbb{N},\ \texttt{k} \in \texttt{(m)}\ \}\end{array} \\ \mathcal{P} &:= \mathcal{P}_u^{(\mathbb{N})}\end{aligned}$$

Projections are denoted by list notations: ε for empty ones, $p \cdot l$ for non-empty ones and $l \bullet l'$ for concatenation.

Projections *Example*

The projection $\texttt{[}\eta\texttt{-1-}i\texttt{<}\eta\texttt{]} \cdot \texttt{(1/3)} \cdot \varepsilon$ denotes the 1^{st} component of the $(\eta - 1 - i)^{\text{th}}$ element of an array of triplets of size η, assuming that the variables of η and the index i are indeed declared in the environment Γ, and the constraint $i \in \texttt{[}\eta\texttt{]}$ is part of $\mathcal{C}$.

What about records?

As far as memory is concerned, records look similar to named tuples: heterogeneous data that are accessed statically. However, depending on whether they are *structural* or *named*, i.e., whether they are compared by fields or by names, records introduce several difficulties (e.g., sub-typing) that have not been considered yet.

3.2.2 Data-types

The projection space is defined independently of any concrete types. However, for given data, only a subset of these projections may apply. First, let us define the syntax for compound types, $\tau \in \mathcal{T}$. As depicted in Figure 3.8, only variables, arrays and tuples are considered. Scalar types such as `int` or `bool` are handled as type variables.

			Types
τ	::=		
	|	$\alpha,\ \beta,\ \ldots$	variables
	|	$\texttt{[}\eta\texttt{]}\tau$	array
	|	$\tau\ \texttt{*}\ \ldots\ \texttt{*}\ \tau$	tuple

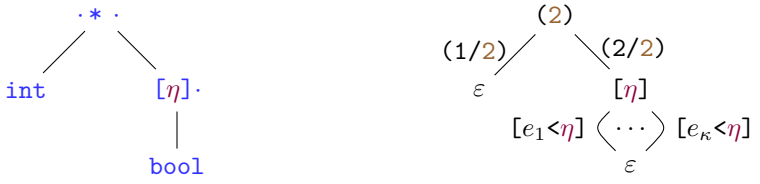
Figure 3.8: *The pure data types*

Definition 3.2 (Domain)**.** Given a type τ, its *domain* — $\pi(\tau) \subset \mathcal{P}$ — denotes the set of projections the address parts of data of type τ. The set of non-trivial projections is denoted $\pi^\star(\tau)$. They are defined inductively:

$$\pi(\tau) = \{\ \varepsilon\ \} \cup \pi^\star(\tau)$$

$$\begin{aligned}\pi^\star(\alpha) &= \emptyset \\ \pi^\star(\texttt{[}\eta\texttt{]}\tau') &= \{\ \texttt{[}e\texttt{<}\eta\texttt{]} \cdot p\ \mid\ p \in \pi(\tau'),\ e \in \texttt{[}\eta\texttt{]}\ \} \\ \pi^\star(\tau_1\ \texttt{*}\ \ldots\ \texttt{*}\ \tau_\texttt{m}) &= \{\ \texttt{(k/m)} \cdot p\ \mid\ p \in \pi(\tau_\texttt{k}),\ \texttt{k} \in \texttt{(m)}\ \}\end{aligned}$$

The domain has a structure that is similar to one of the type: leaves are basic types (variables and scalar types) and inner nodes correspond to compound types (array and tuples). One particularity is worth mentioning: because array indexes are handled uniformly, all the branches of array nodes point to the same sub-tree as depicted by the following example:



Definition 3.3 (Type projection)**.** Given a type τ and a projection $p \in \pi(\tau)$, the *type projection* $\tau.p$ is the type of the sub-part addressed by p:

$$\begin{aligned} \tau.\varepsilon &= \tau \\ [\eta]\tau.([e{<}\eta] \cdot p) &= \tau.p \\ \tau_1 * \ldots * \tau_m.((k/m) \cdot p) &= \tau_k.p \end{aligned}$$

Lemma 3.4 (Concatenation of projections)**.** *The above definition is well-founded, i.e., for any $p \in \pi(\tau)$, $\tau.p$ exists. Moreover, iterated projection is compatible with projection concatenation: given a type $\tau \in \mathcal{T}$ and projections $p \in \pi(\tau)$ and $q \in \pi(\tau.p)$, then,*

$$p \bullet q \in \pi(\tau) \ \wedge \ \tau.p.q = \tau.(p \bullet q)$$

Proof. Both results are established by a straightforward induction on p. □

To separate the structural part, which is manipulated with projection functors, from the typing part, which is blindly transmitted and preserved, we distinguish the projections that address the atomic components of a type, namely the parts that have variable (or scalar) type.

Definition 3.5 (Maximal projections)**.** Given a type τ, the set of *maximal projections* — $\overline{\pi(\tau)}$ — denotes the longest projections. A similar definition applies to non-empty projections, denoted $\overline{\pi^\star(\tau)}$.

$$\overline{\pi(\tau)} = \{\ p \in \pi(\tau) \ \mid \ \pi^\star(\tau.p) = \emptyset\ \}$$

$$\overline{\pi^\star(\tau)} = \{\ p \in \pi^\star(\tau) \ \mid \ \pi^\star(\tau.p) = \emptyset\ \}$$

The following table gives some examples of domains, distinguishing the maximal projections.

τ	$\overline{\pi^\star(\tau)}$	$\pi^\star(\tau) \setminus \overline{\pi^\star(\tau)}$
int	$\emptyset$	$\emptyset$
$[\nu]$int	$\{\ [e{<}\eta] \cdot \varepsilon\ \}$	$\emptyset$
bool * int	$\left\{ \begin{array}{c} (1/2) \cdot \varepsilon \\ (2/2) \cdot \varepsilon \end{array} \right\}$	$\emptyset$
bool * $[\eta]$int	$\left\{ \begin{array}{c} (1/2) \cdot \varepsilon \\ (2/2) \cdot [e{<}\eta] \cdot \varepsilon \end{array} \right\}$	$\{\ (2/2) \cdot \varepsilon\ \}$

The maximal projections allow to represent compound data as a dependent mapping from projections to values of the correct type. The description of data-structure resembles the containters of Abbott, Altenkirch, and Ghani [AAG03]. The similarities with this work remain to be explored. In the language of dependent types, a compound data d of type τ is represented by the dependent product:

$$\Pi\, p : \overline{\pi(\tau)}.\ (\tau.p)$$

3.3 A Formal Representation of Projection Functors

Projection functors describe statically known indirections. They aim at expressing various ways to access the same data by embedding the structures of a typed compound data (the *view*) into another one (the *data*), e.g., by swapping the elements of a pair.

To this end, they map the maximal projections of the view, that correspond to components with base types, to maximal projections of the data: they are interpreted as elements of $\overline{\pi(\tau_v)} \to \overline{\pi(\tau_d)}$. Two extra requirements apply: (i) projection functors preserve types, i.e., components of the view are mapped to components of the data with the same type and (ii) they map array projections in a uniform way, namely, values of indexes cannot be distinguished.

3.3.1 The Language of Projection Functors

The function syntax for projection functors is redundant: because tuple components are matched separately, the projections that are common to all of them are duplicated, both on the pattern and expression sides. For instance, the `map_swap` functor builds a view of an array of pairs where each pair is swapped:

```
let map_swap ν α β = function
  | [i<ν] :: (1/2) :: l_β → [i<ν] :: (2/2) :: l_β
  | [i<ν] :: (2/2) :: l_α → [i<ν] :: (1/2) :: l_α
```

The `[i<ν]` pattern and the `[i<ν]` projections are repeated in both cases. Such a representation is would lead to duplicated formal manipulations. A closer look at projection functors gives a hint of a denser description: the prefix of produced projections only depend on a prefix of the matched pattern.

Syntax We view projection functors as terms of a simple language that processes (finite) sequences of projections with the following operations.

- Consuming a projection from the incoming sequence. These operations correspond to the projection patterns of our function-like representation. They are called *abstractions*.
- Producing a projection into the outgoing sequence. These operations correspond to the projection expressions of our function-like representation. They are called *applications*.
- Forwarding the rest of the projections. This *identity* operation is depicted by the generic list tail pattern — l_α — in the function-like representation.

The syntax of *projection trees* (φ) is given in Figure 3.9. To be consistent with sizes and types, they are denoted by Greek letters. They are abstract objects used by the compiler to represent properties of programs. The identity — $\mathbf{1}_\tau$ — is tagged with the type of the corresponding data (τ). Since array projections are dynamic, they are handled in a uniform way, hence only matched with an *index variable* (i). Array applications use *index expressions* ($e \in$ `[`ν`]`). On the contrary, since tuple projections are static, components are exhaustively enumerated. Tuple abstractions thus introduce a branching that selects a continuation according to the component. Tuple applications are static as well. They are defined by immediate integers (`k` $\in$ `(m)`).

φ ::=		***Projection transformers***
	$\mathbf{1}_\tau$	identity
|	`@[`e`<`η`]`. φ	array application
|	`@(k/m)`. φ	tuple application
|	`λ[`i`<`η`]`. φ	array abstraction
|	`λ(1/m)`. φ	tuple abstraction
	| ...	
	`|(m/m)`. φ	

Figure 3.9: *Syntax of projection functors, tree representation*

This form exhibits a typcal binding rule. The index variable introduced by an array abstraction may be used inside the index expression of array applications that appear in the body of the abstraction. However, they cannot be used in the sizes, that play the role of types in λ-calculus. With this syntax, the above `map_swap` functor is written:

```
λ[i<ν]. @[i<ν]. λ(1/2). @(2/2). 1_β
                |(2/2). @(1/2). 1_α
```

This representation makes it obvious that the produced array projection — `[`i`<`ν`]` — does not depend on the accessed component of the pair. The following projection functor illustrates the converse: the index expressions e of array applications — `@[`e`<`ν`]` — are different in each branch, thus the application cannot be pushed out of the tuple abstraction:

```
λ[i<ν]. λ(1/2). @[     i      <ν]. @(2/2). 1_β
                |(2/2). @[ ν - i - 1  <ν]. @(1/2). 1_α
```

Regular arrays *Remark*

Projection functors only describe accesses to regular arrays because the index variables cannot appear in sizes.

Typing We eliminate the ill-formed projection functors with the help of a typing judgment — $\Gamma \,;\, \mathcal{C} \vdash \varphi : \tau_d \to \tau_v$ — that relates a subset of bounded size variables Γ, a set of polynomial inequalities $\mathcal{C}$, a projection functor φ and the types of the data τ_d and the view τ_v. The deduction rules are given in Figure 3.10, alongside the *index range* relation — $\Gamma \,;\, \mathcal{C} \vdash e \in [\eta]$ — that ensures that array applications respect bounds. Given an index expression e and a size η (both multivariate polynomials), we denote $e \in [\eta]$ the constraint $0 \leq e < \eta$.

Projection Functor Typing $\Gamma \,;\, \mathcal{C} \vdash \varphi : \tau \to \tau$

$$\text{T-Id}\ \frac{}{\Gamma \,;\, \mathcal{C} \vdash \mathbf{1}_\tau : \tau \to \tau}$$

$$\text{T-ArrAbs}\ \frac{\Gamma, i \,;\, \mathcal{C} \wedge i : [\eta] \vdash \varphi : \tau_1 \to \tau_2}{\Gamma \,;\, \mathcal{C} \vdash \lambda[i{<}\eta].\ \varphi : \tau_1 \to [\eta]\tau_2}$$

$$\text{T-TplAbs}\ \frac{\begin{array}{c}\Gamma \,;\, \mathcal{C} \vdash \varphi_1 : \tau \to \tau_1 \\ \dots \\ \Gamma \,;\, \mathcal{C} \vdash \varphi_{\mathtt{m}} : \tau \to \tau_{\mathtt{m}}\end{array}}{\begin{array}{c}\lambda(1/\mathtt{m}).\ \varphi_1 \\ \Gamma \,;\, \mathcal{C} \vdash \mid \quad \dots \quad : \tau \to \tau_1 * \dots * \tau_{\mathtt{m}} \\ \mid(\mathtt{m}/\mathtt{m}).\ \varphi_{\mathtt{m}}\end{array}}$$

Index Range $\Gamma \,;\, \mathcal{C} \vdash e \in [\eta]$

$$\text{I-Bounds}\ \frac{\forall \Gamma,\ C \implies e \in [\eta]}{\Gamma \,;\, \mathcal{C} \vdash e \in [\eta]}$$

$$\text{T-ArrApp}\ \frac{\Gamma \,;\, \mathcal{C} \vdash \varphi : \tau_1 \to \tau_2 \qquad \Gamma \,;\, \mathcal{C} \vdash e \in [\eta]}{\Gamma \,;\, \mathcal{C} \vdash @[e{<}\eta].\ \varphi : [\eta]\tau_1 \to \tau_2}$$

$$\text{T-TplApp}\ \frac{\Gamma \,;\, \mathcal{C} \vdash \varphi : \tau_i \to \tau}{\Gamma \,;\, \mathcal{C} \vdash @(\mathtt{i}/\mathtt{m}).\ \varphi : \tau_1 * \dots * \tau_{\mathtt{m}} \to \tau}$$

Figure 3.10: *Typing rules for projection functors*

The array abstraction rule (T-ArrAbs) extends the environment with the index variable i, assuming $i \notin \Gamma$, and adds the constraint $i \in [\eta]$ to $\mathcal{C}$. As mentioned above, this index cannot be captured by the size η since the resulting type $[\eta]\tau_2$ must be well-scoped in Γ (without i).

The second premise of the array application rule (T-ArrApp) can only be derived if any valuation of the bound variables Γ that fulfill the constraints $\mathcal{C}$ also verifies $e \in [\nu]$, hence if the access is *safe*.

The tuple abstraction rule (T-TplAbs) ensures that all the continuations $\varphi_1, \dots, \varphi_{\mathtt{m}}$ are consistent, i.e., that they apply to the same type of data. Tuple application (rule T-TplApp) only provides partial information on the data type, the type of non selected components is arbitrary.

Semantics Projection functors can be modeled as mappings between maximal projections. The semantics of a projection functor $\varphi : \tau_d \to \tau_v$ is a function $[\![\varphi]\!] \in \overline{\pi(\tau_v)} \to \overline{\pi(\tau_d)}$ that preserves projection types:

$$\forall p \in \overline{\pi(\tau_v)},\ \tau_v.p = \tau_d.([\![\varphi]\!](p))$$

The semantics is defined constructively by:

$$\begin{array}{rcrcl}
[\![\mathbf{1}_\alpha]\!] & = & \varepsilon & \mapsto & \varepsilon \\
[\![@[e{<}\eta].\ \varphi]\!] & = & p & \mapsto & [e{<}\eta] \cdot [\![\varphi]\!](p) \\
[\![@(\mathtt{k}/\mathtt{m}).\ \varphi]\!] & = & p & \mapsto & (\mathtt{k}/\mathtt{m}) \cdot [\![\varphi]\!](p) \\
[\![\lambda[i{<}\eta].\ \varphi]\!] & = & [e{<}\eta] \cdot p & \mapsto & [\![\varphi\,\{e/i\}]\!](p) \\
\left[\!\!\left[\begin{array}{l}\lambda(1/\mathtt{m}).\ \varphi_1 \\ \mid \quad \dots \\ \mid(\mathtt{n}/\mathtt{m}).\ \varphi_{\mathtt{n}}\end{array}\right]\!\!\right] & = & (\mathtt{k}/\mathtt{m}) \cdot p & \mapsto & [\![\varphi_{\mathtt{k}}]\!](p)
\end{array}$$

Theorem 3.6 (Soundness)**.** *Let φ be a projection functor and τ_d, τ_v be types. If $\Gamma \,;\, \mathcal{C} \vdash \varphi : \tau_d \to \tau_v$, then $[\![\varphi]\!]$ exists.*

Proof. By induction on the structure of the projection functor. □

3.3.2 Primitives

The correctness of projection functors cannot be established in general as it requires to check polynomial inequalities. Hence, projection functors are only used as an internal description of

some built-in operators. The underlying projection functors are presented below, in both function-like and tree syntax alongside their types. They are named from a data point of view, i.e., their names reflect how the resulting view is built from the source.

Invertible array views Let us start with the simplest array transformations. As their types suggest, all of them are bijective. The inverse functor of `single` is `extract`, while `reverse` and `transpose` are involutions.

`let single` α `= function` `| [`i`<1] ::` l_α $\to$ l_α	$\alpha \to [1]\alpha$ $\lambda[i<1].\ \mathbf{1}_\alpha$

`let extract` α `= function` `|` l_α $\to$ `[0<1] ::` l_α	$[1]\alpha \to \alpha$ $@[0<1].\ \mathbf{1}_\alpha$

`let reverse` ν α `= function` `| [`$i<\nu$`] ::` l_α $\to$ `[`$\nu-1-i<\nu$`] ::` l_α	$[\nu]\alpha \to [\nu]\alpha$ $\lambda[i<\nu].\ @[\nu-1-i<\nu].\ \mathbf{1}_\alpha$

`let transpose` ν μ α `= function` `| [`$j<\mu$`] :: [`$i<\nu$`] ::` l_α $\to$ `[`$i<\nu$`] :: [`$j<\mu$`] ::` l_α	$[\nu]\,[\mu]\alpha \to [\mu]\,[\nu]\alpha$ $\lambda[j<\mu].\ \lambda[i<\nu].\ @[i<\nu].\ @[j<\mu].\ \mathbf{1}_\alpha$

The `single` projection functor `drops` an index of size 1, thus the resulting view has one dimension more that the object it applies to. Conversely, the `extract` one *inserts* a projection on an array of size 1. The `reverse` functor highlights that index expressions may capture sizes.

Complex array views As sketched in Chapter 2, the above operators are insufficient to express algorithms such as convolutions. They are supplemented with a set of functors that introduce non trivial computations on sizes. We recall their graphical representation in Figure 3.11.

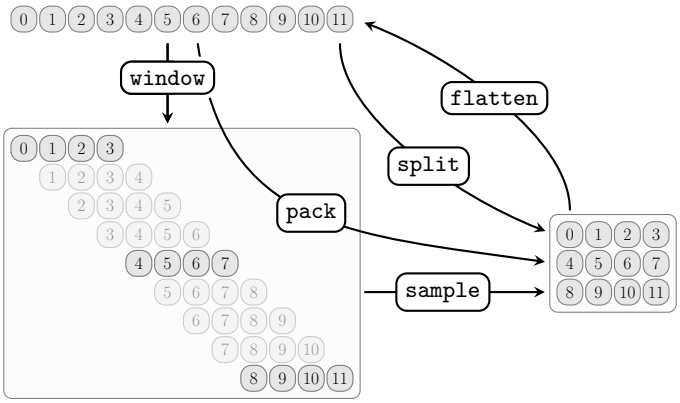


Figure 3.11: *Some primitive array transformations*

We refer the reader to Section 2.4.3 for an informal introduction to these operators.[1] These functions introduce index computations with the corresponding size relations: index addition for `window`, index multiplication by a size for `sample`. The underlying projection functors are:

`let window` ν κ α `= function` `| [`$i<\nu$`] :: [`$j<\kappa$`] ::` l_α $\to$ `[`$i+j<\nu+\kappa-1$`] ::` l_α	$[\nu+\kappa-1]\alpha \to [\nu]\,[\kappa]\alpha$ $\lambda[i<\nu].\ \lambda[j<\kappa].\ @[i+j<\nu+\kappa-1].\ \mathbf{1}_\alpha$

`let sample` ν κ α `= function` `| [`$i<\nu$`] ::` l_α $\to$ `[`$i*\kappa<\nu*\kappa-\kappa+1$`] ::` l_α	$[\nu*\kappa-\kappa+1]\alpha \to [\nu]\alpha$ $\lambda[i<\nu].\ @[i*\kappa<\nu*\kappa-\kappa+1].\ \mathbf{1}_\alpha$

`let split` ν κ α `= function` `| [`$i<\nu$`] :: [`$j<\kappa$`] ::` l_α $\to$ `[`$i*\kappa+j<\nu*\kappa$`] ::` l_α	$[\nu*\kappa]\alpha \to [\nu]\,[\kappa]\alpha$ $\lambda[i<\nu].\ \lambda[j<\kappa].\ @[i*\kappa+j<\nu*\kappa].\ \mathbf{1}_\alpha$

The inverse of `split`, named `flatten`, cannot be represented as a projection functor because it would involve non-polynomial operations. This results in a constraint on possible memory locations: only the matrix can be defined as a view on the vector. Chapter 6 discusses the mechanisms and limits of memory location inference.

[1] In SCADE, `window` designates a temporal sliding buffer (that contains the k last values of a stream). But it is not as different to our `window` operator: their semantics are identical once arrays are viewed as streams.

`let repeat` $\nu\ \alpha$ `= function` `  | [i<`ν`] ::` $l_\alpha \rightarrow l_\alpha$	$\alpha \rightarrow [\nu]\alpha$ λ`[i<`ν`]. 1`$_\alpha$

The `repeat` projection functor defines an array broadcast: all the elements refer to the same value. This allows to build a constant array without copy or memory cost. The single operator is a particular case with a much stronger property: it is invertible, hence bijective.

The general sampling operator: `pack` *Remark*

Among the above projection functors, `split` is a special case of `pack`, that is obtained by composing `window` and `sample`.
These redundant built-ins are preferred because they convey additional properties that are harder to retrieve from `pack` size parameters. Namely that `window` is surjective, `sample` is injective and `split` is bijective.

Array destruction A crucial declarative operation is missing: concatenation. Like `flatten`, it cannot be expressed as a projection functor, but for a different reason. To access the result of a concatenation of a pair of arrays, the right component of the pair has to be selected depending on the index value, hence introducing non-regular index projections. As above, concatenation is internally represented with its inverse operation, called `cut`, that splits an array into its left and right parts:

`let cut` $\nu\ \mu\ \alpha$ `= function` `  | (1/2) ::` $[i<\nu]$ `::` $l_\alpha \rightarrow$ $[i+0<\nu+\mu]$ `::` l_α `  | (2/2) ::` $[i<\mu]$ `::` $l_\alpha \rightarrow$ $[i+\nu<\nu+\mu]$ `::` l_α	$[\nu]\alpha * [\mu]\alpha \rightarrow [\nu+\mu]\alpha$ λ`(1/2).` λ`[i<`ν`].@[i+0<`$\nu+\mu$`]. 1`$_\alpha$ `|(2/2).` λ`[i<`μ`].@[i+`ν`<`$\nu+\mu$`]. 1`$_\alpha$

Immediate arrays are represented by a family of projection functors $\texttt{array}_\texttt{n}$ that allow to view arrays as tuples. The size `n` is a constant. The $\texttt{array}_\texttt{n}$ functor maps the projection over a `n`-ary tuple to projections over an array of size `n`. From the data point of view, it allows to view an array as a tuple. These projection functors are used to represent the immediate array construct, provided by the syntax `[| ... |]`.

`let` $\texttt{array}_\texttt{n}$ α `= function`
`  | (1/n) ::` $l_\alpha \rightarrow$ `[1-1<n] ::` l_α
`  |` `...`
`  | (n/n) ::` $l_\alpha \rightarrow$ `[n-1<n] ::` l_α
	$[\nu]\alpha \rightarrow \alpha * \ldots * \alpha$
	λ`(1/n).@[1-1<n]. 1`$_\alpha$
	`|` `...`
	`|(n/n).@[n-1<n]. 1`$_\alpha$

Views on pairs To illustrate projection functors acting on tuples, we provide two primitive operations on pairs: swapping component and rotating nested pairs.

`let swap` $\alpha\ \beta$ `= function` `  | (1/2) ::` $l_\beta \rightarrow$ `(2/2) ::` l_β `  | (2/2) ::` $l_\alpha \rightarrow$ `(1/2) ::` l_α	$\alpha * \beta \rightarrow \beta * \alpha$ λ`(1/2).@(2/2). 1`$_\beta$ `|(2/2).@(1/2). 1`$_\alpha$

`let rotate_left` $\alpha\ \beta\ \gamma$ `= function` `  | (1/2) :: (1/2) ::` $l_\alpha \rightarrow$ `(1/2) ::` l_α `  | (1/2) :: (2/2) ::` $l_\beta \rightarrow$ `(2/2) :: (1/2) ::` l_β `  | (2/2) ::` $l_\gamma \rightarrow$ `(2/2) :: (2/2) ::` l_γ	$\alpha * (\beta * \gamma) \rightarrow (\alpha * \beta) * \gamma$ λ`(1/2).` λ`(1/2).@(1/2). 1`$_\alpha$ `|(2/2).@(2/2).@(1/2). 1`$_\beta$ `|(2/2).@(2/2).@(2/2). 1`$_\gamma$

3.3.3 Normalization

A projection functors may be described by multiple trees because independent applications and projections commute. For instance, the `cst_fst` functor that maps all the elements of an array to the first component of a pair can be described by two projection functors:

`let cst_fst` $\nu\ \alpha$ `= function` `  | [i<`ν`] ::` $l_\alpha \rightarrow$ `(1/2) ::` l_α	$\varphi_1 = \lambda$`[i<`ν`].@(1/2). 1`$_\alpha$ $\varphi_2 =$`@(1/2).` λ`[i<`ν`]. 1`$_\alpha$

To compare such projection functors, we define a normal form. It is obtained by pushing applications outward as much as possible, e.g., the φ_2 version above. This form carries stronger structural information An application appears under an abstraction only if the former depends on

the latter. From a projection stream point of view, the associated process is the one that computes the output stream as soon as possible. Formally, pushing applications outward is obtained by applying the following *commutation* rewriting rules at any place in the tree.

$$\lambda[i{<}\eta].\,@[e{<}\eta'].\,\varphi \implies @[e{<}\eta'].\,\lambda[i{<}\eta].\,\varphi \quad \text{if } i \notin \mathcal{FV}(\,e\,) \qquad \lambda[i{<}\eta].\,@(\mathtt{k}/\mathtt{m}').\,\varphi \implies @(\mathtt{k}/\mathtt{m}').\,\lambda[i{<}\eta].\,\varphi$$

$$\begin{array}{l}\lambda(\mathtt{1}/\mathtt{m}).\,@[e{<}\eta'].\,\varphi_1\\ \mid \quad \ldots\\ \mid(\mathtt{m}/\mathtt{m}).\,@[e{<}\eta'].\,\varphi_{\mathtt{m}}\end{array} \implies \begin{array}{l}@[e{<}\eta'].\,\lambda(\mathtt{1}/\mathtt{m}).\,\varphi_1\\ \mid \quad \ldots\\ \mid(\mathtt{m}/\mathtt{m}).\,\varphi_{\mathtt{m}}\end{array} \qquad \begin{array}{l}\lambda(\mathtt{1}/\mathtt{m}).\,@(\mathtt{k}/\mathtt{m}').\,\varphi_1\\ \mid \quad \ldots\\ \mid(\mathtt{m}/\mathtt{m}).\,@(\mathtt{k}/\mathtt{m}').\,\varphi_{\mathtt{m}}\end{array} \implies \begin{array}{l}@(\mathtt{k}/\mathtt{m}').\,\lambda(\mathtt{1}/\mathtt{m}).\,\varphi_1\\ \mid \quad \ldots\\ \mid(\mathtt{m}/\mathtt{m}).\,\varphi_{\mathtt{m}}\end{array}$$

Such commutation is only possible if the application does not depend on the abstraction. For the tuple abstractions, this amounts to checking that the applications are identical in each branch. For array abstractions, the index variable must not appear in the index expression.

Some redundancies remain. The identity projection $\mathbf{1}_{[\nu]\tau}$ is equivalent to $\lambda[i{<}\nu].\,@[i{<}\nu].\,\mathbf{1}_\tau$. A similar equivalence applies to tuples. These overdetailed identities are simplified only when they occur in tail position by the two following *simplification* rewriting rules, which complete our normalization scheme:

$$\lambda[i{<}\eta].\,@[i{<}\eta].\,\mathbf{1}_\tau \implies \mathbf{1}_{[\eta]\tau} \qquad \begin{array}{l}\lambda(\mathtt{1}/\mathtt{m}).\,@(\mathtt{1}/\mathtt{m}).\,\mathbf{1}_{\tau_1}\\ \mid \quad \ldots\\ \mid(\mathtt{m}/\mathtt{m}).\,@(\mathtt{m}/\mathtt{m}).\,\mathbf{1}_{\tau_{\mathtt{m}}}\end{array} \implies \mathbf{1}_{\tau_1 * \ldots * \tau_{\mathtt{m}}}$$

Theorem 3.7 (Termination and confluence of normalization)**.** *The above rewrite system, made of commutation and simplification rules, terminates. It is confluent.*

Proof. For abstract rewrite systems, Newman's lemma states the equivalence between confluence and termination with local confluence.

Termination. We define an *abstraction count* metric $\mathcal{N}^\lambda(c,\ \varphi)$ that sums the number of abstractions that surround applications starting at an integer level c, and simply write $\mathcal{N}^\lambda(\varphi)$ the *outer* abstraction count $\mathcal{N}^\lambda(0,\ \varphi)$:

$$\begin{aligned}\mathcal{N}^\lambda(c,\ \mathbf{1}_\tau) &= 0\\ \mathcal{N}^\lambda(c,\ @[i{<}\eta].\,\varphi) &= \mathcal{N}^\lambda(c,\ \varphi) + c\\ \mathcal{N}^\lambda(c,\ @(\mathtt{k}/\mathtt{m}).\,\varphi) &= \mathcal{N}^\lambda(c,\ \varphi) + c\end{aligned} \qquad \begin{aligned}\mathcal{N}^\lambda(c,\ \lambda[i{<}\eta].\,\varphi) &= \mathcal{N}^\lambda(c+1,\ \varphi)\\ \mathcal{N}^\lambda\left(c,\ \begin{array}{l}\lambda(\mathtt{1}/\mathtt{m}).\,\varphi_1\\ \mid \quad \ldots\\ \mid(\mathtt{m}/\mathtt{m}).\,\varphi_{\mathtt{m}}\end{array}\right) &= \sum_{k=1}^{\mathtt{m}} \mathcal{N}^\lambda(c+1,\ \varphi_k)\end{aligned}$$

It is trivial to check that, for any c, φ, φ' such that $\varphi \implies \varphi'$, the abstraction count metric strictly decreases: $\mathcal{N}^\lambda(c,\ \varphi') < \mathcal{N}^\lambda(c,\ \varphi)$. Indeed, the commutation rules push one application outside of an abstraction, and the tuple abstraction rules decrease the number of applications since tuples are not empty. Both simplification rules eliminate an application that is behind an abstraction (hence with a non-zero count), thus decreasing the metric.
Because this metric over natural number is strictly decreasing, there are no infinite rewrite chains and the rewrite system terminates.

Local confluence. Given $\varphi, \varphi_1, \varphi_2$ projection functors such that $\varphi_1 \impliedby \varphi \implies \varphi_2$, there is a φ' such that $\varphi_1 \implies \varphi' \impliedby \varphi_2$.
This is because all rules apply to an abstraction followed by an application, and no pairs of distinct rules are applicable simultaneously, the two reductions are either identical or modify unrelated parts of the projection functor. Thus, they can be applied in any order, giving equal terms. Hence the local convergence. □

Theorem 3.8 (Preservation of type)**.** *Given projection functors φ and φ' such that $\varphi \implies^* \varphi'$, and a typing environment with constraints $\Gamma\,;\,\mathcal{C}$, then:*

$$\Gamma\ ;\ \mathcal{C} \vdash \varphi :\ \tau_d \to \tau_v \implies \Gamma\ ;\ \mathcal{C} \vdash \varphi' :\ \tau_d \to \tau_v$$

Proof. The typing relation is preserved by all the rewriting rules, hence also by normalization. □

Comparison *Implementation note (1)*

Comparing projection functors is performed on normalized forms. It checks that abstractions and applications are identical, up to renaming of index variables. For instance, the following functors are equivalent:

$$\begin{cases} \lambda[i{<}\eta].\ @(1/2).\ @[\eta - i - 1{<}\eta].\ \mathbf{1}_\tau \\ @(1/2).\ \lambda[j{<}\eta].\ @[\eta - j - 1{<}\eta].\ \mathbf{1}_\tau \end{cases}$$

Uniqueness of representation Different normalized projection functors may be equal. Indeed, two equally structured projection functors have the same semantics if their array applications coincide *on their domain*, i.e., their differences vanish on all the domains of index variables. Hence, the following functors are equivalent because $i - i^2$ vanishes on $[2] = \{0, 1\}$:

$$\begin{cases} \lambda[i{<}2].\ @[i^2{<}2].\ \mathbf{1}_\alpha \\ \lambda[i{<}2].\ @[i{<}2].\ \mathbf{1}_\alpha \end{cases}$$

Using the simplification rewriting rules, these functors are equivalent to $\mathbf{1}_{[2]\alpha}$. A complete check of equivalence is hopeless in the presence of polynomial constraints $\mathcal{C}$. However, the special case of unary arrays is worth implementing, since they are introduced and destructed by the `single` and `extract` primitives. Because $[1]$ is the singleton $\{0\}$, unary array abstractions and applications commute. The normalized form of $\lambda[i{<}1].\ @[0{<}1].\ \mathbf{1}_\alpha$ is $@[0{<}1].\ \lambda[i{<}1].\ \mathbf{1}_\alpha$ since 0 does not depend on i. This should be simplified further into the $\mathbf{1}_{[1]\alpha}$ projection functors. A simple rewritting rule is insufficient. It would fail to simplify $@[0{<}1].\ \lambda[i{<}\eta].\ \lambda[j{<}1].\ \mathbf{1}_\alpha$ into $\lambda[i{<}\eta].\ \mathbf{1}_\alpha$. In our prototype, it is implemented by a dedicated analysis.

Access regularity *Remark*

We argue that there is little possibility for different representations of actually equal projection functors because of the regularity of access in *dense* array computations. Indeed all the provided array operations feature accesses that are linear into the indexes. This lowers the risk of equivalent array applications, since non-zero polynomials that vanish on n values have at least degree n.

3.4 Operations

Projection functors aim at representing how memory locations are accessed at compile-time before specialization, hence they must be generic in both sizes and types. Some fundamental declarative properties such as uniqueness and completeness of definitions stem from the ones of projection functors (i.e., injectivity, surjectivity). Rather than trying to establish them with pessimistic analyses, these properties are flagged for the built-ins functors, and they are propagated for user-defined ones. This approach relies on the ability to compose projection functors.

3.4.1 Composition

Beside the aforementioned functional properties, composition preserves the correctness of array applications. This operation amounts to a simple β-reduction. So that no unintentional captures occur, we assume that array abstractions introduce distinct variables. In the implementation, a renaming is embedded in the composition procedure.

Global composition Following our data-centeric point of view, the composition of projection functors φ and ψ, denoted $\varphi \gg \psi$, is the projection functor that builds a view through φ and then ψ. From the projection-list point of view, the order is reversed: ψ is applied first, then φ. Figure 3.12 illustrates the composition order with the `pack` functor. Syntactically, projection functor composition is defined as follows:

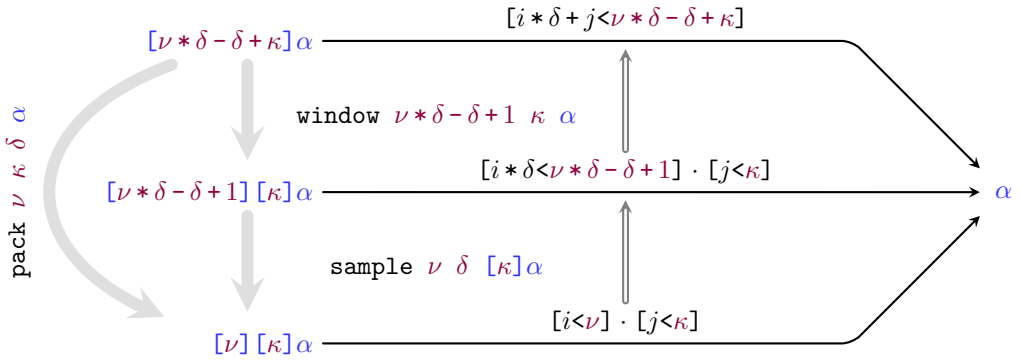


Figure 3.12: *Functor composition:* `pack` $\nu\ \kappa\ \delta\ \alpha$ = `window` $(\nu * \delta - \delta + 1)\ \kappa\ \alpha \gg$ `sample` $\nu\ \delta\ [\kappa]\alpha$

$$
\begin{array}{l|l}
\varphi \gg \mathbf{1}_\tau := \varphi & \big(@[e{<}\eta].\ \varphi\big) \gg \psi := @[e{<}\eta].\ \big(\varphi \gg \psi\big) \\
\mathbf{1}_\tau \gg \psi := \psi & \big(@(\mathtt{k}/\mathtt{m}).\ \varphi\big) \gg \psi := @(\mathtt{k}/\mathtt{m}).\ \big(\varphi \gg \psi\big) \\
\hline
\big(\lambda[i{<}\eta].\ \varphi\big) \gg \big(@[e{<}\eta].\ \psi\big) := \varphi\{e/i\} \gg \psi & \varphi \gg \big(\lambda[i{<}\eta].\ \psi\big) := \lambda[i{<}\eta].\ \big(\varphi \gg \psi\big) \\
\begin{pmatrix} \lambda(\mathtt{1}/\mathtt{m}).\ \varphi_1 \\ \mid \quad \dots \\ \mid (\mathtt{m}/\mathtt{m}).\ \varphi_m \end{pmatrix} \gg \big(@(\mathtt{k}/\mathtt{m}).\ \psi\big) := \varphi_k \gg \psi & \varphi \gg \begin{pmatrix} \lambda(\mathtt{1}/\mathtt{m}).\ \psi_1 \\ \mid \quad \dots \\ \mid (\mathtt{m}/\mathtt{m}).\ \psi_m \end{pmatrix} := \begin{array}{l} \lambda(\mathtt{1}/\mathtt{m}).\ \big(\varphi \gg \psi_1\big) \\ \mid \qquad \dots \\ \mid (\mathtt{m}/\mathtt{m}).\ \big(\varphi \gg \psi_m\big) \end{array}
\end{array}
$$

The bottom-left corner features β-reduction. If structures match, the index variable is substituted in the array case, or the appropriate functor is selected for tuples. The right-hand-side rules transmit the applications of the first argument as well as the abstractions of the second one. They are not exclusive. The composition $\big(@(\mathtt{k}/\mathtt{n}).\ \varphi\big) \gg \big(\lambda[i{<}\nu].\ \psi\big)$ can be reduced in to either of following two functors:

$$
\begin{array}{l}
@(\mathtt{k}/\mathtt{n}).\ \lambda[i{<}\nu].\ \big(\varphi \gg \psi\big) \\
\lambda[i{<}\nu].\ @(\mathtt{k}/\mathtt{n}).\ \big(\varphi \gg \psi\big)
\end{array}
$$

These functors are equivalent and their normalized forms coincide. The implemented composition favors abstractions, as it produces a form that is closer to the normalized one. However, normalization is still needed since the substitutions may break dependencies and identities may appear. For instance, composing transposition effectively simplifies to an identity:

$$
\begin{aligned}
&\texttt{transpose}\ \mu\ \nu\ \alpha \gg \texttt{transpose}\ \nu\ \mu\ \alpha \\
&= \big(\lambda[i{<}\nu].\ \lambda[j{<}\mu].\ @[j{<}\mu].\ @[i{<}\nu].\ \mathbf{1}_\alpha\big) \gg \big(\lambda[i{<}\mu].\ \lambda[j{<}\nu].\ @[j{<}\nu].\ @[i{<}\mu].\ \mathbf{1}_\alpha\big) \\
&= \lambda[i{<}\nu].\ \lambda[j{<}\mu].\ @[i{<}\nu].\ @[j{<}\mu].\ \mathbf{1}_\alpha \\
&= \lambda[i{<}\nu].\ @[i{<}\nu].\ \lambda[j{<}\mu].\ @[j{<}\mu].\ \mathbf{1}_\alpha && \text{(commutation)} \\
&= \lambda[i{<}\nu].\ @[i{<}\nu].\ \mathbf{1}_{[\mu]\alpha} && \text{(simplification)} \\
&= \mathbf{1}_{[\nu][\mu]\alpha} && \text{(simplification)}
\end{aligned}
$$

Lemma 3.9 (Composition type)**.** *Given projection functors φ, ψ, types τ_1, τ_2, τ_3, an environment Γ and polynomial inequalities $\mathcal{C}$ such that:*

$$
\begin{cases}
\Gamma\ ;\ \mathcal{C} \vdash \varphi :\ \tau_1 \to \tau_2 \\
\Gamma\ ;\ \mathcal{C} \vdash \psi :\ \tau_2 \to \tau_3
\end{cases}
$$

Then $\varphi \gg \psi$ exists and $\Gamma\ ;\ \mathcal{C} \vdash \varphi \gg \psi :\ \tau_1 \to \tau_3$

Proof. By induction on the structure of φ and ψ.

- If $\varphi = \mathbf{1}_\tau$ or $\psi = \mathbf{1}_\tau$, then $\tau = \tau_2$ and the result trivially holds.

- If φ is an application (resp. ψ is an abstraction), it is pushed outward by composition, by the top-(resp. bottom-)right rules, allowing for induction hypothesis use.
- If φ is an abstraction and ψ is an application, because of the common type τ_2 they have the same size or arity, hence the β-reduction is possible. □

Lemma 3.10 (Associativity of composition)**.** *Composition is associative:*
Given three projection functors φ, ψ and χ,

$$(\varphi \gg \psi) \gg \chi = \varphi \gg (\psi \gg \chi)$$

Proof. By induction on the structure of the projection functors:

- If one of the functors is an identity, the result holds immediately.
- If φ is an application, i.e., $\varphi = @[e{<}\eta].\ \varphi'$

$$\begin{array}{l} \big((@[e{<}\eta].\ \varphi') \gg \psi\big) \gg \chi \\ \big(@[e{<}\eta].\ (\varphi' \gg \psi)\big) \gg \chi \\ @[e{<}\eta].\ \big((\varphi' \gg \psi) \gg \chi\big) \end{array} \quad \Bigg| \quad \begin{array}{c} (@[e{<}\eta].\ \varphi') \gg (\psi \gg \chi) \\ @[e{<}\eta].\ (\varphi' \gg (\psi \gg \chi)\big) \end{array}$$

  Similarly, if χ is an abstraction, i.e., $\chi = \lambda[e{<}\eta].\ \chi'$:

$$\begin{array}{l} (\varphi \gg \psi) \gg (\lambda[e{<}\eta].\ \chi') \\ \lambda[e{<}\eta].\ \big((\varphi \gg \psi) \gg \chi'\big) \end{array} \quad \Bigg| \quad \begin{array}{c} \varphi \gg \big(\psi \gg (\lambda[e{<}\eta].\ \chi')\big) \\ \varphi \gg \big(\lambda[e{<}\eta].\ (\psi \gg \chi')\big) \\ \lambda[e{<}\eta].\ (\varphi \gg (\psi \gg \chi')\big) \end{array}$$

- It remains to check that composition commutes with β-reduction. We detail how the induction hypothesis might be applied for array β-reductions below. Similar rewriting applies to tuples.

$$\begin{array}{c} \big((\lambda[i{<}\eta].\ \varphi') \gg (@[e{<}\eta].\ \psi')\big) \gg \chi \\ (\varphi'\,\{e/i\} \gg \psi') \gg \chi \end{array} \quad \Bigg| \quad \begin{array}{c} (\lambda[i{<}\eta].\ \varphi') \gg \big((@[e{<}\eta].\ \psi') \gg \chi\big) \\ (\lambda[i{<}\eta].\ \varphi') \gg (\lambda[e{<}\eta].\ (\psi' \gg \chi)\big) \\ \varphi'\,\{e/i\} \gg (\psi' \gg \chi) \end{array}$$

  The second case is slightly more subtle: it relies on the fact φ cannot capture the index i, hence $(\varphi \gg \psi')\,\{e/i\} = \varphi \gg \psi'\,\{e/i\}$:

$$\begin{array}{l} \big(\varphi \gg (\lambda[i{<}\eta].\ \psi')\big) \gg (@[e{<}\eta].\ \chi') \\ (\lambda[i{<}\eta].\ (\varphi \gg \psi')\big) \gg (@[e{<}\eta].\ \chi') \\ (\varphi \gg \psi')\,\{e/i\} \gg \chi' \\ (\varphi \gg \psi'\,\{e/i\}) \gg \chi' \end{array} \quad \Bigg| \quad \begin{array}{c} \\ \\ \varphi' \gg \big((@[e{<}\eta].\ \psi') \gg (\lambda[i{<}\eta].\ \chi')\big) \\ \varphi \gg (\psi'\,\{e/i\} \gg \chi') \end{array}$$

□

Local composition The zipping example introduced in Section 3.1.2 requires to drop one dimension in both components of the pair of arrays. Instead of building a custom projection functor that modifies both components, the `extract` primitive could be applied to each of the subparts. This is achieved with *local composition* — $\varphi \underset{p}{\gg} \psi$ — where p is a list of *generic projections*:

- `[_<`η`]` — For arrays, local composition amounts to transforming each element. This is emphasised by the placeholder used in array projection
- `(k/m)` — For tuples, local composition only transforms the selected component.

Local composition specifies the part of the view through φ on which ψ should apply. Syntactically, it is defined as follow, falling back on global composition on empty projection lists:

$$\varphi \underset{\varepsilon}{\gg} \psi \quad := \quad \varphi \gg \psi$$

$$(\lambda[i{<}\eta].\ \varphi) \underset{[\_{<}\eta]\cdot p}{\gg} \psi \quad := \quad \lambda[i{<}\eta].\ \left(\varphi \underset{p}{\gg} \psi\right)$$

$$\begin{pmatrix} \lambda(1/m).\ \varphi_1 \\ \mid \quad \dots \\ \mid (k/m).\ \varphi_k \\ \mid \quad \dots \\ \mid (m/m).\ \varphi_m \end{pmatrix} \underset{(k/m)\cdot p}{\gg} \psi \quad := \quad \begin{array}{l} \lambda(1/m).\ \varphi_1 \\ \mid \qquad \dots \\ \mid (k/m).\ \left(\varphi_k \underset{p}{\gg} \psi\right) \\ \mid \qquad \dots \\ \mid (m/m).\ \varphi_m \end{array}$$

Local composition is equivalently defined with *projection functor extensions*, denoted $\varphi \Uparrow p$, that insert abstractions and applications to make φ operate on the subpart that p refers to:

$$\varphi \underset{p}{\gg} \psi \;:=\; \varphi \gg (\psi \Uparrow p)$$

$$\begin{aligned}
\varphi \Uparrow \varepsilon &:= \varphi \\
\varphi \Uparrow \texttt{[_<}\eta\texttt{]} \cdot p &:= \lambda\texttt{[}i\texttt{<}\eta\texttt{]}.\, @\texttt{[}i\texttt{<}\eta\texttt{]}.\, \varphi \Uparrow p \\
\varphi \Uparrow \texttt{(k/m)} \cdot p &:= \begin{array}{l} \lambda\texttt{(1/m)}.\, @\texttt{(1/m)}.\, \mathbf{1}_{\alpha_1} \\ \texttt{|} \qquad\quad \ldots \\ \texttt{|(k/m)}.\, @\texttt{(k/m)}.\, \varphi \Uparrow p \\ \texttt{|} \qquad\quad \ldots \\ \texttt{|(m/m)}.\, @\texttt{(m/m)}.\, \mathbf{1}_{\alpha_m} \end{array}
\end{aligned}$$

To illustrate local composition, let us build a projection functor that describes a list-like `cons` operation, i.e., that appends an element in front of an array, as depicted in Figure 3.13.

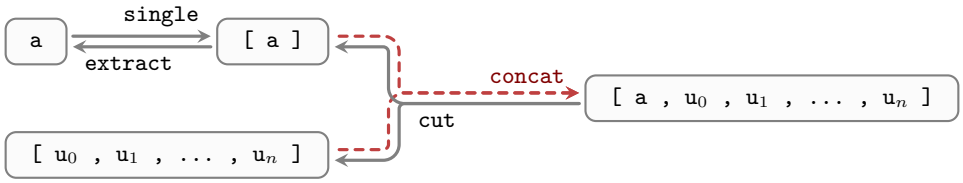


Figure 3.13: *Decomposition of a list-like (un)cons operation*

Only the reverse functor `uncons` can be directly represented with a projection functor, by applying `extract` to the first component of the view built by `cut`:

$$\begin{aligned}
\texttt{uncons}\ \nu\ \alpha &= \texttt{cut}\ 1\ \nu\ \alpha \underset{\texttt{(1/2)}}{\gg} \texttt{extract}\ \alpha \\
&= \begin{array}{l} \lambda\texttt{(1/2)}.\, @\texttt{[0<}\nu\texttt{+1]}.\, \mathbf{1}_{\alpha} \\ \texttt{|(2/2)}.\, \lambda\texttt{[}i\texttt{<}\nu\texttt{]}.\, @\texttt{[}i\texttt{+1<}\nu\texttt{+1]}.\, \mathbf{1}_{\alpha} \end{array}
\end{aligned}$$

Local composition *Implementation note (2)*

The extension-based definition of local composition requires to guess correct types for the unmodified components of tuples ($\mathbf{1}_{\alpha_1}$, …, $\mathbf{1}_{\alpha_m}$). These types are determined by the projection functor to compose with. For that reason, our implementation does not provide a functor extension operation but it computes local composition with the preceding definition.

3.4.2 Properties of Projection Functors

Beside the correctness of index applications, additional properties are needed for describing memory locations and scheduling.

Classes of projection functors Views describe how to access data for both reading and writing. Because projection functors map projections of the view type to projections of the underlying object type, reading data through a view is unconditionally correct. On the contrary, writing through views requires care, so as to guarantee the fundamental declarative properties of

- *Uniqueness* — In order to be unambiguously defined, at most one value may be written for each component. This amounts to imposing that projection functors be *injective*.
- *Completeness* — In order to be fully defined, all the parts of data must be given a value. This amounts to imposing that projection functors be *surjective*.

Bijective projection functors, that fulfill both conditions, will play a special role in the language. For some of them, such as the `cut` and `split` ones, their inverse cannot be expressed as projection functors.

Definition 3.11 (Injective, surjective)**.** In a context Γ of size variables and under polynomial constraints $\mathcal{C}$, a projection functor φ is *injective* (resp. *surjective*) if its semantics — $[\![\varphi]\!]$ — is *injective* (resp. *surjective*) for any valuation of Γ that satisfies $\mathcal{C}$.

Polynomial constraints *Example*

Recall the definition of the `sample` operator of type $[\nu * \kappa - \kappa + 1]\,\alpha \to [\nu]\,\alpha$:

$$\texttt{sample}\ \nu\ \kappa\ \alpha\ =\ \lambda[i{<}\nu].\,@[i * \kappa{<}\nu * \kappa - \kappa + 1].\,\mathbf{1}_\alpha$$

This definition is valid if $\kappa \geq 0$. However, a slightly stronger requirement — $\kappa > 0$ — allows the `sample` functor to be injective.

Representation of classes *Implementation note (3)*

Deciding injectivity and surjectivity is hopeless in general. However, projection functors are not written directly in our language. they serve rather as an internal representation for a set of base operators, whose correctness and functional properties are assumed. Functors are flagged with injectivity and surjectivity markers that are propagated by composition. A similar mechanism is used to keep track of inverse of projection functors.

Set relations A correct scheduling of operator contents must be compatible with data-flow dependencies and memory allocation, i.e., it must avoid *interferences*. Two values, or more precisely, two *writes*, interfere if they modify *intersecting* memory locations. In such cases, all the reads to one of the value must be scheduled before the write of the other. To build a conservative approximation of intersection, we define the *inclusion* and *disjunction* relations.

Definition 3.12 (Inclusion, disjunction)**.** Given a context Γ of size variables, polynomial constraints $\mathcal{C}$ and projection functors φ and ψ on the same type, i.e., such that

$$\exists \tau\ \tau_1\ \tau_2,\ \begin{cases} \Gamma\ ;\ \mathcal{C} \vdash \varphi :\ \tau \to \tau_1 \\ \Gamma\ ;\ \mathcal{C} \vdash \psi :\ \tau \to \tau_2 \end{cases}$$

projection functors are *disjoint* — $\varphi\ \#\ \psi$ — or included — $\varphi\ \subset\ \psi$ — if similar relations apply to their images, that is:

$$\begin{aligned} \varphi\ \#\ \psi &\iff \left(\forall \Gamma,\ \Gamma \vdash \mathcal{C} \implies Im\left([\![\varphi]\!]\right) \cap Im\left([\![\psi]\!]\right) = \emptyset\right) \\ \varphi\ \subset\ \psi &\iff \left(\forall \Gamma,\ \Gamma \vdash \mathcal{C} \implies Im\left([\![\varphi]\!]\right) \subset Im\left([\![\psi]\!]\right)\right) \end{aligned}$$

The easiest cases of disjoint functors are projections to different components of tuples, e.g., for a pair of type $\alpha * \beta$, $\left(@\texttt{(1/2)}.\ \mathbf{1}_\alpha\right)\ \#\ \left(@\texttt{(2/2)}.\ \mathbf{1}_\beta\right)$. In the case of array projections, disjunction or inclusion properties arise from the range of index expression. For instance, the projections to the left and right parts of a concatenation of type $[\nu + \mu]\,\alpha$ are disjoint:

$$\left(\lambda[i{<}\nu].\,@[i + 0{<}\nu + \mu].\,\mathbf{1}_\alpha\right)\ \#\ \left(\lambda[i{<}\mu].\,@[i + \nu{<}\nu + \mu].\,\mathbf{1}_\alpha\right)$$

Composition-based approximation Inclusion and disjunction are easily computed for tuples, because their projections are static. On the contrary, the dynamic (polynomial) array projections give few chances of checking these relations from definitions, apart from trivial (constant) cases.

In the compiler, projection functors are represented as lists of composed functors. For instance, the `pack` $\nu\ \kappa\ \delta\ \alpha$ functor presented in Figure 3.12 is internally stored as the composition `window` $\nu * \delta - \delta + 1\ \kappa\ \alpha \gg$ `sample` $\nu\ \delta\ [\kappa]\,\alpha$. The inclusion and disjunction properties are conservatively approximated using the following rules:

Inclusion $\varphi\ \subset\ \psi$

$$\text{I-LeftComp}\ \frac{\varphi\ \subset\ \psi}{(\varphi \gg \varphi')\ \subset\ \psi} \qquad \text{I-RightComp}\ \frac{\varphi\ \subset\ \psi \qquad \psi' \text{ surjective}}{\varphi\ \subset\ (\psi \gg \psi')}$$

Disjunction $\varphi\ \#\ \psi$

$$\text{D-LeftComp}\ \frac{\varphi \text{ injective} \qquad \psi_1\ \#\ \psi_2}{(\varphi \gg \psi_1)\ \#\ (\varphi \gg \psi_2)} \qquad \text{D-RightComp}\ \frac{\varphi_1\ \#\ \varphi_2}{(\varphi_1 \gg \psi_1)\ \#\ (\varphi_2 \gg \psi_2)}$$

As an example, the above disjunction between the left and right parts of an array concatenation is established using the D-LeftComp rule. Indeed, these two projection functors are obtained by the following compositions:

$$\lambda[i{<}\nu]\,.\,@[i+0{<}\nu+\mu]\,.\,\mathbf{1}_{\alpha} = \texttt{cut}\ \nu\ \mu\ \alpha \gg \left(@(1/2)\,.\,\mathbf{1}_{[\nu]\alpha}\right)$$
$$\lambda[i{<}\mu]\,.\,@[i+\nu{<}\nu+\mu]\,.\,\mathbf{1}_{\alpha} = \texttt{cut}\ \nu\ \mu\ \alpha \gg \left(@(2/2)\,.\,\mathbf{1}_{[\mu]\alpha}\right)$$

The D-LeftComp rule applies because the `cut` ν μ α projection functor is bijective and the tuple projections are trivially disjoint.

Conclusion

Projection functors formally describe mappings between the *structures* of compound data, i.e., independently of the values they contain. These functors only depend on array sizes and tuple arities, that are both statically known in our context. They allow to represent projection computations at compile-time.

Projection functors were primarily designed to provide a generic representation of *first-order array combinators* [Hen+17], using index computations. However, the handling of tuples turned out to be necessary to represent concatenation, or, more precisely, the reverse operation `cut`, as a single bijective projection functor.

Our choice of index language was driven by a need to balance expressiveness and formal manipulation. Our proposal, based on polynomials, allows to compose, simplify and compare projection functors easily, while still allowing to express complex operations like sampling.

A best-effort representation Since most of the properties of projection functors depend on undecidable constraints on multivariate polynomials, the internal representation of our compiler stores additional information alongside the functors. They are provided for the few projection functors used to represent a small set of built-in operators that construct views.

- Extra flags denote *injectivity* and *surjectivity*. For *invertible* functors, the inverse is also kept alongside the functor. These properties are propagated by composition. For instance, when composing invertible functors, the reversed composition of inverses is also computed.
- The list of composed projection functors is also stored alongside the reduced and normalized form of the functors. This list allows to over-approximate inclusion and disjunction relations, whereas the reduced form is used for comparison and as a fallback when composition-based analyses fail.

The *correctness* of projection functors is indirectly enforced. The built-in functors are valid under statically checked size assumptions, and correctness is preserved by composition.

The projection functors are the building blocks for the description of memory in our language. They are mainly used to check that memory allocations are correct and to deduce a valid scheduling of the *computational content* of programs. These points are described in the following chapters.

4 A Memory-Aware Declarative Language

Introduction

A program in a *sequential* language is a *sequence* of instructions that read from and write to a global *memory*. The values produced by a program result from the *order* in which instructions are executed, their semantics and the *allocation* of accessed values, i.e., the place where they are stored in memory.

Functional languages abstract from memory. A functional program specifies definitions independently of their representation in memory. The compiler or interpreter is in charge of finding a correct allocation so that the implementation follows the mathematical specification. In particular, this implies that the location of values may not be reused as long as they are needed for some computation. This recycling of memory may be determined at runtime by using dynamic allocation and relying on a companion program, the *garbage collector*, to retreive usused pieces of memory.

Data-flow, or *declarative* languages abstract from execution order. A data-flow program is made of mutually recursive definitions whose syntactic order is irrelevant. The compiler or the interpreter is in charge of finding a correct order so that values are computed before being used. This order may be determined at runtime, using for instance lazy evaluation.

A static language SCADE is a deterministic data-flow (synchronous) language. Because it targets safety-critical embedded software, SCADE programs must be compiled with strong guarantees on generated code. Notably, both memory and worst-case execution time must be statically bounded. This favors a static management of memory and execution order, which is easier to analyse and verify. The SCADE compiler, KCG, produces statically allocated and scheduled code.

Given a program, the SCADE compiler, named KCG, must determine a static allocation for the flows, i.e., the values of the language, and a static scheduling of computations. These choices are entangled. KCG gives priority to scheduling: the order of computations is selected under the hypothesis of the least restrictive allocation. Then allocation is optimized, considering the scheduling as fixed. The possibilities of allocation strongly depend on the selected order. For instance, the two sequences of assignments below, that have the same semantics, impose different allocation constraints.

```
x = 1;                  // (C)
z = x+1;
y = 2;
// x not used after
```

```
x = 1;                  // (C)
y = 2;
z = x+1;
// x not used after
```

The version on the left allows for `x` and `y` to be stored at the same location whereas this would lead to an incorrect implementation in the version on the right, because `x` would be read after writing `y`. In SCADE, the choice between one or the other implementation is out of programmer's control. Neither the scheduling nor the allocation may be constrained from the program. The above example highlights that optimizing allocation after scheduling gives fewer chances of memory sharing. With a view to giving control of memory usage from withing the language, allocation requirements must be taken into account by scheduling.

Memory First We propose a *Memory-Aware Declarative Language*, abbreviated MADL. It aims to bridge the gap between data-flow programs and their sequential implementations by making

both declarative and imperative worlds coexist in the same representation. Here and throughout the thesis, we use the term *imperative* to designate the allocation part of sequential programming models, leaving aside scheduling. Contrary to SCADE, MADL gives priority to allocation over scheduling. Hence, programs are expressed in an unordered form, called *two-sided* where memory is fixed. This situation is depicted in Figure 4.1.

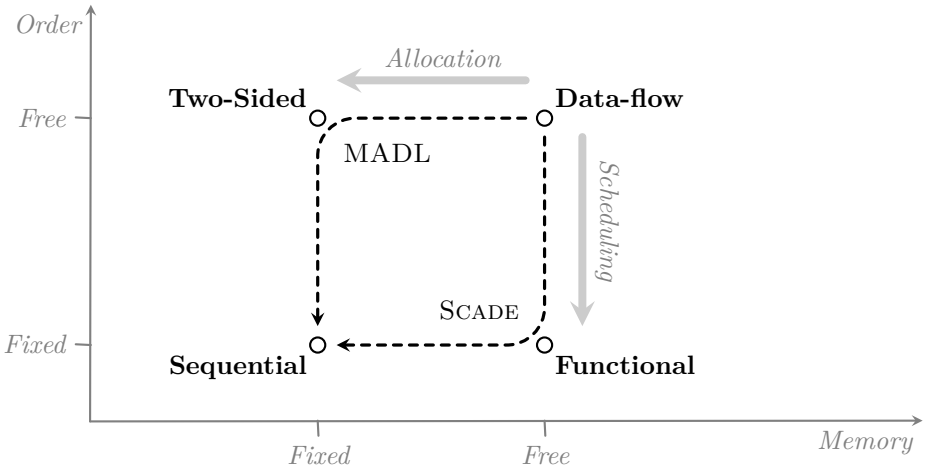


Figure 4.1: *The various programming models*

MADL is intended as a new intermediate representation for the compilation of SCADE. However, we study it as a standalone language. It does not rely on structural properties that would be checked on a higher-level language, but instead provides dedicated verifications to ensure the correctness of programs.

MADL is a *low-level* declarative language. Programs constrain allocation precisely. This contrasts with SCADE, where models abstract from implementation details. Although allocation is considered, MADL remains a *declarative* language. Its semantics are defined solely using data-flow dependencies. This results in a multi-level language, that we illustrate in Figure 4.2. Levels are separated by analyses rather than transformations. Hence, each level allows to interpret the same programs, but with additional information. As each analysis imposes some correctness criteria, the set of valid programs shrinks from one level to the next.

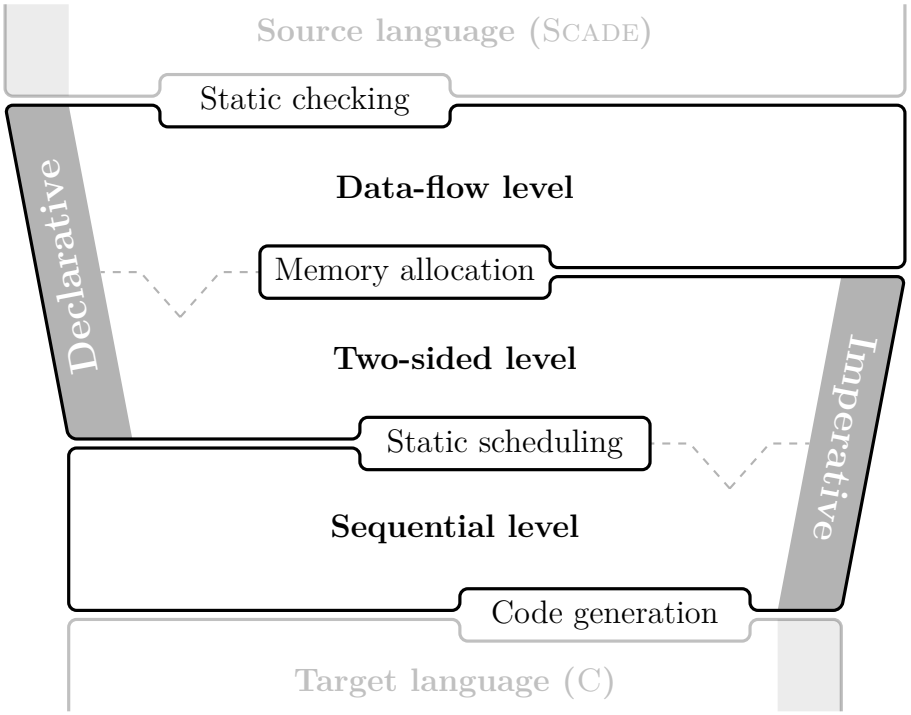


Figure 4.2: *The MADL language, a multi-level compilation intermediate*

We briefly review the three levels and outline how they appear in this thesis.

1. At the *data-flow* level, only data dependencies are taken into account. We sketch in Section 4.3.2 how a functional semantics could be defined for this level.

2. At the *two-sided* level, programs are understood by mixing data dependencies and allocation information. We elaborate in Section 4.2 on a *two-sided* semantics that conciliates the declarative and imperative sides.

3. At the *operational* level, programs may be interpreted using only allocation and scheduling information. This is the targeted execution model. It is explored by describing code generation in Section 7.2.

The design of MADL is guided by the two principles: (i) complete and precise memory allocations shall be expressible and (ii) these specifications shall preserve the declarative style, i.e., they should not interfere with the functional semantics, which is defined exclusively in terms of by data dependencies. Instead of relying on optimizations, the MADL language requires the programmer to specify how to use memory in the declarative language and the compiler checks that this is consistent with the data-flow dependencies. Specifying memory at this more abstract level benefits from structural properties that are difficult to reconstruct at a lower level, e.g., the uniqueness of reads and writes when iterating over arrays.

Imperative leaks in a declarative world vice versa Although the memory concerns must not have any effects on the semantics, the language provides some imperative features to allow for precise control of allocation.

First, in order to fully control memory, *every copy must be explicit*. They are represented with a dedicated construct, written `!`$\cdot$, which is handled as the identity function in the functional semantics. Consequently, all other constructs are copyless, notably:

- *Equations as aliases* — Declarative languages allow to introduce or remove variables without changing the semantics. This property, known as *referential transparency* or the substitution principle, is valid for the functional semantics, where the `=` symbol of equations denotes a declarative definition. In the two-sided semantics, a restricted version of referential transparency is extended to memory specification: equations are interpreted as physical aliases instead of imperative assignments. In MADL, a `=` b means that a and b are in the same location, whereas such assignments introduce copies in a language like C. These aliases allow to decouple memory allocation from program variables.

- *Data constructions as memory constraints* — The declarative style requires to construct all the sub-parts of a compound value before aggregating them. If unwisely compiled, each part must be copied into the final result. Some optimizations, e.g., the *built-in-place* pass of Sisal [Gau+97], try to allocate the parts at the right locations, but without any guarantees of success. The two-sided semantics turns this into a constraint. Building compound data, e.g., concatenating arrays, is only allowed in the two-sided semantics if the parts have been computed at their final destination in memory, i.e., side by side.

- *Data accesses as views* — As for data construction, accesses, i.e., projections, shall not introduce any copies. This applies to *views* as well, which extend the so called *first-order array combinators* of Futhark [Hen+17]. These operators amount to changing the structure of compound values without changing the data. In the two-sided semantics, the results of views share the same location as the original data, but their accesses are mapped so as to reflect the underlying transformation.

The second imperative aspect is to draw a distinction between *computational* and *structural* expressions. The former write new values to memory whereas the latter do not modify memory. Here lies the restriction on referential transparency: inserting or removing variables has no effect as long as it does not eliminate of duplicate *computational* expressions. Otherwise, the existence of the two-sided semantics might be affected. Two identical computational expressions may write their result at distinct locations, whereas a single one cannot write its result twice. The second situation is more restrictive regarding scheduling possibilities.

An explicit state The SCADE language describes stream functions. Values are streams, scalar operations are applied point-to-point and streams can be delayed (`pre`) and initialized (`->`). Such a stream abstraction conflicts with our guiding principles for two reasons: (i) delayed streams implicitly introduce copies that may only be eliminated if previous values are used before computing new ones and (ii) the initialization operator requires to distinguish the first instant from the others, which induces some branching and initialization flag management in transition functions.

The SCADE code generator optimizes many simple cases to avoid these extra costs, but this is not imposed by the language. MADL provides lower-level handling of state that eliminates the possibility of costly implementation. First, reading a delayed value — `last` — introduces scheduling constraints that ensure that no copy is needed. Reads must be completed before new values are written. Second, initializing the state — `init` — does not add any code to the transition function as initialization is performed between cycles. Both features are crucial for array iteration, because it uses the state to perform accumulation. To obtain reasonable performance, no branching shall be introduced inside these iterations.

4.1 The MADL Language

We refer the reader to Section 1.5 for an informal overview of MADL syntax and semantics. The abstract syntax of MADL is presented in Figure 4.3. It largely draws on the internal representation (AST) of our compiler. In particular, our presentation of expressions is unusual. In order to set up the different features of the language, we delay introducing the constructs that are dedicated to array iteration. They are discussed in Chapter 5. Throughout this chapter and the following ones, x and `f` denote, respectively, flow variables and operator names, taken from a given set of identifiers. We now review the different constructs progressively.

4.1.1 Top-level Constructs

MADL programs (pr) consist of a list of *top-level declarations* (td) that define potentially mutually recursive *operators* (o). Their definitions — $[ok]$ `f` b — specify an optional *operator kind* (ok), a name (`f`) and a *block* (b) called the *body* of the operator.

Operator kinds For each definition, the optional operator kind (ok) classifies the operator into one of four nested categories. The more restrictive at definition, the more permissive at instantiation. Figure 4.4 illustrates some of the expressions that are legal in the definitions of each kind of operator. When omitted, the stronger possible kind is selected by the compiler, according to the operator's contents.

- The `node` kind denotes *sequential* operators, also named *stateful*, because they are compiled into transition functions that expect a state. This kind does not impose any constraints on operator definitions, rather it restricts possible uses. These operators may only be instantiated at places that support stateful operators.
- The `func` kind marks *combinational* operators, also named *stateless*: their outputs only depend on their inputs, they are compiled into simple functions without state. In particular, these operators cannot instantiate the sequential ones (`node`).
- The `view` kind indicates *structural* operators. Their body must have no *computational content*, i.e., neither copies nor computations. Intuitively, structural operators define relations between the structure of their arguments and those of their results without modifying data. They are the central building blocks of memory specification.
- The `patt` kind stands for *bijective* structural operators. This is a restricted class of structural operators whose arguments and results are in bijection. Thanks to this property, they may be used in left-hand-side expressions.

Scopes Computation is organized into nested *scopes* — i: c $[r]$ — that are introduced either at operation declaration o or with a block construct (see Section 4.1.3). Each scope comprise an *interface* i) a local computation c and an optional *reset condition* r that specifies at which instants the block should be reset. Interfaces — a ... a «$s, \ldots, s$» `(`p`)` — are made of the following elements:

v	::=		***Meta variables***
	|	`_`	anonymous
	|	x	declared
	|	$'x$	undeclared
s	::=		***Size annotation***
	|	v	variable
	|	$0,\ 1,\ \ldots$	constant
	|	s `+` s	sum
	|	s `-` s	difference
	|	s `*` s	product
t	::=		***Type annotation***
	|	v	variable
	|	`int, bool, ...`	scalar
	|	$[s]t$	array
	|	t `*` $\ldots$ `*` t	tuple

				d	p	c
e	::=		***Expressions***			
	|	e`,` e	group	✓	✓	✓
	|	e `(`e`)`	compose	✓	✓	✓
	|	`(|` $\cdot$ `|)`	tuple	✓	✓	✓
	|	`[|` $\cdot$ `|]`	array	✓	✓	✓
	|	`f` «$s, \ldots, s$»	instance	⋆	†	✓
	|	$\cdot$ `of` t	type spec.	✓	✓	✓
	|	$\cdot$ `at` d	data spec.	✓	✓	✓
	|	v	data variable	✓	×	×
	|	`_`	ignore pattern	×	✓	×
	|	x	variable	×	✓	✓
	|	$\cdot$ `as` p	alias	×	✓	✓
	|	p `=`	equation	×	×	✓
	|	c`;` $\cdot$	local compute	×	×	✓
	|	op	operation	×	×	✓
	|	`!`$\cdot$	copy	×	×	✓
	|	`block` i`:` c $[r]$	block	×	×	✓
	|	`last` $\cdot$	last	×	×	✓
	|	$\cdot$ `init` c	init	×	✓	✓

ak	::=	`size` | `type` | `data`	***Abstract kinds***
a	::=	`(`ak x $\ldots$ x`)`	***Abstract parameters***
r	::=	`reset` c	***Reset conditions***
i	::=	a $\ldots$ a «$s, \ldots, s$» `(`p`)` `returns` `(`c`)`	***Interface***
ok	::=	`patt` | `view` | `func` | `node`	***Operator kinds***
o	::=	$[ok]$ `f` i`:` c $[r]$	***Operator***
td	::=	`let` $[$`rec`$]$ o `and` $\ldots$ `and` o	***Declarations***
pr	::=	$td \ldots td$	***Programs***

Figure 4.3: *Abstract syntax of the MADL language. Expressions are divided into three sub-languages: location annotations or data (d), patterns (p) and computations (c) (see Section 4.1.3).*

(i) The *locally abstract variables* — $'a' \ldots 'a$ — introduce abstract sizes, types or locations to be used in annotations. The last kind — `data` — represents memory locations. These meta-variables may not be substituted by inference passes.

(ii) The *size parameters* — «$s, \ldots, s$» — expose some sizes in the signature of the operator. At instantiation, these sizes may be specified to insert extra constraints to help type inference. When no size parameters are given, the delimiters « » are omitted as well.

(iii) An argument pattern p and result computation c. These syntactical classes are defined in Section 4.1.3.

Size parameters — «$s, \ldots, s$» — contain *size expressions*. For example, the following operator `f` may only be used — `f` «k» `(`$\ldots$`)` — with an even size k:

```
let f (size n) «2*n» (u of [2*n]_) returns (v of [2*n]_): ...
```

In most cases, size parameters expose *abstract sizes*. We thus provide a convenient syntactical shortcut that combines abstract variables and size parameters. If the list of size parameters is made of free variables, i.e., variables that are not introduced by the abstract parameters, the appropriate abstract sizes are inserted to bind them. For example, the two declarations below are equivalent.

```
let f (size n k) «n, k» (p) returns (c): c' r
let f «n, k» (p) returns (c): c' r
```

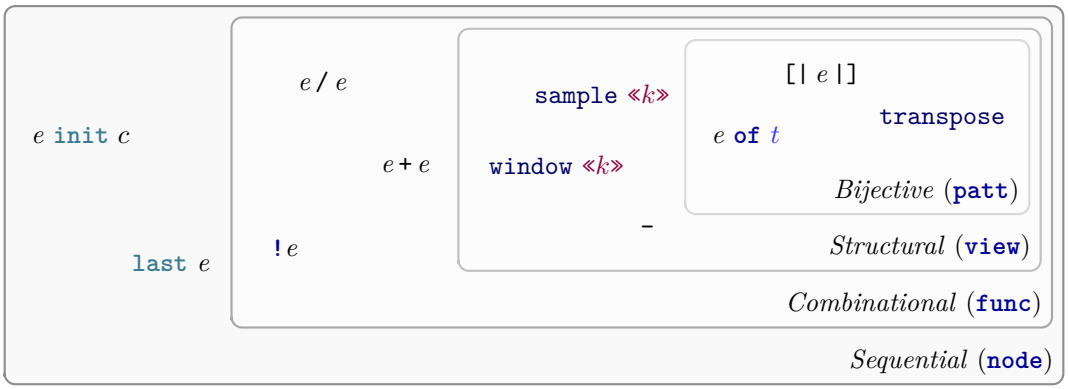


Figure 4.4: *Some of the authorized expressions for each kind of operators*

We distinguish abstract sizes from size parameters because they have different roles. Abstract sizes must not be substituted, while size parameters introduce sizes in the signatures to help inference.

Examples We illustrate the syntax of MADL by gradually defining four simple operators. Their definitions will be completed as constructs are presented (the ... denotes computations to complete). The `cons` and `snoc` operators provide a list-like interface for array manipulation by 'inserting' an element in first or last position. The `set_first` operator replaces the first element of an array. Finally, the `fib` operator computes the values of the Fibonacci sequence, using a state to save the two preceding ones.

```
let patt cons (type t) «n» (a, u) returns (v): ...
and patt snoc (type t) «n» (u, a) returns (v): ...

let func set_first (data d) (h, u) returns (v): ...

let node fib () returns (n): ...                    Incomplete program
```

The `cons` and `snoc` operators are *bijective*. They introduce a local abstract type t and a local abstract size parameter n, to be used in annotations. As these size and types are *abstract*, the definition may not impose any constraints on their values, which can thus be generalized. In the *stateless* `set_first` operator, a similar abstract memory variable d is introduced. Abstract locations resemble abstract sizes or types in that they may too not be constrained. This location variable will serve in location annotations. Last, the `fib` operator has no inputs and produces a single output.

4.1.2 Annotations

Beside the actual computations, the source code contains three kinds of annotations: *sizes*, *types* and *locations*. The latter describes memory allocation. Each kind of annotation reflects some static information that the compiler partially infers and uses for verification and code generation.

To relate sizes, types or locations that appear at several parts of the program, annotations possess their own variables, named *meta-variables* (v). The *anonymous* ones, denoted by a placeholder '_', represent *free variables* of the underlying size, type or location system. *Declared* variables are introduced by the abstract parameters of blocks and *undeclared* ones are prefixed with a '.

As defined in Figure 4.5, size annotations (s) and type annotations (t) enjoy a transparent syntactical representation of the formal sizes and types, except that the size annotation language contains an explicit difference. The description of memory locations is trickier: unlike for sizes and types, a direct description, based on a syntax for projection transformers, would neither be readable nor safe enough, because the correctness of projection transformers relies on uncheckable polynomial inequalities. Therefore, memory allocation is indirectly specified with *data annotations*, a sub-language of expressions that uses meta-variables. These location variables represent memory locations, independently of any values.

v ::=		***Meta variables***
	_	anonymous
|	x	declared
|	$'x$	undeclared

s ::=		***Size annotation***
|	v	variable
|	$0,\ 1,\ \ldots$	constant
|	$s + s$	sum
|	$s - s$	difference
|	$s * s$	product

t ::=		***Type annotation***
|	v	variable
|	`int, bool, ...`	scalar
|	$[s]t$	array
|	$t * \ldots * t$	tuple

Figure 4.5: *Syntax of MADL annotations*

Annotations are partial descriptions of sizes, types and locations. To distinguish them from the compiler computed sizes, types and locations, we use latin variables instead of greek ones for the syntactical annotations.

Examples To supplement the above definition skeletons with annotations, we introduce two expressions —e `of` t and e `at` d— that insert, respectively, type and data constraints. The full language of data annotations —d— is introduced in Section 4.1.3. In our simple examples, data annotations are only location variables.

```
let patt cons (type t) «n» (a of t, u of [n]t) returns (v of [n+1]t): ...
and patt snoc (type t) «n» (u of [n]t, a of t) returns (v of [n+1]t): ...

let func set_first (data d) (h, u at d) returns (v at d): ...    Incomplete program
```

In the `cons` and `snoc` operators, the type annotations completely specify the type of the interface, but memory locations are left implicit. The a, u, and v variables are actually not independent, but this will be inferred from their definitions.

In the `set_first` operator, the data annotations —$\cdot$ `at` d— now make clear that argument u and result v share the same memory location. A similar description of sharing was proposed by Gérard et al. [Gér+12].

4.1.3 Expressions

Expressions represent first-order transition functions that expect multiple arguments and produce multiple results. Their syntax is recalled in Figure 4.6. At this point, such generality seems largely excessive for simple expressions, e.g., variables are handled as nullary functions that produce a single value. However, this more compact syntax simplifies the formalism and follows more closely our implementation. Moreover, array iteration, explored in Chapter 5, will build on this second order syntax.

Expressions are described by their *arity* —$n_i \to n_o$— where n_i is the number of expected arguments and n_o the number of produced results. This type skeleton is formally defined in Section 4.1.4. Intuitively, expression arity represents the number of wires or input and output flows to connect one expression with another.

The three flavors Expressions serve three purposes: (i) *data annotations* (d) represent memory locations, independently of any value, (ii) *patterns* (p) are the left-hand-side expressions in equations and (iii) *computations* (c) are the right-hand-side expressions. Because these syntactical classes share several constructs, their definitions are factorized in the *expressions* class (e). To obtain each sub-language, the occurrences of e in the two first constructs are replaced by d, p or c, respectively, and the constructs marked with $\times$ are dropped.

In particular, the introduction of an alias —$\cdot$ `as` p— and initialization —$\cdot$ `init` c— are allowed in both patterns and computations, but not in data annotations.

The case for operator instances is subtle: only structural operators (`view`) may be instantiated inside data annotations (the $\star$ mark), while patterns are even stricter and require bijective (`patt`)

e ::=	***Expressions***	d	p	c
e , e	group	✓	✓	✓
e (e)	compose	✓	✓	✓
(| · |)	tuple	✓	✓	✓
[| · |]	array	✓	✓	✓
f «$s, \ldots, s$»	instance	⋆	†	✓
· of t	type spec.	✓	✓	✓
· at d	data spec.	✓	✓	✓
v	data variable	✓	×	×
_	ignore pattern	×	✓	×
x	variable	×	✓	✓
· as p	alias	×	✓	✓
p =	equation	×	×	✓
c ; ·	local compute	×	×	✓
op	operation	×	×	✓
!·	copy	×	×	✓
block i: c [r]	block	×	×	✓
last ·	last	×	×	✓
· init c	init	×	✓	✓

Figure 4.6: *Syntax MADL expressions*

operators (the † mark). This cannot be ensured syntactically, i.e., during parsing, and is checked by a subsequent verification.

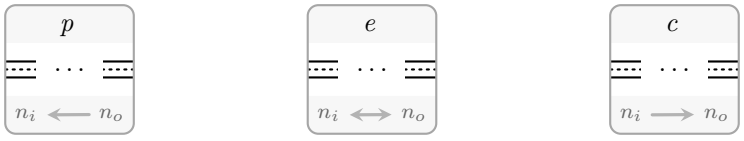
First order and a half In the abstract syntax of MADL expressions, the '·' markers stand for a sub-expression to plug in, using the compose construct. Hence the declaration of a variable y as a copy of a variable x is represented by the leftmost expression:

$$(y\ =\ )\ (!\cdot\ (x)) \quad \equiv \quad (y\ =\ )\ (!x) \quad \equiv \quad y\ =\ !x$$

Admittedly, such a decomposed description is barely readable and even less writable. We provide a pervasive syntactical shortcut by allowing expressions — location annotations, patterns or computations, depending on the kind of the expression — to be inserted in place of the '·' markers. The two expressions on the right are equivalent and our simple copy recovers a much more peasant form.

Compared to classical abstract syntax trees, the composition-based representation frees the formalization (and the compilation!) from the noise introduced by managing sub-expressions. Moreover, it allows handling the intrinsically second-order constructs, instances, operations and blocks, just like any others. Despite this, *MADL is not a higher-order language.* The syntax of expressions is second order, but the language is first order in essence. Flows, in particular variables, only contain data (scalars or compound values). That is the reason for our midway syntax. The composition construct has an applicative syntax instead of the more general form $e \circ e$.[1]

Furthermore, we decided to eliminate of the usual expression/equation distinction. In our opinion, it is purely syntactical and a unified description is more suitable from a diagrammatic point of view. Without any formalization ambition, we illustrate expressions as follows:

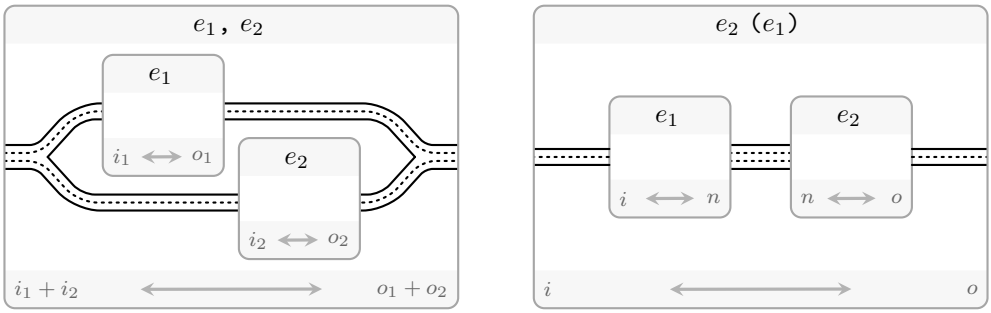


The numbers n_i, n_o indicate the input and output arity, on the left and right respectively. The bottom arrows indicate the direction of the data flow: *patterns* (p) define their arguments (on the left) from their results (on the right) while *computations* (c) build their results out of their arguments. The double arrows represent expressions that may be used in both patterns and computations. Expressions expect and produce multiple values, which is depicted by multiple lines.

[1] Applications are the most-used degenerate case of composition where the first function is nullary.

The oakum For causality reasons, synchronous languages must draw a clear distinction between streams of structures, e.g., streams of tuples, and structures of streams. The former build run-time values that are actually structures, while the latter are unraveled at compile-time, so that the various components to be scheduled independently. In LUSTRE, similar compile-time structures are provided with so-called *lists*. SCADE follows the same approach and SCADE6 named these wiring constructs *groups*. We keep this nomenclature.

The group — e_1 , e_2 — and compose — e_2 (e_1) — constructs structure the description of the computation, without playing any role in the semantics or compilation. The former is like parallel composition: it dispatches inputs and gathers outputs. The latter describes sequential composition: it plugs the result of e_1 into the arguments of e_2. These constructs are depicted below.

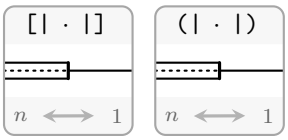


Expressions are flattened since the group construct is associative. Hence, we use the n-ary group notation $e_1, \ldots, e_n$. For instance, the following expressions have arity $0 \to 4$, they are equivalent (denoted $\equiv$):

$$x\text{ , }(y\text{ , }z)\text{ , }t \quad \equiv \quad x\text{ , }y\text{ , }z\text{ , }t$$

The common constructs Along with the grouping constructs, several other expressions may be used in any context. They are all structural operations which do not need any computation and thus do not introduce any instructions in the generated code.

- The *tuple* — `(| · |)` — and *array* — `[| e |]` — constructors build a single value from their inputs: they have arity $n \to 1$. For instance, the expression `[| x , y , z |]`, that corresponds to `[| · |] (x , y , z)`, builds an array of size 3 containing the values of variables x, y and z.



- The *type annotations* — e **of** t — and *data annotations* — e **at** d — introduce type or memory location constraints. Because type annotations describe single flows, they has arity $1 \to 1$. By contrast, the data annotations may describe multiple flows, hence they have $n \to n$ arity.

- *Instantiation* — **f** «$s, \ldots, s$» — may occur in all expressions, with the aforementioned restrictions: patterns requires *bijective* operators and data annotations may only use *structural* ones.

Unlike groups, tuples and arrays affect semantics and compilation. In particular, they are strict: a result depends on all its parts. Anticipating on causality analysis, consider, for instance, the following versions of the identity function.

```
let id (x) returns (z):
    y,z = x,y

let id (x) returns (z):
    y = x;
    z = y;
```

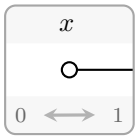
```
let id (x) returns (z):
  (|y,z|) = (|x,y|)
```

```
Error: Non causal definition of y
Note: Dependency cycle: y -> y
Note: Variable y used here
```

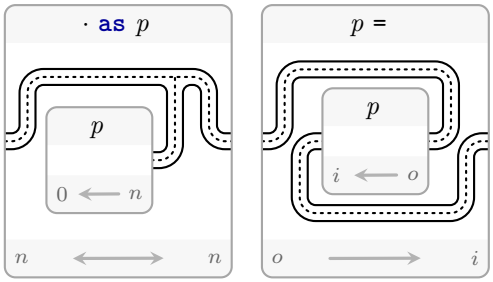
The left definitions are valid and equivalent. The right definition is rejected because the variable y depends on itself. This program would be incorrect for other reasons (memory locations, ...) but causality is checked first.

The specific constructs Like type and size annotations, location annotations possess their own *meta-variables* — v —, to be used in place of term variables. Notice that the ignore construct — `_` — is specific to pattern expressions. In location annotations, the identical syntax — `_` — denotes an anonymous location variable.

- *Variables.* Term expressions, i.e., patterns and computations, use *term variables* — x — whose scoping rules are detailed below. The *ignore* construct — `_` — is like for an unnamed variable. It allows to omit a value in patterns without the compiler complaining about an unused variable. Variables represent a single value. They have arity $0 \to 1$.



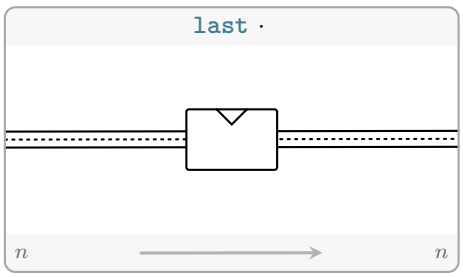
- *Declarations. Aliases* — `· as p` — and *equations* — `p =` — introduce patterns. They declare the variables contained in p. They are subtly different. The former require the pattern p to have arity $0 \to n$ and the resulting expression has arity $n \to n$ while the latter is a reversed composition. If p has arity $i \to o$, the expression `p =` has arity $o \to i$. Both constructs are illustrated at right.

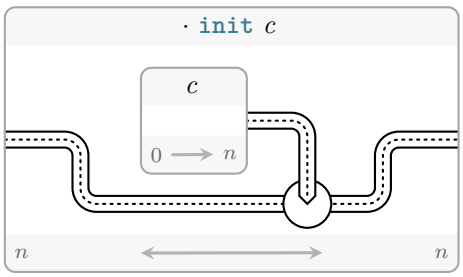


- *Computational content.* Only very few constructs actually boil down to instructions in the generated code. The external *operations* — *op* — are computations that are transparently forwarded to the target language. They are supposed to be free of side effects. In our prototype, they encompass arithmetic expressions: the term $x + 5 * y$ is encoded as a dedicated binary operation — `_+5*_` — that is applied to the group of sub-terms (x and y): `(_+5*_) (x, y)`. The copy operation is set apart and provided as a dedicated construct — `!·` — of arity is $1 \to 1$. From a functional point of view, it is the identity function, but its argument and result are not *equivalent*, a notion defined below. In the two-sided semantics, copies allow to duplicate values in memory.

  The computational content also includes instances of operators — `f` $\ll s_1, \ldots, s_n \gg$ — that contain computations, i.e., `node` and `func`. For readability, we use *local computation* — `c; ·` — to group expressions that have no arguments or results, i.e., that have arity $0 \to 0$. In particular, they allow to separate equations: p_1 `=` c_1`;` p_2 `=` c_2`;` ...

- *Sequential constructs.* In Lustre, unit delays are introduced with the operator `pre`, that gives access to the previous value of a flow. To help with sharing values between the states of automata, Lucid Synchrone [Ham02] and Scade [CPP17] supplement this operator with a `last` construct that denotes the previous value of a variable in its declaration scope. This gives a special role to variables and a state-based flavor. MADL goes one step further. Our last construct — `last ·` — is based on memory locations instead of variables. Hence, it applies to a general expression and has arity $n \to n$. To be valid at the first instant and upon reset, expressions used in `last` must be *initialized* — `· init` c — with a computation expression c of appropriate arity $0 \to n$. Initialization may be introduced in computations or in patterns. This expression also has arity $n \to n$.





  The diagrams give the intuitive semantics of initialization — `· init` c —. It initializes the flows that pass from its inputs to its outputs with the value computed by c. We will come back to initialization shortly.

Examples Let us now complete the definitions of our running examples

```
let patt cons «n» (a of 't, u of [n]'t) returns (v of [n+1]'t):
  v = concat «1,n» ([| a |], u);
and patt snoc «n» (u of [n]'t, a of 't) returns (v of [n+1]'t):
  v = concat «n,1» (u, [| a |]);
```

```
let func set_first (data d) (h, u at d)
returns (v at d):
  cons (_, t) = u;
  v = cons (!h, t);
```

```
let node fib () returns (n):
  n        = !last a;
  a init 0 = !last b;
  b init 1 = n + a;
```

The `cons` and `snoc` operators are defined using an array constructor and the built-in `concat` bijective operator. The `set_first` operator instantiates the `cons` pattern in both pattern and computations without specifying the size parameters («n»). We could have equivalently written `cons «_» (...)`. Last, our implementation of `fib` starts at 0: it computes the sequence 0, 1, 1, 2, 3, *etc.* This operators *copies* the previous values (`last a`, `last b`). The existence of the two-sided semantics depends on them. They allow the different variables to have distinct memory locations, which allows the existence of a valid scheduling.

Initializing expressions The `init` construct applies to *expressions*. The position of initialization does not matter as long as they are inserted on *equivalent* flows, a notion that we formalize in Section 4.3.4. To illustrate this flexibility, here are three semantically equivalent versions of the Fibonacci operator.

```
let fib () returns (n):
  n        = !last a;
  a init 0 = !last b;
  b init 1 = n + a;
```

```
let fib () returns (n):
  n = !last (a init 0);
  a = !last (b init 1);
  b = n + a;
```

```
let fib () returns (n):
  n = !last a;
  a = !last b init 0;
  b = n + a init 1;
```

The first one introduces `init` in patterns, while the others insert them in computations (the last one initializes $n + a$). This example might be misleading since no variables are even necessary. For instance, a 0-initialized unit delay may be equivalently defined as:

```
let pre (i) returns (o):
  o = !last (!i init 0);
```

```
let pre (i) returns (o):
  m init 0 = !i;
  o = !last m;
```

```
let pre (i) returns (o):
  m init 0;
  m = !i;
  o = !last m;
```

The version on the left initializes the unamed flow `!i`. It is equivalent to the two other versions. The rightmost one shows how initialization may be set aside in its own expression.

To illustrate initialization, the expression p `init` c `=` e `as` q is depicted in Figure 4.7. Although the `init` construct is located on the same side as p, it initializes all the flows that are connected to p by structural expressions. They are depicted by the wires that traverse expressions. Both patterns p and q are initialized as well as any pattern that is connected to the result of e.

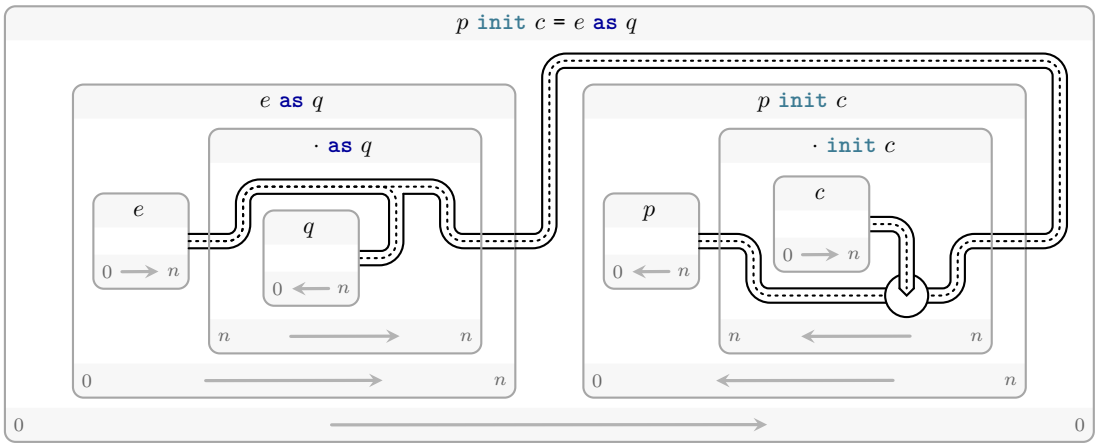


Figure 4.7: *Indirect initialization*

Builtin operators A set of array primitives is provided by the compiler. These structural operators are the language counterpart of projection functions, as their memory specification will highlight (see Chapter 6). Figure 4.8 summarizes these operators, which where introduced in detail in Section 2.4.3.

In MADL, type schemes — $\forall \vec{\nu}.\ \forall \vec{\alpha}.\ \langle\!\langle \vec{\eta} \rangle\!\rangle\ \vec{\tau_i} \longrightarrow \vec{\tau_o}$ — are made of a set of universally quantified sizes — $\vec{\nu}$ — and types — $\vec{\alpha}$ —, some size parameters — $\langle\!\langle \vec{\eta} \rangle\!\rangle$ — and inputs — $\vec{\tau_i}$ — and output — $\vec{\tau_o}$ — types. Apart from the `concat` operator, all these built-ins have arity $1 \to 1$. The type signature of `concat` — $[\nu]\alpha,\ [\mu]\alpha \longrightarrow [\nu + \mu]\alpha$ — indeed has two arguments and one result. To distinguish groups from tuples, the former are denoted by comma separated lists of types — $\tau_1, \ldots, \tau_n$ — instead of the tuples syntax $\tau_1 * \ldots * \tau_n$. For instance, the input of the `concat` operator is a group of two arrays.

$$
\begin{aligned}
\texttt{reverse} &: \forall \nu.\ \forall \alpha.\ [\nu]\alpha \longrightarrow [\nu]\alpha \\
\texttt{transpose} &: \forall \nu\, \mu.\ \forall \alpha.\ [\nu]\,[\mu]\alpha \longrightarrow [\mu]\,[\nu]\alpha \\
\texttt{concat} &: \forall \nu\, \mu.\ \forall \alpha.\ \langle\!\langle \nu, \mu \rangle\!\rangle\ [\nu]\alpha,\ [\mu]\alpha \longrightarrow [\nu + \mu]\alpha \\
\texttt{window} &: \forall \nu\, \kappa.\ \forall \alpha.\ \langle\!\langle \kappa \rangle\!\rangle\ [\nu + \kappa - 1]\alpha \longrightarrow [\nu]\,[\kappa]\alpha \\
\texttt{sample} &: \forall \nu\, \kappa.\ \forall \alpha.\ \langle\!\langle \kappa \rangle\!\rangle\ [(\nu - 1) * \kappa + 1]\alpha \longrightarrow [\nu]\alpha \\
\texttt{split} &: \forall \nu\, \kappa.\ \forall \alpha.\ \langle\!\langle \kappa \rangle\!\rangle\ [\nu * \kappa]\alpha \longrightarrow [\nu]\,[\kappa]\alpha \\
\texttt{flatten} &: \forall \nu\, \kappa.\ \forall \alpha.\ \langle\!\langle \kappa \rangle\!\rangle\ [\nu]\,[\kappa]\alpha \longrightarrow [\nu * \kappa]\alpha \\
\texttt{repeat} &: \forall \nu.\ \forall \alpha.\ \langle\!\langle \nu \rangle\!\rangle\ \alpha \longrightarrow [\nu]\alpha
\end{aligned}
$$

Figure 4.8: *The type schemes of built-in operators*

4.1.4 Structure of the Language

Before illustrating MADL with examples and explaining its semantics, we define some well-formedness conditions on programs.

Expression arity Flows handle first-order data. If expression state is ignored (see Chapter 6), expressions are described with a type $\tau_1,\ \ldots,\ \tau_n \longrightarrow \tau'_1,\ \ldots,\ \tau'_p$ where $\tau_1,\ \ldots,\ \tau_n$ are the types of arguments and $\tau'_1,\ \ldots,\ \tau'_p$ are the types of results. The skeleton of this signature is summed up by the *expression arity* $n \to p$. We use a comma — , — as separator to distinguish multiple flows from tuples. For instance, the signature $\tau_1,\ \tau_2 \longrightarrow \tau_1 * \tau_2$ describes the pair constructor, whose arity is $2 \to 1$. It expects two values and produces single tuple.

The rules for arity are defined in Figure 4.9. The arity of an expression e is independent of whether e appears in a pattern, in a computation or in a location annotation. Hence, in the expression — $(y,\ z)$ `=` $(x,\ y)$ — both the sub-expressions $y,\ z$ and $x,\ y$ have arity $0 \to 2$. The diagrammatic representation and syntax tree annotated with arity are depicted below:

Expression Arity $\quad e : n_i \to n_o$

$$\text{A-Group}\ \frac{e_1 : n_1 \to p_1 \qquad e_2 : n_2 \to p_2}{e_1,\ e_2 : n_1 + n_2 \to p1 + p_2} \qquad \text{A-Compose}\ \frac{e_1 : n \to p \qquad e_2 : p \to q}{e_2\ (e_1) : n \to q}$$

$$\text{A-Tuple}\ \frac{}{(|\ \cdot\ |) : n \to 1} \qquad \text{A-Array}\ \frac{}{[|\ \cdot\ |] : n \to 1} \qquad \text{A-Inst}\ \frac{\mathcal{O}(\texttt{f}) = n \to p}{\texttt{f}\ \langle\!\langle s, \ldots, s \rangle\!\rangle : n \to p}$$

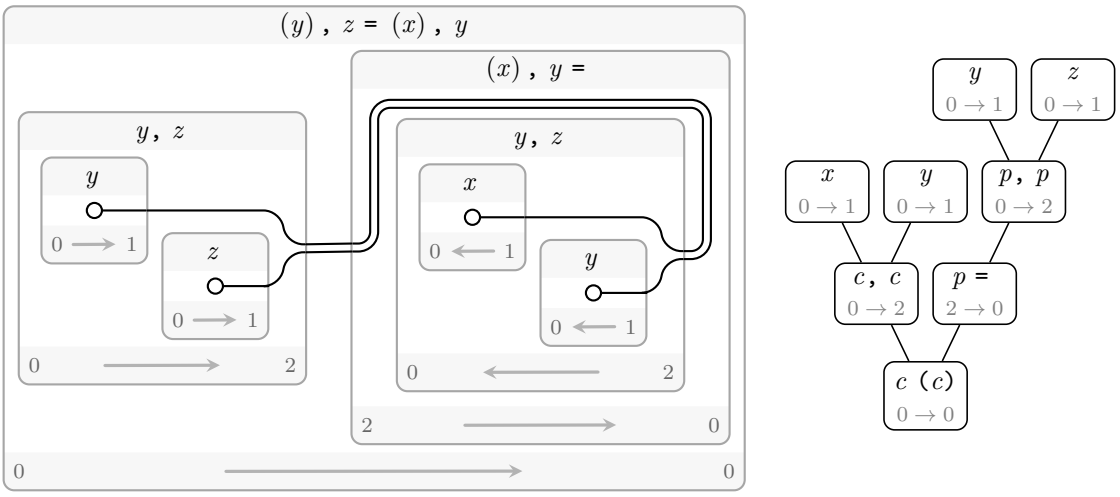
$$\text{A-Type}\ \frac{}{\cdot\ \texttt{of}\ t : 1 \to 1} \qquad \text{A-Data}\ \frac{d : 0 \to n}{\cdot\ \texttt{at}\ d : n \to n} \qquad \text{A-Data-var}\ \frac{}{v : 0 \to 1}$$

$$\text{A-Ign}\ \frac{}{\_ : 0 \to 1} \qquad \text{A-Var}\ \frac{}{x : 0 \to 1} \qquad \text{A-Ali}\ \frac{p : 0 \to p}{\cdot\ \texttt{as}\ p : p \to p} \qquad \text{A-Equ}\ \frac{p : n \to p}{p\ \texttt{=}\ : p \to n}$$

$$\text{A-Compute}\ \frac{c : 0 \to 0}{c\texttt{;}\ \cdot : n \to n} \qquad \text{A-Op}\ \frac{\mathcal{O}(op) = n \to p}{op : n \to p} \qquad \text{A-Copy}\ \frac{}{\texttt{!}\cdot : 1 \to 1} \qquad \text{A-Last}\ \frac{}{\texttt{last}\ \cdot : n \to n}$$

$$\text{A-Block}\ \frac{p : 0 \to n \qquad c : 0 \to p \qquad c' : 0 \to 0 \qquad c_r : 0 \to 1}{\texttt{block}\ a\ \ldots\ a\ \langle\!\langle s, \ldots, s \rangle\!\rangle\ (p)\ \texttt{returns}\ (c)\texttt{:}\ c'\ \texttt{reset}\ c_r : n \to p} \qquad \text{A-Init}\ \frac{c : 0 \to n}{\cdot\ \texttt{init}\ c : n \to n}$$

Figure 4.9: *Arity of expressions. The $\mathcal{O}$ function associates an arity to each operator `f` and operation op*



Arity check *Implementation note (4)*

Our prototype checks the arity of expressions during name resolution. This early verification simplifies the subsequent passes that assume well structured terms.

Name-spaces and scopes Because they are syntactically separated, the various identifiers are taken from five distinct name-spaces, that allow to reuse names without clashes. (i) The declaration name-space, ranged over by `f`, contains the names of operators. These identifiers are only used for operator declarations and instantiations. Operators share the same name-space regardless of their kind (`node`, `patt`, ...), so that name resolution is independent of semantic properties. (ii) The term variables, ranged over by x, represent single values. (iii-iv-v) The variables of annotations — size, type and data — are taken among dedicated name-spaces, that contains both declared and undeclared variables.

Each name-space is *scoped*. For operators, scopes open at each top-level declaration (`let`), hence mutually recursive definitions must have distinct names. For the other name spaces, scopes open at each block. Meta variables (size, type and location) are declared by the abstract parameters and terms variables are declared by the patterns of the block, as formalized in Figure 4.10.

Unlike in OCaml, the scope of an undeclared variable (`'x`) is the smallest block that contains all its occurrences. For instance, the MADL `id` function below on the left is valid: the undeclared type `'b` will be substituted by an abstract type a. On the contrary, an equivalent OCaml definition on the right is ill-typed, because the scope of an undeclared variable is the outer `let`.

```
let id (a) returns (b):
  b = (block (type a) (x of a)
       returns (x of 'b)) (a);
```

```
let id a =                              (* OCaml *)
  (fun (type a) (x:a): 'b -> x) a

Error: types mismatch
The type constructor a would escape its scope
```

Blocks and interface The block expression — `block` b — introduces a local block This opens a new scope for meta and term variables and allows for a local reset. For instance, a resettable integral of a signal s may be computed with:

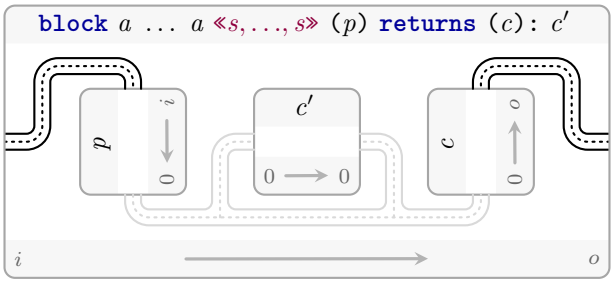
```
let integral (s, c) returns (i):
  i = (block (ls) returns (li):
         li init 0 = last li + ls;
       reset c) (s);
```

This syntax emphasizes the nature of blocks. They are expressions that may be composed like any other ones. In a block $a \ \ldots \ a$ «$s, \ldots, s$» (p) `returns` (c): c' `reset` c_r, that is either local or the body of an operator, the argument p has arity $0 \to i$, the result c has arity $0 \to o$ and the local computation has arity $0 \to 0$. The reset condition is a single value, i.e., of arity $0 \to 1$. The block is depicted below. Although the different parts look disconnected, they are actually linked by the local variables (represented by the faded out links).

$$
\begin{array}{rl}
\mathcal{FV}(\ e_1\texttt{, }e_2\ ) = \mathcal{FV}(\ e_1\ ) \cup \mathcal{FV}(\ e_2\ ) & \mathcal{DV}(\ e_1\texttt{, }e_2\ ) = \mathcal{DV}(\ e_1\ ) \uplus \mathcal{DV}(\ e_2\ ) \\
\mathcal{FV}(\ e_1\ \texttt{(}e_2\texttt{)}\ ) = \mathcal{FV}(\ e_1\ ) \cup \mathcal{FV}(\ e_2\ ) & \mathcal{DV}(\ e_1\ \texttt{(}e_2\texttt{)}\ ) = \mathcal{DV}(\ e_1\ ) \uplus \mathcal{DV}(\ e_2\ ) \\
\mathcal{FV}(\ \texttt{(|}\cdot\texttt{|)}\ ) = \emptyset & \mathcal{DV}(\ \texttt{(|}\cdot\texttt{|)}\ ) = \emptyset \\
\mathcal{FV}(\ \texttt{[|}\cdot\texttt{|]}\ ) = \emptyset & \mathcal{DV}(\ \texttt{[|}\cdot\texttt{|]}\ ) = \emptyset \\
\mathcal{FV}(\ \texttt{f «}s, \ldots, s\texttt{»}\ ) = \emptyset & \mathcal{DV}(\ \texttt{f «}s, \ldots, s\texttt{»}\ ) = \emptyset \\
\mathcal{FV}(\ \cdot\ \texttt{of}\ t\ ) = \emptyset & \mathcal{DV}(\ \cdot\ \texttt{of}\ t\ ) = \emptyset \\
\mathcal{FV}(\ \cdot\ \texttt{at}\ d\ ) = \emptyset & \mathcal{DV}(\ \cdot\ \texttt{at}\ d\ ) = \emptyset \\
\mathcal{FV}(\ \texttt{\_}\ ) = \emptyset & \\
\mathcal{FV}(\ x\ ) = \{x\} & \mathcal{FV}(\ x\ ) = \emptyset \\
\mathcal{FV}(\ \cdot\ \texttt{as}\ p\ ) = \mathcal{FV}(\ p\ ) & \mathcal{DV}(\ \cdot\ \texttt{as}\ p\ ) = \mathcal{FV}(\ p\ ) \\
\mathcal{FV}(\ p\ \texttt{=}\ \ ) = \mathcal{FV}(\ p\ ) & \mathcal{DV}(\ p\ \texttt{=}\ \ ) = \mathcal{FV}(\ p\ ) \\
\mathcal{FV}(\ c\texttt{;}\ \cdot\ ) = \mathcal{FV}(\ c\ ) & \mathcal{DV}(\ c\texttt{;}\ \cdot\ ) = \mathcal{DV}(\ c\ ) \\
\mathcal{FV}(\ op\ ) = \emptyset & \mathcal{DV}(\ op\ ) = \emptyset \\
\mathcal{FV}(\ \texttt{!}\cdot\ ) = \emptyset & \mathcal{DV}(\ \texttt{!}\cdot\ ) = \emptyset \\
\mathcal{FV}(\ \texttt{block}\ b\ ) = \mathcal{FV}_b(b) & \mathcal{DV}(\ \texttt{block}\ b\ ) = \mathcal{DV}_b(b) \\
\mathcal{FV}(\ \texttt{last}\ \cdot\ ) = \emptyset & \mathcal{DV}(\ \texttt{last}\ \cdot\ ) = \emptyset \\
\mathcal{FV}(\ \cdot\ \texttt{init}\ c\ ) = \emptyset & \mathcal{DV}(\ \cdot\ \texttt{init}\ c\ ) = \emptyset
\end{array}
$$

$$
\begin{array}{l}
\mathcal{FV}_b(\ \texttt{(}p\texttt{)}\ \texttt{returns}\ \texttt{(}c\texttt{)}\texttt{:}\ c'\ \texttt{reset}\ c_r\ ) = \mathcal{FV}(\ c_r\ ) \cup \mathcal{FV}(\ c'\texttt{;}\ c\ ) \setminus (\mathcal{FV}(\ p\ ) \uplus \mathcal{DV}(\ c'\texttt{;}\ c\ )) \\
\mathcal{DV}_b(\ \texttt{(}p\texttt{)}\ \texttt{returns}\ \texttt{(}c\texttt{)}\texttt{:}\ c'\ \texttt{reset}\ c_r\ ) = \mathcal{DV}(\ c_r\ )
\end{array}
$$

Figure 4.10: *The free variable of patterns and computations —* $\mathcal{FV}(\ p\ )$; $\mathcal{FV}(\ c\ )$ — *and declared variables of computation —* $\mathcal{DV}(\ c\ )$*. We omit abstract meta-variables and size parameters in blocks.*



The above syntax makes the scope of the block clear: it contains p, c, c'. However, it is not convenient, because it requires to make inputs (resp. outputs) to coincide with arguments (resp. results). Moreover, this form would barely be readable without the sized parentheses of our abstract syntax. In particular, it is impractical for the concrete syntax of MADL. For that reason, we use an equation-like block interface where the inputs and outputs are plugged inside the interface. With this syntax, our integral operator is defined as:

```
let integral (s, c) returns (i):
  block (ls = s) returns (i = li):
    li init 0 = last li + ls;
  reset c
```

Although more condensed, the equation syntax for blocks hides an important point: the right-hand-side (resp. left-hand-side) of arguments (resp. results), i.e., s and i in the example above, are part of the enclosing scope instead of the inner block's scope. Since scopes are nested, the content of blocks may use the variables of the enclosing scope. Our resettable integral may thus be defined as:

```
let integral (v, c) returns (i):
  block returns (i = s):
    s init 0 = last s + v;
  reset c
```

Whereas the arguments of a block are unnecessary in this example, the iteration construct, presented in Chapter 5 will use them extensively. Moreover blocks allow to introduce abstract types, sizes or locations. In the following operator,

```
let fn (u of [4]_) returns (o):
  block (size n) (lu of [n]_ = u) returns (o = lo):
    ...
```

the size n of the local array lu is quantified universally. It may not be substituted. In particular, the content of the block may not rely on the equality $n = 4$. Local abstract variables have several advantages:

- The type and location system, presented in Chapter 6, ensures that the local abstract sizes do not escape their scopes, i.e., that the content of a block is independent of its concrete values.
- Local abstract sizes help size inference by allowing polynomial constraints decomposition, as Section 2.4.4 explains.

Reset conditions Blocks also hold reset conditions — `reset` c — that specify at which cycles the content of the block is to be initialized. The initialization computations are introduced by the `init` constructs of their body: upon reset, each contained `init` is executed once, writing its value to the state memory location. The reason for limiting the reset to a block instead of any expression comes from the model of state which is detailed in Section 4.3.4. The intuition is simple: blocks are considered as atomic operations, i.e., each output depends on all the inputs, whose evaluation is preceded by a conditional reset of their state. Since the initialization occurs before the evaluation of the block, it may not use any of the values computed in the block.

Causality Operators are made of recursive definitions. However, this recursion is limited. Programs such as $x = x$ or $x = x + 1$ must be rejected because they are not *constructive*, namely, they do not define a unique stream in a computable way. To rule out these cases at the functional semantics level, a syntactical causality analysis ensures that all dependency cycles contain a `last`, e.g., x `init` $0 =$ `last` $x + 1$ is correct but $x = x + 1$ isn't. For now, this check is simplistic: all operators are considered atomic. Several type systems [CP01; Ben+14] have been proposed and used to track causality relations more precisely. They are candidates for future extensions.

We formalize causality by associating *time-stamps* to flows, following [Ben+14]. Time-stamps organize the computation inside each cycle by giving an execution rank. Expressions with a given time-stamp may be evaluated as soon as all the expressions with smaller time-stamps have been evaluated. Time-stamps d are taken from a set equipped with a complete order, denoted $<$. Expressions are described with a *causality signature* — $\vec{d_i} \longrightarrow \vec{d_o}$ — that collects argument time stamps ($\vec{d_i}$) and result time stamps ($\vec{d_o}$). *Causality constraints* — $d <_k d'$ — are order constraints over time-stamps, with a *direction* k that is either $+$ or $-$: $d <_+ d'$ denotes the constraint $d < d'$ while $d <_- d'$ denotes the constraint $d' < d$. We extend this notation for vectors of time-stamps: $\vec{d} <_k \vec{d'}$.

Constructiveness is defined by the *causality checking* relation $\Gamma \vdash_k e : \vec{d_i} \longrightarrow \vec{d_o} \dashv \mathcal{C}$. Here as well, the k index is a *direction* ($+$ or $-$). This judgement relates an environment Γ, an expression e that is a pattern if $k = -$ or a computation if $k = +$, a causality signature and a set of constraints $\mathcal{C}$. The environment associates to each variable a time-stamp. The judgement is defined in Figure 4.11.

In MADL, all computations (instantiations, operations, copies, blocks) as well as tuples and arrays are strict. Their arguments are used at the same time and results are produced simultaneously, i.e., at the same time-stamp. Causality cycles are broken by the `last` construct, that does not introduce any relation between its argument and result time-stamps. Hence the following expression is causal:

$$n \texttt{ init } (-1) = \texttt{last } n + 1$$

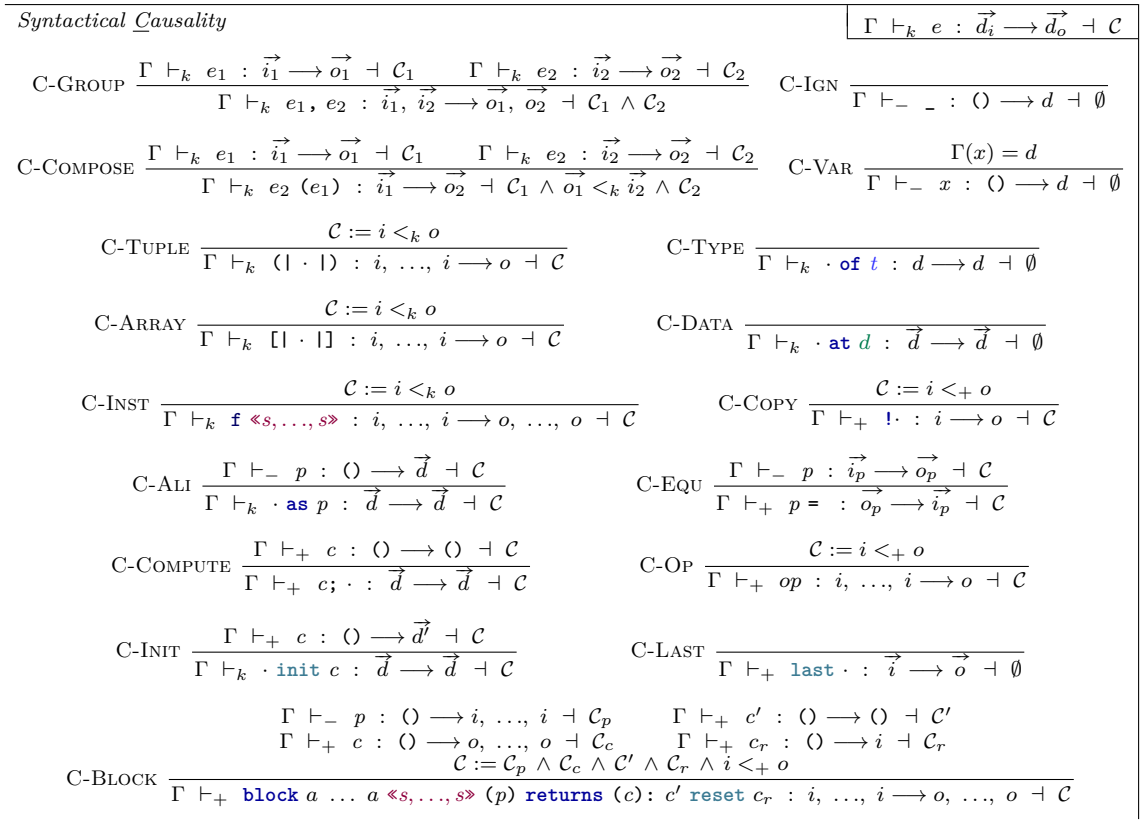


Figure 4.11: *Causality check*

4.1.5 Overview of Static Checks

During compilation, the correction of programs is established with various static checks that ensure several structural properties, e.g., initialization correctness. These verifications may be performed at two different levels, depending on whether memory locations are considered or not. Whereas location-based checks are more precise, they are based on information that is implicit in the program: the inferred allocation. The memory-less checks are thus preferable with a view to bringing the language and error diagnostics closer to source programs. Notably, if these ideas were to be used in the compilation of Scade, most of these verifications should be performed on a form that is closer to the source language.

We briefly present the correctness properties of programs alongside the means to check them statically. They are collected in Figure 4.12. The following sections and chapters explain these verifications in detail.

Constructiveness The conditions for a syntactical causality verification are presented in Section 4.1.4. The causality check ensures that the data-flow graph is acyclic. This property greatly simplifies other verifications. Hence causality is checked early.

For the two-sided semantics, the constructiveness conditions are stricter. A *scheduling* must exist, and it must be compatible with both the data-flow dependencies and memory allocation to ensure that all the uses of a value at a given location are performed before any write to the same location. Scheduling is sketched in Chapter 7.

Alignment of structural expressions Two kinds of constructs represent statically eliminated computations: groups and data-structures. These expressions do not generate any code. The correctness of groups consists in checking for a consistent number of arguments and results between connected expressions. This is ensured by computing expression arity as discussed in Section 4.1.4.

Data structure alignment is only meaningful for the two-sided semantics because it requires a notation of memory location. Checking the correctness of memory locations is much more involved. It is supported by a type system, described in Chapter 6, that guarantees that compound data and their parts will be stored at the same place.

	Functional level	Two-sided level
Constructiveness of operators	Causality	Schedulability
Group alignment	Arity check	Arity check
Data-structure alignment	—	Allocation check
Completeness of declarations	Bijectivity of lhs expressions	Surjectivity of argument locations
Uniqueness of declarations	Bijectivity of lhs expressions	Injectivity of write locations
Initialization	Syntactical inclusion	Location comparison

Figure 4.12: *Overview of the correctness properties and static checks*

Declarative properties In MADL, patterns may contain complex constructs, notably operator instantiations. Some restrictions are needed to ensure that definitions are complete and unique.

At the level of the functional semantics, these properties are ensured by only allowing bijective constructs in patterns. In particular, `view` operators may not be instantiated. As detailed below in Section 4.3, bijectivity is enforced syntactically, i.e., independently of memory locations.

At the level of the two-sided semantics, the bijectivity condition could be slightly relaxed. It suffices that all expressions in patterns are surjective to ensure completeness of definitions, and uniqueness stems from the injectivity of writes.

Initialization As for definitions, the initial values of `last` expressions must be completely and uniquely specified by some `init` constructs. Such conditions are easily enforced in the two-sided semantics where memory locations are known: (i) the location of every `last` expression must be included in the ones of a set of `init` constructs that are related to the delayed expression by structural operations and (ii) two `init` constructs must not conflict, i.e., their locations must be disjoint.

Designing sufficient conditions for the functional semantics is more involved. So as to be independent of locations, it relies on a structural approximation of memory allocation called *structural inclusion*, introduced in Section 4.3.4. This over-approximation allows a restrictive check of initialization correctness.

4.2 A Two-sided Semantics

A semantics for our language must reflect both the declarative and imperative sides. The latter suggests that values are stored at some places that may be modified while the former mainly requires that any occurrence of a variable always reduces to the single value it was defined with. This remark outlines the central component of our proposal: a memory that keeps track of data-flow dependencies. The stored values can be overwritten at any instant, but accesses are guarded. They succeed only if they are consistent with the declarative semantics, i.e., no writes have occurred since the variable was defined.

Purely declarative languages are suitable for *denotational* semantics that associate mathematical objects (e.g., a value) to terms. In the presence of recursion, these denotations are defined by *fixpoints*, i.e., solutions of an equation $x = f(x)$. Denotational semantics are either *functional*, i.e., they build the denotation in a constructive way, or *relational*, i.e., they describe a relation that selects the possible denotations, without constructing them. On the contrary, imperative languages are given *operational* semantics. Instructions modify a global store that represents the state of an abstract machine, e.g., the memory. Results are observed by reading the final state of the run.

The latter approach seems unavoidable here. Our semantics should model variables and results that are stored in a modifiable memory. However, in MADL programs, both memory allocation

and scheduling are partly unspecified. Our semantics, that applies to untransformed terms, thus explores all the possible allocations and scheduling. The apparently large non-determinism that results is an illusion. Our semantics imposes data-flow restrictions on memory accesses so that all the valid choices will produce equivalent results. Actually, these degrees of freedom are the playground for the compiler. Once the static verifications have been conducted, code generation mainly amounts to balancing memory allocation and scheduling.

To emphasize the mechanisms and intuitions of our two-sided semantics, we first consider the smallest sub-language possible. We then sketch how it could be extended in two orthogonal ways: (i) adding compound data structures such as arrays (Section 4.2.4) and (ii) supporting temporal constructions (`last`, `init`) (Section 4.2.5). For this simple core language (defined in Section 4.2.2), values (ranged over by v, ...) are taken from a domain Σ. They are handled as scalars, i.e., reads and writes are considered atomic. We assume that Σ contains a special value $\perp$ that represents an undefined value. The set of variables — $x, y, \ldots$ — is denoted $\mathcal{V}$.

4.2.1 A not so Impure Memory...

The central element of our semantics is a *declarative memory*. Beside providing a mutable storage for values, it keeps track of data-flow dependencies to ensure that (i) variables are fully and uniquely defined and (ii) only uncorrupted memory is accessed for a given variable, i.e., the memory still contains the value that was stored at the definition of the declarative variable. Unlike the usual memory models, ours uses both *memory locations* and *data-flow variables* for accesses and modifications. These operations are given a semantics only if memory is used in a declarative way, i.e., all the accesses to a variable produce the same value.

This approach, based on a data-flow aware memory, is similar to *data-flow integrity enforcement* proposed by Castro, Costa, and Harris [CCH06] for safety purposes. They designed a compile-time analysis to insert data-flow information in the program to check at runtime that memory has not been corrupted, i.e., that it has only been written by recognized instructions. Since these verifications are performed dynamically, an efficient representation of dependencies is central to limit the overhead of the instrumentation. Our proposal, however, only has a semantic purpose, and efficiency is unimportant. It focusing instead on stating consistency between an imperative implementation and the declarative program.

Our memory supplements values with a history of write operations. The memory records a list of *alias sets*, one per write. Each set contains the variables that were defined as aliases of the value stored before the next write occurred. Hence, only the most recent set of variables at each location are accessible. We keep a full list of aliases to ensure a unique definition of variables and prepare the introduction of compound data. Formally:

Definition 4.1. A *change history* $\mathcal{H} \in (\mathcal{P}(\mathcal{V}))^{(\mathbb{N})}$ is a finite list of *aliases* A, that are sets of variables. They are denoted with the standard list notations ε and $A{\cdot}\mathcal{H}$.

Definition 4.2. A *declarative memory* $\mathcal{M}$ is a mapping from a set $\mathcal{L}$ of *memory atoms* to pairs $v, \mathcal{H}$, where $v \in \Sigma$ is a value and $\mathcal{H}$ is a change history. The *uninitialized memory* $\mathcal{M}_\perp$ is the mapping $l \mapsto \perp, \varepsilon$.

Definition 4.3. The set of *declared variables* $\mathcal{FV}(\mathcal{M})$ is the set of data-flow variables that have aliased a memory location, obtained by the disjoint union:

$$\mathcal{FV}(\mathcal{M}) = \bigcupdot_{l \mapsto v, \mathcal{H} \in \mathcal{M}} \bigcupdot_{A \in \mathcal{H}} A$$

Editing the memory For scalar memories, *memory locations* are simple memory atoms. The support of compound data will introduce more complex memory locations. A declarative memory $\mathcal{M}$ is modified with a couple of operations: the *write* operation — $\mathcal{M}{\cdot}\{l \leftarrow v\}$ — associates to location l the value v and the *alias* operation — $\mathcal{M}{\cdot}\{x \mathbin{@} l\}$ — registers a new variable x to represent the value currently stored at location l. These operations are defined as follows (using functional update notations):

$$\textsc{Write}\ \frac{\mathcal{M}(l) = \_, \mathcal{H}}{\mathcal{M}{\cdot}\{l \leftarrow v\} \stackrel{def}{=} \mathcal{M}\{l \mapsto v, \emptyset{\cdot}\mathcal{H}\}} \qquad \textsc{Alias}\ \frac{\mathcal{M}(l) = v, A{\cdot}\mathcal{H} \qquad x \notin \mathcal{FV}(\mathcal{M}) \qquad v \neq \perp}{\mathcal{M}{\cdot}\{x \mathbin{@} l\} \stackrel{def}{=} \mathcal{M}\{l \mapsto v, (\{x\} \cup A){\cdot}\mathcal{H}\}}$$

In this simple setting, memory writes always succeed. They make location history grow — $\emptyset \cdot \mathcal{H}$ — by inserting an empty alias set at the front. By contrast, alias registration is partially defined. It completes only if (i) the inserted variable has not been registered yet — $x \notin \mathcal{FV}(\ \mathcal{M}\ )$ — so that its definition is *unique* and (ii) the memory contains an initialized value — $v \neq \bot$ — so that the definition is *complete*.

Example. Starting from an initial memory $\mathcal{M}_0$ with a unique location l — $\mathcal{M}_0 = \{l \mapsto \bot, \varepsilon\}$ —, let us simulate the write $x = 42$, where x is located at the unique location l. First the value is written at location l:

$$\mathcal{M}_1 := \mathcal{M}_0 \cdot \{l \ \leftarrow \ 42\}$$

The memory now contains the value 42: — $\mathcal{M}_1 = \{l \mapsto 42, \emptyset \cdot \varepsilon\}$ —, but it is inaccessible: no variables are declared at this location. The new alias x is declared with:

$$\mathcal{M}_2 := \mathcal{M}_1 \cdot \{x \ @ \ l\}$$

This operation succeeds because x is not part of $\mathcal{M}_1$ and l is associated to a non-$\bot$ value. The resulting memory — $\mathcal{M}_1 = \{l \mapsto 42, \{x\} \cdot \varepsilon\}$ — is ready for accesses to x...

Accessing memory Memory is accessed with a variable $x \in \mathcal{V}$ and a memory location $l \in \mathcal{L}$, resulting in a value $v \in \Sigma$. To emphasize the data-flow nature of memory reads, they are denoted as relations — $\mathcal{M} \ \vdash \ x \ @ \ l \ \rightsquigarrow \ v$ — where $\mathcal{M}$ is the data-flow memory, x a variable, l the location where we expect to find x and v the value that is stored in $\mathcal{M}$. It is defined with the help of an access check relation — $\mathcal{H} \ \vdash \ x$ — that ensures that the value of variable x is accessible:

Memory read	$\mathcal{M} \ \vdash \ x \ @ \ l \ \rightsquigarrow \ v$
M-Read $\dfrac{\mathcal{M}(l) = v, \mathcal{H} \qquad \mathcal{H} \ \vdash \ x}{\mathcal{M} \ \vdash \ x \ @ \ l \ \rightsquigarrow \ v}$	

Alias check	$\mathcal{H} \ \vdash \ x$
A-Current $\dfrac{x \in A}{A \cdot \mathcal{H} \ \vdash \ x}$	

The access succeeds only if the requested variable is contained in the set of *top-level* aliases for the associated location. Because the above write operation inserts an empty alias set, this ensures that no values have been written between the registration of the variable and its access.

Embedding memory locations The above declarative memory gives a *location*-based interface: each operation — writing, aliasing and reading — refers to a memory location. However, this information is unspecified in MADL programs. The semantics existentially quantifies on a correct memory allocation that associates a location to each variable.

Because these locations serve only for memory operations, we define *memory environments*, that pair declarative memories with allocations. Allocations define the places where variables are stored. Memory environments provide a *variable*-based interface for storage operations.

Definition 4.4 (Memory environment, allocation)**.** A *memory environment* $\mathcal{E}$ is a pair $(\Gamma, \mathcal{M})$ where the *allocation* $\Gamma : \mathcal{V} \to \mathcal{L}$ is a mapping from variables to memory locations and $\mathcal{M}$ is a declarative memory.

By retrieving the location of variables from the environment, the variable-based write and alias operations are defined below, using $\mathcal{E} = \Gamma, \mathcal{M}$:

$$\mathcal{E} \cdot \{x \ \leftarrow \ v\} \ \stackrel{def}{=} \ \Gamma, \mathcal{M} \cdot \{\Gamma(x) \ \leftarrow \ v\} \cdot \{x \ @ \ \Gamma(x)\} \qquad \text{(write)}$$

$$\mathcal{E} \cdot \{y \ @ \ x\} \ \stackrel{def}{=} \ \Gamma, \mathcal{M} \cdot \{y \ @ \ \Gamma(x)\} \qquad \text{(alias)}$$

In particular, values are written to *variables*. This operation first writes the value in memory and then immediately registers the variable as an alias of its location. In both operations, the location mapping is untouched. It plays the role of an oracle that specifies where variables are stored. The write operation introduces a harmless restriction. Undefined values — $\bot$ — may not be written to memory, because variables may not alias $\bot$.

Similarly, the reading relation is revisited in a variable-centeric way — $\mathcal{E} \ \vdash \ x \ \rightsquigarrow \ v$ — where the location of the variable is retrieved from the allocation and we provide a consistency check — $\mathcal{E} \ \vdash \ x$ — to ensure that a variable is *alive*, i.e., that it denotes a value that is accessible in the memory. Both relations are defined below:

Environment Read $\mathcal{E} \vdash x \rightsquigarrow v$

$$\text{E-Read}\ \frac{\mathcal{M} \vdash x\ @\ \Gamma(x) \rightsquigarrow v}{\Gamma, \mathcal{M} \vdash x \rightsquigarrow v}$$

Environment check $\mathcal{E} \vdash x$

$$\text{E-Check}\ \frac{\Gamma(x) = l \qquad \mathcal{M}(l) = v, \mathcal{H} \qquad \mathcal{H} \vdash x}{\Gamma, \mathcal{M} \vdash x}$$

Last, given a memory environment $\mathcal{E} = \Gamma, \mathcal{M}$ and a variable $x \in \mathcal{V}$, we note $\mathcal{E}(x) \stackrel{def}{=} \Gamma(x)$ the location of x.

4.2.2 ... For a not so Pure Language

The syntax of the core fragment we considered here is given in Figure 4.13. It introduces the key constructs of MADL: variables, computations and memory location constraints.

e	::=	***Expressions***
	x	variable
	$op\ (e, \ldots, e)$	operation
	e **at** x	memory constraint
	e **as** x	alias introduction
	e; e	computation

Figure 4.13: *The core fragment*

The *operation* — $op\ (e_1, \ldots, e_n)$ — represents the application of function $op : \Sigma^n \to \Sigma$, an n-ary inner operation of Σ, to arguments $e_1, \ldots, e_n$. A value $v \in \Sigma$ is represented as a 0-ary operator — $v\ ()$ — simply denoted v. Although copies deserve a special syntax — $!e$ —, they are handled as an identity computation — $id\ (e)$ —. The set of computations is the *operational part* of a term. During reduction they are the only constructs that induce changes in memory.

As in the complete language, memory allocation is indirectly specified. The location constraint — e **at** x — indicates that the value of e must be stored at the memory location of variable x, without introducing a copy. Contrary to the full MADL language, we use *variables* instead of *location annotations* to specify memory locations. This avoids having to maintain a set of location variables in our formalization.

The alias construct — e **as** x — introduces a name for the result of an expression. Here as well, the memory location of variable x must coincide with the location of the result of e. For our simple language, all variables share the same scope. Memory constraints and aliases are the *structural part* of terms: they do not modify memory but they do introduce allocation constraints.

Last, the computation — e_1; e_2 — resembles a local definition **let** e_1 **in** e_2, but two differences are worth mentioning: (i) it is unscoped, i.e., the variables introduced in e_1 by aliases — $\cdots$ **as** x — are accessible in every sub-terms of the evaluated expressions, and (ii) it does not impose any evaluation order. The fragments of e_1, e_2 and the surrounding expressions may be computed in any order.

4.2.3 Semantics

Intuitively, the semantics builds a sequence of atomic operations, whose results are stored in the memory. Since our declarative memory expects variables to perform reads, intermediate results are stored using *fresh*, i.e., unbound variables. We suppose the allocation Γ to be defined for both program variables and the extra variables that are used during evaluation.

The semantics of expressions — $\langle e \mid \mathcal{E} \rangle \rightsquigarrow \langle \mathcal{E}' \mid e' \rangle$ — reduces a term e in a memory environment $\mathcal{E}$ to a new term e' with an updated environment $\mathcal{E}'$. The symmetry of our syntax has no other purpose than improving readability in derivation trees. We denote as *evaluated expressions* the ones of the form e; x where e is an expression and x a variable. They represent expressions whose results have been computed in memory and stored at the location of variable x, while leaving some expression to evaluate. We consider that single variables x are evaluated expressions, with an empty unevaluated expression.

The deduction rules for the semantics are given in Figure 4.14. They are partitioned into two categories:

- The *reduction* rules drive the exploration of possible schedules. The rule R-Seq chains reductions together. The R-Op rule allows to reduce one of the arguments of an operator

Semantics of Expressions (1): Reduction $\langle e \mid \mathcal{E} \rangle \rightsquigarrow \langle \mathcal{E}' \mid e' \rangle$

$$\text{R-Seq}\ \frac{\langle e_1 \mid \mathcal{E}_1 \rangle \rightsquigarrow \langle \mathcal{E}_2 \mid e_2 \rangle \qquad \langle e_2 \mid \mathcal{E}_2 \rangle \rightsquigarrow \langle \mathcal{E}_3 \mid e_3 \rangle}{\langle e_1 \mid \mathcal{E}_1 \rangle \rightsquigarrow \langle \mathcal{E}_3 \mid e_3 \rangle}$$

$$\text{R-Op}\ \frac{\langle e_i \mid \mathcal{E} \rangle \rightsquigarrow \left\langle \mathcal{E}' \mid e_i' \right\rangle}{\langle op\ (e_1, \ldots, e_i, \ldots, e_n) \mid \mathcal{E} \rangle \rightsquigarrow \left\langle \mathcal{E}' \mid op\ (e_1, \ldots, e_i', \ldots, e_n) \right\rangle}$$

$$\text{R-As}\ \frac{\langle e \mid \mathcal{E} \rangle \rightsquigarrow \langle \mathcal{E}' \mid e' \rangle}{\langle e\ \texttt{as}\ x \mid \mathcal{E} \rangle \rightsquigarrow \langle \mathcal{E}' \mid e'\ \texttt{as}\ x \rangle} \qquad \text{R-At}\ \frac{\langle e \mid \mathcal{E} \rangle \rightsquigarrow \langle \mathcal{E}' \mid e' \rangle}{\langle e\ \texttt{at}\ x \mid \mathcal{E} \rangle \rightsquigarrow \langle \mathcal{E}' \mid e'\ \texttt{at}\ x \rangle}$$

$$\text{R-ComprL}\ \frac{\langle e_1 \mid \mathcal{E} \rangle \rightsquigarrow \left\langle \mathcal{E}' \mid e_1' \right\rangle}{\langle e_1\,;\ e_2 \mid \mathcal{E} \rangle \rightsquigarrow \left\langle \mathcal{E}' \mid e_1';\ e_2 \right\rangle} \qquad \text{R-ComprR}\ \frac{\langle e_2 \mid \mathcal{E} \rangle \rightsquigarrow \left\langle \mathcal{E}' \mid e_2' \right\rangle}{\langle e_1\,;\ e_2 \mid \mathcal{E} \rangle \rightsquigarrow \left\langle \mathcal{E}' \mid e_1;\ e_2' \right\rangle}$$

Semantics of Expressions (2): Evaluation $\langle e \mid \mathcal{E} \rangle \rightsquigarrow \langle \mathcal{E}' \mid e' \rangle$

$$\text{E-Op}\ \frac{\begin{array}{c}\mathcal{E} \vdash x_1 \rightsquigarrow v_1 \\ \cdots \\ \mathcal{E} \vdash x_n \rightsquigarrow v_n\end{array} \qquad v = op\ (v_1, \ldots, v_n)}{\langle op\ ((e_1\,;\ x_1), \ldots, (e_n\,;\ x_n)) \mid \mathcal{E} \rangle \rightsquigarrow \langle \mathcal{E} \cdot \{x \leftarrow v\} \mid e_1\,;\ \ldots\,;\ e_n\,;\ x \rangle}$$

$$\text{E-As}\ \frac{\mathcal{E} \vdash x \qquad \mathcal{E}(x) = \mathcal{E}(y)}{\langle (e\,;\ x)\ \texttt{as}\ y \mid \mathcal{E} \rangle \rightsquigarrow \langle \mathcal{E} \cdot \{y\ @\ x\} \mid e\,;\ x \rangle} \qquad \text{E-At}\ \frac{\mathcal{E} \vdash x \qquad \mathcal{E}(x) = \mathcal{E}(y)}{\langle (e\,;\ x)\ \texttt{at}\ y \mid \mathcal{E} \rangle \rightsquigarrow \langle \mathcal{E} \mid e\,;\ x \rangle}$$

Figure 4.14: *Semantics of the core language*

application. Likewise the other rules (R-As, R-At, R-ComprL and R-ComprR) perform reduction in any of their sub-expressions.

- The *evaluation* rules define the actual semantics of expressions. They apply to expressions whose parts have already been reduced, i.e., sub-expressions are evaluated expressions $e\,;\ x$. These rules build evaluated expressions. The rule E-Op uses a fresh variable x to store the value returned by the operator. The resulting evaluated expression collects the unevaluated parts of the arguments. The other evaluation rules only forward the result of one of their sub-expressions, they do not compute. The E-As and E-At rules force variable locations to coincide, and the former registers the right variable y as an alias of the left one x, making access to y valid until the next write to its location.

Although intermediate variables appear free in the rules, they can be defined syntactically: one variable is needed per *computation*, i.e., $op\ (e_1, \ldots, e_n)$. For a given expression, both the allocation and scheduling are chosen arbitrarily, but only a few correct guesses fulfill memory location constraints and data-flow dependencies. Moreover, all the valid ones are *equivalent*. They produce the same result, possibly at different memory locations.

Example Let us build an in-place swap. We recall that $\texttt{!}e$ denotes the application of the identity $id\ (e)$, i.e., a copy of the result of e. Let k be a continuation of two arguments. It needs both of them to be accessible simultaneously. We simulate a swap with this continuation below. The copy has the strongest priority, hence $\texttt{!}x\ \texttt{at}\ y$ reads $(\texttt{!}x)\ \texttt{at}\ y$, i.e., the copy of x at location y.

$$k\ (\texttt{!}x\ \texttt{at}\ y,\ \texttt{!!}y\ \texttt{at}\ x)$$

In an imperative language, such an operation requires copying one of the values into a temporary memory location before writing it back to its final location. For instance, a C code that swaps the values of variables `x` and `y` of type `t` looks like:

```
t z ;
z = y;
y = x;
x = z;
```

Each assignment introduces a copy. In our expression $-k\ (\texttt{!}x\ \texttt{at}\ y,\ \texttt{!!}y\ \texttt{at}\ x)-$, this is reflected by the three copies. We suppose that the free variables x and y are both accessible, at distinct memory locations, so that they may be given different values. The continuation takes the value of x stored at the location of y, and conversely. Because it contains 3 copies, this term needs

3 temporary variables z_1, z_2 and z_3 to be reduced. A wisely chosen memory allocation is given below:

$$\Gamma = \begin{cases} x,\ z_3 & \mapsto & l_x \\ y,\ z_2 & \mapsto & l_y \\ z_1 & \mapsto & l_z \end{cases}$$

Our example has to be evaluated in an initial memory environment $\mathcal{E}_0$ where its free variables x and y are readable. Because this term does not introduce any variables (the `as` constructs), the memory environment will only be modified by the copies, producing 3 environments $\mathcal{E}_1$, $\mathcal{E}_2$ and $\mathcal{E}_3$. A valid semantics derivation is given by the following scheduling, where memory environments and evaluation rule instances are detailed afterward. The premises of the bottom-most R-Seq rules appear on top of each other for space reason, but looking at environment indices should help to recover the structure of the derivation.

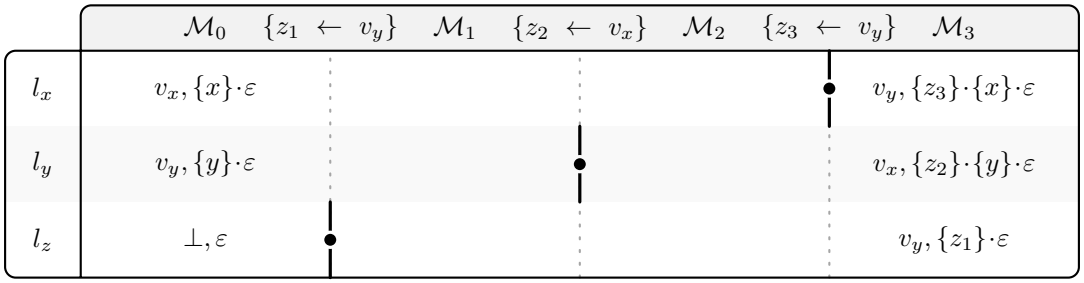
$$\textsc{R-Op}\ \frac{\textsc{R-At}\ \dfrac{\textsc{R-Op}\ \dfrac{\textsc{E-Op}\ \dfrac{\cdots}{\langle \texttt{!}y \mid \mathcal{E}_0\rangle \rightsquigarrow \langle \mathcal{E}_1 \mid z_1\rangle}\ (1)}{\langle \texttt{!!}y \mid \mathcal{E}_0\rangle \rightsquigarrow \langle \mathcal{E}_1 \mid \texttt{!}z_1\rangle}\ (i=1)}{\langle \texttt{!!}y\ \texttt{at}\ x \mid \mathcal{E}_0\rangle \rightsquigarrow \langle \mathcal{E}_1 \mid \texttt{!}z_1\ \texttt{at}\ x\rangle}}{\langle k\ (\texttt{!}x\ \texttt{at}\ y,\ \texttt{!!}y\ \texttt{at}\ x) \mid \mathcal{E}_0\rangle \rightsquigarrow \langle \mathcal{E}_1 \mid k\ (\texttt{!}x\ \texttt{at}\ y,\ \texttt{!}z_1\ \texttt{at}\ x)\rangle}\ (i=2)$$

$$\textsc{R-Op}\ \frac{\textsc{R-Seq}\ \dfrac{\textsc{R-At}\ \dfrac{\textsc{E-Op}\ \dfrac{\cdots}{\langle \texttt{!}x \mid \mathcal{E}_1\rangle \rightsquigarrow \langle \mathcal{E}_2 \mid z_2\rangle}\ (2)}{\langle \texttt{!}x\ \texttt{at}\ y \mid \mathcal{E}_1\rangle \rightsquigarrow \langle \mathcal{E}_2 \mid z_2\ \texttt{at}\ y\rangle} \qquad \textsc{E-At}\ \dfrac{\cdots}{\langle z_2\ \texttt{at}\ y \mid \mathcal{E}_2\rangle \rightsquigarrow \langle \mathcal{E}_2 \mid z_2\rangle}\ (3)}{\langle \texttt{!}x\ \texttt{at}\ y \mid \mathcal{E}_1\rangle \rightsquigarrow \langle \mathcal{E}_2 \mid z_2\rangle}}{\langle k\ (\texttt{!}x\ \texttt{at}\ y,\ \texttt{!}z_1\ \texttt{at}\ x) \mid \mathcal{E}_1\rangle \rightsquigarrow \langle \mathcal{E}_2 \mid k\ (z_2,\ \texttt{!}z_1\ \texttt{at}\ x)\rangle}\ (i=1)$$

$$\textsc{R-Op}\ \frac{\textsc{R-Seq}\ \dfrac{\textsc{R-At}\ \dfrac{\textsc{E-Op}\ \dfrac{\cdots}{\langle \texttt{!}z_1 \mid \mathcal{E}_2\rangle \rightsquigarrow \langle \mathcal{E}_3 \mid z_3\rangle}\ (4)}{\langle \texttt{!}z_1\ \texttt{at}\ x \mid \mathcal{E}_2\rangle \rightsquigarrow \langle \mathcal{E}_3 \mid z_3\ \texttt{at}\ x\rangle} \qquad \textsc{E-At}\ \dfrac{\cdots}{\langle z_3\ \texttt{at}\ x \mid \mathcal{E}_3\rangle \rightsquigarrow \langle \mathcal{E}_3 \mid z_3\rangle}\ (5)}{\langle \texttt{!}z_1\ \texttt{at}\ x \mid \mathcal{E}_2\rangle \rightsquigarrow \langle \mathcal{E}_3 \mid z_3\rangle}}{\langle k\ (z_2,\ \texttt{!}z_1\ \texttt{at}\ x) \mid \mathcal{E}_2\rangle \rightsquigarrow \langle \mathcal{E}_3 \mid k\ (z_2,\ z_3)\rangle}\ (i=2)$$

$$\textsc{R-Seq}\ \frac{\textsc{R-Seq}\ \dfrac{}{\langle k\ (\texttt{!}x\ \texttt{at}\ y,\ \texttt{!}z_1\ \texttt{at}\ x) \mid \mathcal{E}_1\rangle \rightsquigarrow \langle \mathcal{E}_3 \mid k\ (z_2,\ z_3)\rangle}}{\langle k\ (\texttt{!}x\ \texttt{at}\ y,\ \texttt{!!}y\ \texttt{at}\ x) \mid \mathcal{E}_0\rangle \rightsquigarrow \langle \mathcal{E}_3 \mid k\ (z_2,\ z_3)\rangle}$$

The four environments $\mathcal{E}_i = \Gamma, \mathcal{M}_i$ are defined in the following figure, that gives, at the same time, an execution trace. The changes are depicted by the solid vertical lines. These environments make clear the the five elided rules are indeed derivable, as shown below.



$$\textsc{C-Op}\ \frac{\mathcal{E}_0 \vdash y \rightsquigarrow v_y \qquad \mathcal{E}_1 = \mathcal{E}_0\cdot\{z_1 \leftarrow v_y\}}{\langle \texttt{!}y \mid \mathcal{E}_0\rangle \rightsquigarrow \langle \mathcal{E}_1 \mid z_1\rangle}\ (1)$$

$$\textsc{C-Op}\ \frac{\mathcal{E}_1 \vdash x \rightsquigarrow v_x \qquad \mathcal{E}_2 = \mathcal{E}_1\cdot\{z_2 \leftarrow v_x\}}{\langle \texttt{!}x \mid \mathcal{E}_1\rangle \rightsquigarrow \langle \mathcal{E}_2 \mid z_2\rangle}\ (2) \qquad \textsc{C-At}\ \frac{\mathcal{E}_2 \vdash z_2 \qquad \Gamma(z_2) = \Gamma(y)}{\langle z_2\ \texttt{at}\ y \mid \mathcal{E}_2\rangle \rightsquigarrow \langle \mathcal{E}_2 \mid z_2\rangle}\ (3)$$

$$\textsc{C-Op}\ \frac{\mathcal{E}_2 \vdash z_1 \rightsquigarrow v_y \qquad \mathcal{E}_3 = \mathcal{E}_2\cdot\{z_3 \leftarrow v_y\}}{\langle \texttt{!}z_1 \mid \mathcal{E}_2\rangle \rightsquigarrow \langle \mathcal{E}_3 \mid z_3\rangle}\ (4) \qquad \textsc{C-At}\ \frac{\mathcal{E}_3 \vdash z_3 \qquad \Gamma(z_3) = \Gamma(x)}{\langle z_3\ \texttt{at}\ x \mid \mathcal{E}_3\rangle \rightsquigarrow \langle \mathcal{E}_3 \mid z_3\rangle}\ (5)$$

From the last memory state $\mathcal{M}_3$, both z_2 and z_3 are readable, allowing for computation to continue by evaluating k.

4.2.4 Compound Data

The simplest way to add compound data to our simple language would be to extending the domain of values Σ. However, we would miss our objective: describing a fine-grain memory where compound data may be accessed and written by parts without copies. Thus our data-flow memory must be extended to handle data-structures as composite objects.

Fine grain data-flow memory Memory contains monomorphic compound data. They are described by the following type language, where n denotes immediate integers:

			Data-types
t	::=		
	$\mid$	`int, bool, ...`	scalar
	$\mid$	$[n]t$	array
	$\mid$	$t * \ldots * t$	tuple

This type language is a subset of the one used for projection functors in Chapter 3. We refer the reader to Section 3.2.2 for a precise definition of maximal projections and domains. For instance the set of maximal projections of the type $t = [2]\texttt{int} * \texttt{bool} = ([2]\texttt{int}) * \texttt{bool}$, denoted $\overline{\pi\,(t)}$, is:

$$\{\texttt{(1/2)} \cdot [k\texttt{<2}] \cdot \varepsilon \mid 0 \leq k < 2\} \cup \{\texttt{(2/2)} \cdot \varepsilon\}$$

Compound data-structures of type t are represented as dependent functions from maximal projections p of $\overline{\pi\,(t)}$ to values of type $t.p$, i.e., values of dependent type $\Pi\ p{:}\overline{\pi\,(t)}.\ t.p$. For instance, the value $v = \texttt{(| [| 4, 2 |], true |)}$ of type $t = [2]\texttt{int} * \texttt{bool}$ is encoded as the function:

$$v : \begin{cases} \texttt{(1/2)} \cdot [\texttt{0<2}] \cdot \varepsilon & \longmapsto \quad 4 \\ \texttt{(1/2)} \cdot [\texttt{1<2}] \cdot \varepsilon & \longmapsto \quad 2 \\ \texttt{(2/2)} \cdot \varepsilon & \longmapsto \quad \texttt{true} \end{cases}$$

In short, our memory is modeled as a collection of *non-aliasing*, *pure data*, *typed* memory atoms. Two memory atoms are thus disjoint and data contains no indirections. A *memory location* — $l.f$ — consists of a memory atom l alongside a *view* f that is a function from the maximal projections of a given type to the maximal projections of the type of l. For instance, the following view $f : \overline{\pi\,(t * t)} \to \overline{\pi\,([3]t)}$ maps the projections of a pair onto the first and third projections of an array of size 3:

$$v : \begin{cases} \texttt{(1/2)} \cdot \varepsilon & \longmapsto \quad [\texttt{0<3}] \cdot \varepsilon \\ \texttt{(2/2)} \cdot \varepsilon & \longmapsto \quad [\texttt{2<3}] \cdot \varepsilon \end{cases}$$

This view may be used with any memory atom of type $[3]t$. We denote by $\mathcal{F}$ the set of views. In the following, we omit the typing constraints between views and memory locations.

History and accesses To keep track of the partial modifications of compound data, the history $\mathcal{H}$ stored alongside values in memory is augmented with projection information:

$$\mathcal{H} \in \left(\mathcal{P}\left(\overline{\pi\,(t)}\right) \times (\mathcal{V} \rightharpoonup \mathcal{F})\right)^{(\mathbb{N})}$$

where t is the type of the memory atom. A (non-empty) history — $(S, A){\cdot}\mathcal{H}$ — stores for each write the set S of elements, i.e., maximal projections, that have been modified. Moreover, the alias set A is turned into a partial function — $\mathcal{V} \rightharpoonup \mathcal{F}$ — that associates to some variables the view they were defined with.

The write operation — $\mathcal{M}{\cdot}\{l.f \leftarrow v\}$ — is adjusted as follow:

$$\textsc{Write}\ \frac{\mathcal{M}(l) = v_o, \mathcal{H} \qquad f \textit{ injective} \qquad v_n = v_o\,\{f(p) \mapsto e \mid (p, e) \in v\}}{\mathcal{M}{\cdot}\{l.f \leftarrow v\} \stackrel{def}{=} \mathcal{M}\,\{l \mapsto v_n, (Im\,(f), \emptyset){\cdot}\mathcal{H}\}}$$

To write a value v at location $l.f$, the view f must be injective. The old value associated to the memory atom l is updated with all the elements of v. The image of the view and an empty alias set are added to the history.

Similarly, the alias operation — $\mathcal{M}{\cdot}\{x\ @\ l.f\}$ — now is:

$$\textsc{Alias}\ \frac{\mathcal{M}(l) = v, (S, A){\cdot}\mathcal{H} \qquad x \notin \mathcal{FV}(\ \mathcal{M}\ ) \qquad \bot \notin v(Im\,(f))}{\mathcal{M}{\cdot}\{x\ @\ l.f\} \stackrel{def}{=} \mathcal{M}\,\{l \mapsto v, (S, A\,\{x \mapsto f\}){\cdot}\mathcal{H}\}}$$

Unlike for a write, an alias may be registered for non-injective views. To succeed, the variable must be fresh and the memory must be initialized, i.e., the contained value must be different from $\bot$ for all the elements addressed by the view.

Memory read — $\mathcal{M} \vdash x \mathbin{@} l.f \rightsquigarrow v$ — and alias check — $\mathcal{H} \vdash x : f$ — are rephrased as follows:

Memory read $\mathcal{M} \vdash x \mathbin{@} l.f \rightsquigarrow v$

$$\textsc{M-Read}\ \frac{\mathcal{M}(l) = v, \mathcal{H} \qquad \mathcal{H} \vdash x : f}{\mathcal{M} \vdash x \mathbin{@} l.f \rightsquigarrow f \gg v}$$

Alias check $\mathcal{H} \vdash x : f$

$$\textsc{A-Cur}\ \frac{A(x) = f}{(S, A)\cdot\mathcal{H} \vdash x : f} \qquad \textsc{A-Old}\ \frac{\mathcal{H} \vdash x : f \qquad S \# Im(f)}{(S, A)\cdot\mathcal{H} \vdash x : f}$$

The value returned by memory reads is built by composing a location's view with the value stored in memory. The conditions for accessible variables are more complex. The variable must be associated with the right view (rule A-Cur) in an alias set that is accessible by dropping unrelated writes only (rule A-Old), i.e., whose modified elements — S — and the image of the requested view — $Im(f)$ — are disjoint.

For completeness, memory environments $\mathcal{E}$ are defined as a pair $\Gamma, \mathcal{M}$ where the allocation $\Gamma : \mathcal{V} \to \mathcal{L} \times \mathcal{F}$ associates to each variable x a memory location $l.f$. The definition of environment write, alias and read operation are strictly identical. Only the environment check — $\mathcal{E} \vdash x$ — must be adjusted:

Environment check $\mathcal{E} \vdash x$

$$\textsc{E-Check}\ \frac{\Gamma(x) = l.f \qquad \mathcal{M}(l) = v, \mathcal{H} \qquad \mathcal{H} \vdash x : f}{\Gamma, \mathcal{M} \vdash x}$$

Array operations To give an intuition of how structural operations may be given a semantics, we extend our language with a few constructs in Figure 4.15. For brevity, we consider only an array constructor and a concatenation operation.

e	::=	...	***Expressions***
	|	`[|` e_1`,` ... `,` e_n `|]`	array
	|	`concat (`e_1`,` e_2`)`	concat

Figure 4.15: *The compound data operations*

Their two-sided semantics is defined in Figure 4.16. In the evaluation rules, we omit the unevaluated part of *evaluated expressions*. As for the other expressions, reduction may occur in any sub-terms (rules R-Array and R-Concat). The evaluation rules E-Array and E-Concat are very similar. All the operands must be accessible — $\mathcal{E} \vdash x_i$ — and the rules introduce constraints between operands and result memory locations. The result variable is registered as an alias of its own location without any write, that is, the memory is untouched. The location constraints are built by composing a result view with some predefined views. In particular, the $\texttt{prj}_{1,n}$ view is the function $p' \mapsto$ `[1<`n`]` $\cdot\, p'$.

4.2.5 Stream Operations

The data-flow core of Scade enjoys a concise semantics in terms of streams. The elegance of this formalization lies in the lifting of operations to infinite streams. For our operational semantics, such abstract operations are impossible. The evaluation relation describes how to compute each value of the streams using scalar operations only. For these combinational expressions, memory is used to store temporary values of the computation *during* the synchronous cycle.

The stream operations are introduced by the `last` and `init` constructs whose syntax is recalled in Figure 4.17. In terms of transition functions, these operations represent accesses to the *state*, a part of memory that stores values *across* synchronous cycles. Accesses to the previous values of streams — `last` e — must read memory before the current values of these streams are computed. Initializations — e `init` e' — write the memory upon reset before the state is accessed. We have not yet formalized the overall evaluation strategy, in particular, we did not formalize reset. In the following, we only present how the two-sided semantics may be amended to describe transition functions.

Semantics of Expressions (1): Reduction (extension) $\langle e \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}' \mid e'\rangle$

$$\textsc{R-Array}\ \frac{\langle e_i \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}' \mid e_i'\rangle}{\langle \texttt{[|}\, e_1, \ldots, e_i, \ldots, e_n \,\texttt{|]} \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}' \mid \texttt{[|}\, e_1, \ldots, e_i', \ldots e_n \,\texttt{|]}\rangle}\ i = 1, \ldots, n$$

$$\textsc{R-Concat}\ \frac{\langle e_i \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}' \mid e_i'\rangle \qquad e'_{3-i} = e_{3-i}}{\langle \texttt{concat}\ (e_1, e_2) \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}' \mid \texttt{concat}\ (e_1', e_2')\rangle}\ i = 1, 2$$

Semantics of Expressions (2): Evaluation (extension) $\langle e \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}' \mid e'\rangle$

$$\textsc{E-Array}\ \frac{\begin{array}{l}\mathcal{E} \vdash x_1 \\ \cdots \\ \mathcal{E} \vdash x_n\end{array} \qquad \mathcal{E}(x) = l.p \qquad \begin{array}{l}\mathcal{E}(x_1) = l.\left(\texttt{prj}_{1,n} \gg p\right) \\ \cdots \\ \mathcal{E}(x_n) = l.\left(\texttt{prj}_{n,n} \gg p\right)\end{array}}{\langle \texttt{[|}\, (x_1, \ldots, x_n \,\texttt{|]} \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}\cdot\{x \mathbin{@} x\} \mid x\rangle}$$

$$\textsc{E-Concat}\ \frac{\begin{array}{l}\mathcal{E} \vdash x_1 \\ \mathcal{E} \vdash x_2\end{array} \qquad \mathcal{E}(x) = l.p \qquad \begin{array}{l}\mathcal{E}(x_1) = l.\left(\texttt{cc}_1 \gg p\right) \\ \mathcal{E}(x_n) = l.\left(\texttt{cc}_2 \gg p\right)\end{array}}{\langle \texttt{concat}\ (x_1, x_2) \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}\cdot\{x \mathbin{@} x\} \mid x\rangle}$$

Figure 4.16: *Semantics of compound data operations*

e	::=	...	***Expressions***
	|	$\texttt{last}\ e$	last
	|	$e_1\ \texttt{init}\ e_2$	init

Figure 4.17: *The compound data operations*

Structure of the evaluation The memory $\mathcal{M}$ must keep track of the values that are part of the state. To illustrate how compound data mixes with state, we extend our compound-data memory. Our proposal may be simplified to obtain a semantic for scalar and stateful operations.

The extended memory — $\mathcal{M} : l \in \mathcal{L} \to v, \mathcal{H}, S_o, S_n$ — supplements the value and history of each location with two sets of maximal projections, i.e., components, that denote the older A_o and new A_n locations that are contained in the state. If no compound data are considered, this set is either $\emptyset$ or $\{\varepsilon\}$, i.e., a flag that indicates whether the scalar location is part of the state or not.

Our proposal relies on fixed locations for the state. Any `last` and `init` expressions must be evaluated at the same memory location at each cycle. Other evaluation strategies could be possible, e.g., using double buffering for the state, but they would require a dedicated handling of initialization since `init` and `last` expressions are indirectly connected. We elaborate on this below.

Reading from the past With this augmented memory, we provide a *state read* relation — $\mathcal{M} \vdash \texttt{last} \mathbin{@} l.f \rightsquigarrow v$ — that retrieves the value of the state from memory and a *state check* relation — $\mathcal{H} \vdash \texttt{last} : f$ — that ensures that the state is readable. These judgements resemble the previously defined memory read and alias check relations:

State read $\mathcal{M} \vdash \texttt{last} \mathbin{@} l.f \rightsquigarrow v$

$$\textsc{S-Read}\ \frac{\mathcal{M}(l) = v, \mathcal{H}, S_o, S_n \qquad \mathcal{H} \vdash \texttt{last} : f \qquad Im(f) \subset S_o}{\mathcal{M} \vdash \texttt{last} \mathbin{@} l.f \rightsquigarrow f \gg v}$$

State check $\mathcal{H} \vdash \texttt{last} : f$

$$\textsc{S-Empty}\ \frac{}{\varepsilon \vdash \texttt{last} : f} \qquad \textsc{S-Disjoint}\ \frac{\mathcal{H} \vdash \texttt{last} : f \qquad S \# Im(f)}{(S, A)\cdot\mathcal{H} \vdash \texttt{last} : f}$$

To access the old state (rule S-Read), the view f must target parts of the location l that are contained in the old state (S_o). Moreover, the accessed parts must not conflict with any of the partial writes that occurred (rule S-Empty and S-Disjoint).

Writing for the future The above relations access memory independently of data-flow dependencies. In our model of state, the sets S_o, S_n do not account for aliases. This is a feature rather

than a limitation, as it allows to separate initialization from accesses to the state.

The consistency of state accesses is guaranteed in another way. The semantics imposes that the writing of the new state is unique. We define a *state alias operation* — $\mathcal{M}\cdot\{\texttt{next} \mathbin{@} l.f\}$ — that adds a memory location to the set of new state locations. Its definition strongly resembles the alias operation:

$$\textsc{Next}\ \frac{\mathcal{M}(l) = v, \mathcal{H}, S_o, S_n \qquad S_n \# \mathit{Im}(f) \qquad v(\mathit{Im}(f)) \# \{\bot\}}{\mathcal{M}\cdot\{\texttt{next} \mathbin{@} l.f\} \overset{def}{=} \mathcal{M}\{l \mapsto v, \mathcal{H}, S_o, S_n \cup \mathit{Im}(f)\}}$$

The Next rule ensures that the location $l.f$ is disjoint from the already registered state locations. This is redundantly denoted with the $S_n \# \mathit{Im}(f)$ premise and the disjoint union $S_n \cup \mathit{Im}(f)$. As for the alias operation, the registered location must not contain uninitialized values. This ensures that state reads produce well-defined values.

Stream operations The state-check relation and the state-alias operation are extended to memory environments $\mathcal{E} = \Gamma, \mathcal{M}$ as was previously done for access checks and aliases. The allocation Γ is extended to associates to each occurrence of `last` and `init` a location. The semantics of stream operations is sketched in Figure 4.18. It uses an extra construct — `next` e — to represent the write of the new state.

Semantics of Expressions (1): Reduction (extension) $\langle e \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}' \mid e'\rangle$

$$\textsc{R-Last}\ \frac{\langle e \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}' \mid e'\rangle}{\langle \texttt{last}\ e \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}' \mid \texttt{last}\ e'\rangle} \qquad \textsc{R-Next}\ \frac{\langle e \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}' \mid e'\rangle}{\langle \texttt{next}\ e \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}' \mid \texttt{next}\ e'\rangle}$$

$$\textsc{R-Init-0}\ \frac{\langle e_i \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}' \mid e_i'\rangle}{\langle e\ \texttt{init}\ e_i \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}' \mid e\ \texttt{init}\ e_i'\rangle} \qquad \textsc{R-Init-1}\ \frac{\langle e \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}' \mid e'\rangle}{\langle e\ \texttt{init}\ e_i \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}' \mid e'\ \texttt{init}\ e_i\rangle}$$

Semantics of Expressions (2): Evaluation (extension) $\langle e \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}' \mid e'\rangle$

$$\textsc{E-Last}\ \frac{\mathcal{E} \vdash \texttt{last}\ lx \qquad \mathcal{E}(lx) = \mathcal{E}(\texttt{last}\ e)}{\langle \texttt{last}\ e \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}\cdot\{lx \mathbin{@} lx\} \mid \texttt{next}\ (e\ \texttt{at}\ lx);\ lx\rangle} \qquad \textsc{E-Next}\ \frac{\mathcal{E} \vdash x}{\langle \texttt{next}\ x \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}\cdot\{\texttt{next} \mathbin{@} x\} \mid x\rangle}$$

$$\textsc{E-Init-0}\ \frac{\mathcal{E} \vdash x \qquad \mathcal{E}(x) = \mathcal{E}(e\ \texttt{init}\ x)}{\langle e\ \texttt{init}\ x \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E}\cdot\{\texttt{next} \mathbin{@} x\} \mid x\rangle} \qquad \textsc{E-Init-n}\ \frac{\mathcal{E} \vdash x \qquad \mathcal{E}(x) = \mathcal{E}(x\ \texttt{init}\ e_i)}{\langle x\ \texttt{init}\ e_i \mid \mathcal{E}\rangle \rightsquigarrow \langle \mathcal{E} \mid x\rangle}$$

Figure 4.18: *Semantics of stream operations*

The evaluation for the `last` construct proceeds in two steps. First, the rule E-Last introduces a variable lx at fixed location given by the allocation — $\mathcal{E}(\texttt{last}\ e)$ —. This location must be part of the state — $\mathcal{E} \vdash \texttt{last}\ lx$ — and it is registered in memory — $\mathcal{E}\cdot\{lx \mathbin{@} lx\}$ —. The expression e does not need to be evaluated. The produced expression — $\texttt{next}\ (e\ \texttt{at}\ x);\ x$ — is an evaluated expression whose unevaluated part will compute the value of e and register it as the new state. The memory constraint — $e\ \texttt{at}\ x$ — ensures that this new value will be stored at the same memory location as the old one. Second, the evaluation of the `next` construct (rule E-Next) registers the location of the result in the new state. The rules for initialization partially define the semantics because they depend on the overall evaluation strategy. The E-Init-0 rule should be used during initialization and upon reset. It is similar to the E-Next rule. During cycle evaluation, `init` constructs are reduced with the E-Init-n rule, that only enforces the computed value to be stored at a fixed location ($\mathcal{E}(x\ \texttt{init}\ e_i)$). This guarantees that the flow, which is part of the state, is computed at the same location.

4.3 A Functional Look at MADL

The two-sided semantics gives an insight into the compilation scheme. Flows are given well-chosen memory locations so that structural constructs do not need any computations. These constructs include array constructions, views, accesses to the state, variable declarations, *etc.* The semantics highlights some of the conditions, schedulability left aside, that allocation must fulfill:

- *Injective writes.* The write operation (Write) requires injective locations, i.e., all components are mapped to distinct parts of the memory atom (see Section 3.4.2). This allows to consider each write as atomic. Hence scheduling is defined at the level of operations

and writes, even if their implementations induce multiple instructions, e.g., the copy of a compound data structure.

- *Initialized state.* Accesses to the state (S-READ) must refer to locations that have been initialized. This amounts to checking the inclusion of the image of `last` locations in the union of images of `init` locations. The uniqueness of initialization will rise from injective write and schedulability.

These properties may be statically ensured with the allocation type system that is presented in Chapter 6. However, we claim that MADL could be given a functional semantics, that is independent of memory allocation. To this end, the correctness of definitions and state must be checked at a more syntactical level.

4.3.1 Explicit State

In order to control memory precisely, the `last` and `init` constructs are *structural*: the memory location of `last` x is the memory location of x. This no-copy access to the state is all the more crucial since state is a natural candidate for in-place updates. For that reason, the stream interpretation of the delay operator of SCADE — `->` — is not suitable for MADL. The language adopts a state-based approach for the temporal operation.

Operators as stream functions Synchronous programming languages such as LUSTRE [Hal+91] or SCADE [CPP17] are based on streams. Expressions represent infinite streams and scalar operations are lifted point-to-point. Two constructs introduce a stream-dependent semantics: (i) the `pre` operator defines an uninitialized delay and (ii) the `->` one initializes a stream. Colaço and Pouzet [CP02] propose a dedicated analysis that ensures that `pre` and `->` operators are consistently used so that the values taken by the outputs of a program never depend on uninitialized values.

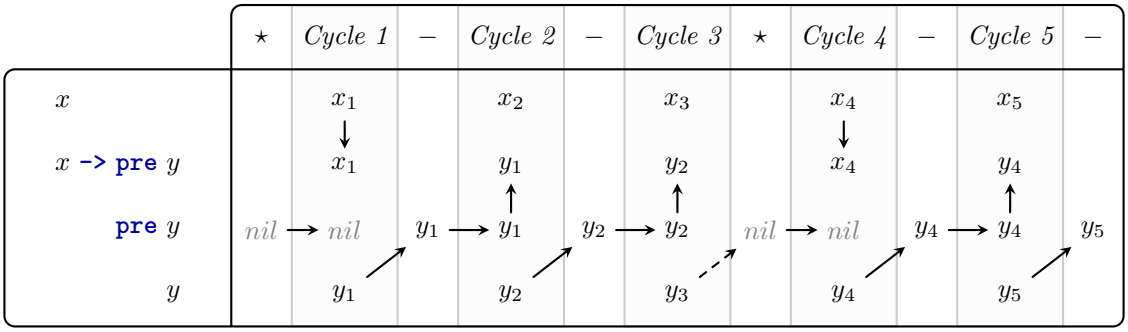


Figure 4.19: *The sequential operators of LUSTRE.*

Figure 4.19 illustrates a typical LUSTRE expression — x `->` `pre` y — that defines an initialized delayed stream. Here, both x and y are streams even though the value of x is only used at the first cycle and upon reset, which are represented by the '⋆' between cycles.

To implement such a stream function as a transition function, a state is extracted. It is represented by the inter-cycle values. Moreover, the `->` operator must distinguish the first instant after initialization or reset, in order to select the right flow. In general, this results in a conditional in the transition function and an additional init flag. The case of constant initialization is worth a dedicated compilation as the initial value can be computed outside of the transition function, i.e., upon reset, and directly stored in the state. It avoids the conditional and the initialization mechanism.

Figure 4.20 illustrates the two situations by showing the code generated by KCG, the code generator for SCADE.[2] For general initialization (Figure 4.20a), the reset function `vReg_reset` raises the initialization flag (line 15). Depending on the value of this flag (line 3), the step function `vReg` computes the output using the left or right argument of the `->` operator. With constant initialization (Figure 4.20b), the initial value is computed in the reset function `cReg_reset` (lines 13-15). The step function `cReg` is simpler since it does not depend on whether the cycle follows a reset or not.

[2] We renamed the type specific copy function from `kcg_copy_array_int32_4` to `copy_int32_4` for space reasons.

```
node vReg (x: int32^4)
returns (z: int32^4)
  z = x -> pre x;
```

```
node cReg (x: int32^4)
returns (z: int32^4)
  z = 0^4 -> pre x;
```

```c
void vReg(inC_vReg *inC, outC_vReg *outC)
{
  if (outC->init) {
    outC->init = kcg_false;
    copy_int32_4(&outC->z, &inC->x);
  }
  else {
    copy_int32_4(&outC->z, &outC->mem_x);
  }
  copy_int32_4(&outC->mem_x, &inC->x);
}

void vReg_reset(outC_vReg *outC)
{
  outC->init = kcg_true;
}
```

(a) Variable initialization

```c
void cReg(inC_cReg *inC, outC_cReg *outC)
{
  copy_int32_4(&outC->z, &outC->mem_x);
  copy_int32_4(&outC->mem_x, &inC->x);
}



void cReg_reset(outC_cReg *outC)
{
  kcg_size idx;

  for (idx = 0; idx < 4; idx++) {
    outC->mem_x[idx] = kcg_lit_int32(0);
  }
}
```

(b) Constant initialization

Figure 4.20: *Registers and initialization in* SCADE.

Operators as transition functions The special treatment for registers initialized with constants is used in the source-to-source compilation proposed by Biernacki et al. [Bie+08]. The expression `x -> y` is rewritten as:

```
if (true -> pre false) then x else y
```

The VÉLUS compiler [Bou+21] does the same: its early normalization pass ensures that `fby`[3] operators have constant left arguments. In MADL, the `init` construct is limited to constants. By forcing constant initialization, the language ensures that no conditional is needed in the generated code. This is all the more important for the iteration construct presented in Chapter 5 because it avoids branching in loops.

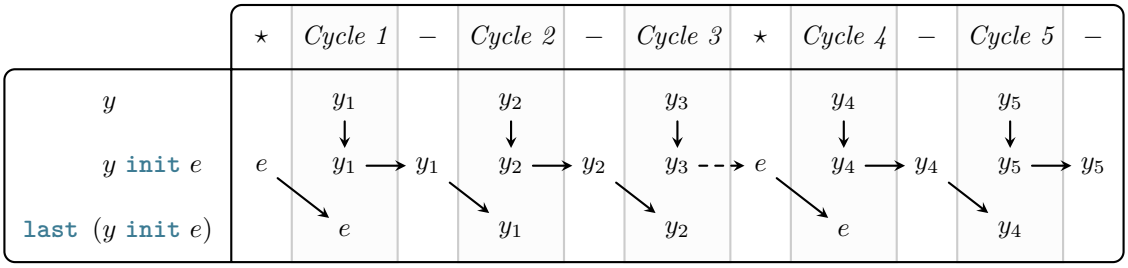


Figure 4.21: *The sequential operators of MADL.*

The MADL equivalent of a constant initialized delay — `last` (*y* `init` *e*) — is depicted in Figure 4.21. Here, the expression *e* is not part of any flow. It is only computed upon reset, i.e., between cycles. Expression *e* may contain some computation, but no variables may occur. For instance, the following counter operator is rejected:

```
1 let i_nat (i) returns (n):
2   n init !i = last n + 1
```

```
Error: Undefined variable i
```

Compilation fails during name resolution. The init expression — `!`*i* — may not use the flow *i*. The reason for a scoping issue instead of a dedicated *non-constant init* error will become clear in Section 5.1.2.

The source-to-source compilation sketched above shows how variable initialization boils down to constant initialization. To do this, a conditional construct is needed. Otherwise, operators that

[3] The `fby` operator is a contraction of the `-> pre` pattern: *x* `fby` *y* is equivalent to *x* `-> pre` *y*. Introduced in LUCID SYNCHRONE [CP96], this operator is inspired from LUCID [WA+85].

return a conventional value at the first cycle(s) may not be defined. This is the case of a discrete derivative or a rising-edge detector which are usually given value 0 at the first cycle.

```
let deriv (h, x) returns (d):
  d = (x - last (!x init 0)) / h;
```

In the above MADL implementation, the value at the first instant is x/h. Currently, there is no way to implement the zero-initialized derivative in MADL since no conditional expressions are available. It could be provided as an external operator, but we anticipate that it would benefit from dedicated handling for memory locations.

State and memory locations The stateful examples we have presented so far share a striking similarity: they all use copies. Indeed, the `last` construct is *structural*. It does not introduce any computation or data movement. As a consequence, its argument and result are stored at the same memory location. Thus an expression such as $(x - \texttt{last}\ x) / h$ leads to a scheduling issue. It can neither be computed before evaluating x, since it depends on x, nor after because the value of `last` x would no longer be accessible. We elaborate on the scheduling constraints in Chapter 7.

Copies allow to isolate the state. To give a more precise meaning to this claim, let us compare two equivalent and correct implementations of our favorite `nat` node:

```
let nat () returns (n):
  n = !last p;
  p = n + 1 init 0;
```

```
let nat' () returns (n):
  n init -1 = last n + 1;
```

Compared to the usual SCADE definition — `n = 0 -> pre n + 1` —, both versions may look convoluted. In the implementation on the left, the state is initialized with 0. Hence we must return `last` p so that the sequence of outputs starts at 0. However, the computation of p — $n + 1$ — will overwrite the value of the state, that must be copied first. In the second implementation, we reverse the step function so that it computes the desired value — 0 — at the first cycle. As explained above, such transformations are not always possible.

The second version spares a copy but its use is more constrained because the result — n — *is* the state. This has two consequences: (i) we cannot write a value at the same location as the result, as it would corrupt the state and (ii) the state must be stored at a fixed location, which will prevent aggregating the output of `nat'` to build an array (see Chapter 5).

By contrast, the returned value of the first version is independent of the state. It can be overwritten or aggregated. In this version, the variable p may be inlined, which brings the code closer to the SCADE version (except for the copy):

```
let nat () returns (n):
  n = !last (n + 1 init 0)
```

4.3.2 Hints for a Memory-agnostic Semantics

The usual semantics for synchronous languages like LUSTRE are based on streams. As streams are infinite objects, Caspi and Pouzet [CP96] proposed a *functional* definition of semantics built on a lazily evaluated representation of streams. *Relational* semantics provide an alternative to the manipulation of infinite data: these semantics define a relation between streams. This approach is used in VELUS [Bou+17; Bru20], the formally verified compiler for a LUSTRE-like language.

The definition of a stream-based semantics for MADL has not yet been completed. We review here some of the difficulties we anticipate.

Pattern evaluation In MADL, operator instantiations may appear in both patterns and computations. For instance, the following equation is legal:

```
concat «n, p» (a, b) = ...
```

For the two-sided semantics and for code generation, the `concat` operator does not introduce any computation, but only constrains the locations of a and b. Hence, computing the right-hand-side expression defines the value of a and b as well. Without memory locations, this indirect evaluation is impossible, it must be expressed in another way.

- A *functional* semantics computes the values of streams. For patterns, it must invert operations to define arguments (a and b) from the result. These bi-directional evaluation rules must be provided for (i) the structural constructs such as composition or array construction and (ii) for the built-in operators that may appear in patterns.

- A *relational* semantics eases the handling of complex patterns. The semantics of p `=` c is defined by (i) deriving relations between p and c and their sub-expressions and (ii) imposing that the results coincide.

Contrary to the relational semantics, the functional one is deterministic. The expression `fst` `(`p`)` `=` a where `fst` is the first projection of a pair, would have no functional semantics (`fst` has no inverse) but would have several relational semantics. Section 4.3.3 presents a compile-time analysis to enforce determinism of MADL programs. It consists in a structural check of operator kinds, to distinguish the bijective ones (`patt`).

Memory-based initialization For the reasons we have explained in Section 4.3.1, temporal operations are provided with explicit accesses and initializations of the state of the underlying transition function. In particular, the state is initialized with constant values. As an example, the MADL definition of a 0-initialized register follows. It corresponds to the Lustre equation `o = 0 -> pre i`.

```
let i_pre (i) returns (o):
  o = !last (!i init 0);
```

This description is at a lower level than the one expressed in Lustre. The need for copies was explained in Section 4.3.1. A memory-less semantics of the `last` construct is similar to the one of the `-> pre` operator, whose left operator (the initializer) is derived from the `init` constructs.

The `last` and `init` constructs are structural expressions, i.e., they do not introduce any copies. Hence initializations are subject to the same location constraints as initialized flows. Flows that share memory locations must share initialization as well. In other words, the `last` and `init` are memory-based, i.e., they apply to memory locations rather than flows.

The following problem arises: on which of the flows that share a same location should the initialization value be attached ? To illustrate this, consider the expression:

$$v \texttt{ init [| 1, 2, 3, 4, 5 |] = concat «2,3» (}a\texttt{, }b\texttt{)}$$

The intuition is the following: v and `concat «`n, p`» (`a`,` b`)` are *physically* the same value hence a and b are part of v. Thus they cannot be initialized independently because the initial values would be stored in conflicting locations. Following this memory-based view, a and b must be initialized by the v `init [| 1, 2, 3, 4, 5 |]` expression. If the previous values of flows a, b and v are all needed, they can be expressed in different ways, according to the design of the language.

1. *Variable-based init.* The usual style of the `pre` operator may be enforced by restraining `last` and `init` to variables. Hence, only `last` v would be legal with the above equation. The values of `last` a and `last` b must be computed, for example by writing:

   $$\texttt{concat «}n, p\texttt{» (}last\_a\texttt{, }last\_b\texttt{) = last }v$$

   This solution enjoys a simple semantics by extracting a state made of initialized variables, but it leads to code duplication.

2. *Flow-based init.* The above code duplication may be avoided by allowing to propagate initialization along flows. The results of structural constructs, such as `concat`, on initialized flows are initialized as well. Hence, by modifying the position of initialization as below, all the variables would be initialized:

   $$v \texttt{ = concat «2,3» (}a\texttt{ init [| 1, 2 |], }b\texttt{ init [| 3, 4, 5 |])}$$

   However, this example shows how the initialization values must be distributed among the parts, which may need additional code. Moreover, array accumulation, introduced in Chapter 5, benefits from a more general initialization.

3. *Memory-based init.* The most general solution consists in initializing a, b and v with the original expression:

$$v \texttt{ init [| 1, 2, 3, 4, 5 |] = concat «2, 3» (}a\texttt{, }b\texttt{)}$$

Initialization is propagated across the structural expressions regardless of the data-flow direction In the example, a and b are both upstream from v.

For a memory-less semantics, the increasing flexibility of these three options comes at the price of initialization complexity. In the variable-based option (1), the initial value of `last` x is directly derived from the mandatory x `init` c_i expression. The flow-based init (2) requires to evaluate programs starting from the `init` constructs to deduce the initial values. The bidirectional init (3) needs an extended evaluation for initialization. Section 4.3.4 presents a memory-less check of initialization correctness.

Copies Even in the memory-less semantics, copies cannot be completely ignored. They are identity *computations*. Indeed, they behave as identities during cycles but not for initialization. To highlight the importance of copies for initialization, we introduce the *Hello world!* function of synchronous languages: the node `nat` that computes the sequence of natural numbers, starting at 0:

```
1 let nat () returns (n):
2   n = !last p;
3   p init 0 = 1 + last p;
```

```
val nat: int
```

```
1 let nat () returns (n):
2   n = last !p;
3   p init 0 = 1 + last p;
```

```
Error: Uninitialized last
```

In the correct version on the left, the variable n is defined as a copy of the previous value of p, which is well-initialized by the expression p `init` 0. On the contrary, the rejected version on the right defines n as the previous value of the copy of p. Intuitively, the memory location of the copy result is independent of the one of p, thus it is uninitialized. In short, copies limit initialization propagation.

4.3.3 User-defined Patterns

The determinism of MADL stems from the restriction of instantiations in patterns to the operators of kind `patt`. Some of them are provided as built-ins, so a programmer can define custom ones.

For instance, the following operators are equivalent definitions of array interleaving, where bijective operators are moved around from the computation to the pattern. We refer the reader to Figure 3.3 for a detailed illustration of this operation.

```
let patt zip (a, b) returns (z): z = flatten «_» (transpose ([| a, b |]))
let patt zip (a, b) returns (z): split «_» (z) = transpose ([| a, b |])
let patt zip (a, b) returns (z): transpose (split «_» (z)) = [| a, b |]
```

Structural check of bijectivity We could try to check that operators are bijective by comparing the memory locations of their arguments and results. The analysis would be fragile and dependent on allocation inference. So as to guarantee the existence of a memory-less semantics before any allocation information is available, a restrictive structural check establishes that an operator is bijective if:

(i) It contains only *bijective* expressions, in particular, instantiated operators are all bijective, both in pattern and computation parts;
(ii) no values are ignored, i.e., the _ construct is not used in patterns;
(iii) all variables are used exactly once

The conjunction of the last two restrictions ensures that no arguments may be dropped, since the definitions are constructive, i.e., syntactically causal. Hence such a function may be used as a left-hand-side expression since its result fully and uniquely defines its arguments. These conditions are fulfilled for the above definitions of `zip`.[4] This pattern can now be used both in computations and patterns.

[4] Recall that the «_» syntax denotes an anonymous size variable, not an ignored value.

Contrary to a location-based analysis, our structural check would reject the following redefinition of the `split` operator since neither `sample` nor `window` are bijective operators:

```
let patt split «k» (a) returns (b):
  b = sample «k» (window «k» (b))
```
Incorrect program: non-bijective operator

Location-based check The bijective check has not yet been implemented. Currently, the compiler checks that operators used in patterns are surjective so that the variables are fully defined and that writes are injective. Hence the following functions are rejected for different reasons:

```
1 let f (a) returns (b):
2   sample <<_>> (b) = !a;
```
Error: Non surjective function in pattern

```
1 let f (a) returns (b):
2   window <<_>> (b) = !a;
```
Error: Non-injective write

4.3.4 Initialization

We sketched in Section 4.3.2 how initialization is propagated in a memory-less semantics. Roughly speaking: a flow is initialized if its initial value can be derived from some `init` constructs by evaluating only structural operations. This outlines the contours of a syntactical verification of initialization correctness.

Structural inclusion Our formalization of initialization propagation is based on *flows*. Because expressions represent transition functions rather than streams, we present our analysis in the same way than causality: expressions are described with a *flow signature* — $x_1, \ldots, x_n \longrightarrow y_1, \ldots, y_p$ — where $x_1, \ldots, x_n$ and $y_1, \ldots, y_p$ are identifiers that name flows. In the example we use term variables as flow identifiers since these variables represent single flows, i.e., expressions of arity $0 \to 1$.

We interpret these flow identifiers as sets that intuitively correspond to memory atoms (e.g., bytes) that represent the flows. Our coarse approximation of memory allocation called *structural inclusion*, consists in inclusion relations between these named sets. In practice, our analysis extracts inclusion relations between one flow x and a bunch of other flows $x_1, \ldots, x_n$, either with disjoint union — $x \subseteq x_1 \uplus \ldots \uplus x_n$ — or normal union — $x \subseteq x_1 \cup \ldots \cup x_n$ —. For instance, the following structural inclusions holds for the expression u `= [| a, b |]`:

$$\begin{cases} u \subseteq a \uplus b \\ a \subseteq u \\ b \subseteq u \end{cases}$$

For an expression e in an environment Γ that associates to each variable a flow identifier, the *inclusion signature* judgment — $\Gamma \vdash e : \overrightarrow{x_i} \longrightarrow \overrightarrow{x_o} \dashv \mathcal{I}$ — names the flows of the interface of the expression and extracts the set of structural inclusions $\mathcal{I}$. It is defined in Figure 4.22. In the rules, we denote with *syntactical equality* — $x_1 \uplus \ldots \uplus x_n = y_1 \uplus \ldots \uplus y_p$ — the conjunction of inclusion relations.

$$\begin{cases} x_1 \subseteq y_1 \uplus \ldots \uplus y_p \\ \ldots \\ x_1 \subseteq y_1 \uplus \ldots \uplus y_p \end{cases} \wedge \begin{cases} y_1 \subseteq x_1 \uplus \ldots \uplus x_n \\ \ldots \\ y_p \subseteq x_1 \uplus \ldots \uplus x_n \end{cases}$$

As a summary of the rules, syntactical equivalence relates flows across *structural* expressions: the I-Group rule gathers the interfaces and inclusion; the I-Compose makes connected flows match by substituting in $\mathcal{I}_2$ inclusions the input of e_2 with the output of e_1; I-Tuple and I-Array rules insert equivalences between the compound data and their parts; annotations rules (I-Type and I-Data, alias (I-Alias), compute (I-Compute) and init (I-Init) simply transmit their flows.

Computational expressions (rules I-Copy and I-Op) do not introduce any inclusions. Neither does the `last` construct, even though its argument and result must be stored at the same location. The reason is that the old and new values are different.

The inclusion signature of operators is given by an environment I. At instantiation (rule I-Inst), it is instantiated with fresh variables. The inclusion signature of builtin operators is given

Inclusion signature $\quad \Gamma \vdash e : \vec{x} \longrightarrow \vec{x} \dashv \mathcal{I}$

$$\text{I-Group}\ \frac{\Gamma \vdash e_1 : \vec{x_1} \longrightarrow \vec{y_1} \dashv \mathcal{I}_1 \qquad \Gamma \vdash e_2 : \vec{x_2} \longrightarrow \vec{y_2} \dashv \mathcal{I}_2}{\Gamma \vdash e_1, e_2 : \vec{x_1}, \vec{x_2} \longrightarrow \vec{y_1}, \vec{y_2} \dashv \mathcal{I}_1 \wedge \mathcal{I}_2} \qquad \text{I-Ign}\ \frac{}{\Gamma \vdash \_ : () \longrightarrow x \dashv \emptyset}$$

$$\text{I-Compose}\ \frac{\Gamma \vdash e_1 : \vec{x_1} \longrightarrow \vec{y_1} \dashv \mathcal{I}_1 \qquad \Gamma \vdash e_2 : \vec{x_2} \longrightarrow \vec{y_2} \dashv \mathcal{I}_2}{\Gamma \vdash e_2\ (e_1) : \vec{x_1} \longrightarrow \vec{y_2} \dashv \mathcal{I}_1 \wedge \mathcal{I}_2\{\vec{y_1}/\vec{x_2}\}} \qquad \text{I-Var}\ \frac{\Gamma(x) = y}{\Gamma \vdash x : () \longrightarrow y \dashv \emptyset}$$

$$\text{I-Tuple}\ \frac{\mathcal{I} := x \ = \ x_1 \uplus \ldots \uplus x_n}{\Gamma \vdash (\lvert \cdot \rvert) : x_1, \ldots, x_n \longrightarrow x \dashv \mathcal{I}} \qquad \text{I-Type}\ \frac{}{\Gamma \vdash \cdot\ \texttt{of}\ t : x \longrightarrow x \dashv \emptyset}$$

$$\text{I-Array}\ \frac{\mathcal{I} := x \ = \ x_1 \uplus \ldots \uplus x_n}{\Gamma \vdash [\lvert \cdot \rvert] : x_1, \ldots, x_n \longrightarrow x \dashv \mathcal{I}} \qquad \text{I-Data}\ \frac{}{\Gamma \vdash \cdot\ \texttt{at}\ d : \vec{x} \longrightarrow \vec{x} \dashv \emptyset}$$

$$\text{I-Inst}\ \frac{I(\mathtt{f}) = \vec{x} \longrightarrow \vec{y} \mid \mathcal{I}}{\Gamma \vdash \mathtt{f}\ \text{«}s, \ldots, s\text{»} : \vec{x'} \longrightarrow \vec{y'} \dashv \mathcal{I}_2\{\vec{x'}/\vec{x}\}\{\vec{y'}/\vec{y}\}} \qquad \text{I-Copy}\ \frac{}{\Gamma \vdash \texttt{!}\cdot : x \longrightarrow y \dashv \emptyset}$$

$$\text{I-Alias}\ \frac{\Gamma \vdash p : () \longrightarrow \vec{x} \dashv \mathcal{I}}{\Gamma \vdash \cdot\ \texttt{as}\ p : \vec{x} \longrightarrow \vec{x} \dashv \mathcal{I}} \qquad \text{I-Equ}\ \frac{\Gamma \vdash p : \vec{x} \longrightarrow \vec{y} \dashv \mathcal{I}}{\Gamma \vdash p\ \texttt{=} : \vec{y} \longrightarrow \vec{x} \dashv \mathcal{I}}$$

$$\text{I-Compute}\ \frac{\Gamma \vdash c : () \longrightarrow () \dashv \mathcal{I}}{\Gamma \vdash c; \cdot : \vec{x} \longrightarrow \vec{x} \dashv \mathcal{I}} \qquad \text{I-Op}\ \frac{}{\Gamma \vdash op : \vec{x} \longrightarrow y \dashv \emptyset}$$

$$\text{I-Init}\ \frac{}{\Gamma \vdash \cdot\ \texttt{init}\ c : \vec{d} \longrightarrow \vec{d} \dashv \emptyset} \qquad \text{I-Last}\ \frac{}{\Gamma \vdash \texttt{last}\ \cdot : \vec{x} \longrightarrow \vec{y} \dashv \emptyset}$$

$$\text{I-Block}\ \frac{\begin{array}{ll}\Gamma \vdash p : () \longrightarrow \vec{x} \dashv \mathcal{I}_p & \Gamma \vdash c' : () \longrightarrow () \dashv \mathcal{I}' \\ \Gamma \vdash c : () \longrightarrow \vec{y} \dashv \mathcal{I}_c & \Gamma \vdash c_r : () \longrightarrow x \dashv \mathcal{I}_r\end{array}}{\Gamma \vdash \texttt{block}\ a\ \ldots\ a\ \text{«}s, \ldots, s\text{»}\ (p)\ \texttt{returns}\ (c)\texttt{:}\ c'\ \texttt{reset}\ c_r : \vec{x} \longrightarrow \vec{y} \dashv \mathcal{I}_p \wedge \mathcal{I}_c \wedge \mathcal{I}' \wedge \mathcal{I}_r}$$

Figure 4.22: *Flow naming and structural inclusions extraction*

below. Bijective ones introduce syntactical equivalence, whereas non-bijective ones only provide an inclusion of the result in the argument, even for surjective operators such as `window` or `repeat`.

$$\begin{array}{rl}\texttt{concat}: & x,\ y \longrightarrow z \mid z \ = \ x \uplus y \\ \texttt{reverse}, \texttt{transpose}, \texttt{split}, \texttt{flatten}: & y \longrightarrow z \mid z \ = \ y \\ \texttt{window}, \texttt{sample}, \texttt{repeat}: & y \longrightarrow z \mid z \ \subseteq \ y\end{array}$$

Precision loss The precision of structural inclusions is limited by the structure of expression: unlike for memory locations, structural inclusion does not account for the different components of data-structures. Consider the following expression:

$$[\lvert\ c,\ d\ \rvert] = [\lvert\ a,\ b\ \rvert]$$

The above rules build the equivalences $a \uplus b \ = \ e$ and $c \uplus d \ = \ e$ where e denotes the flow that pass through the `=` sign, i.e., the array of size two. By transitivity, we have:

$$\left\{\begin{array}{l} a \ \subseteq \ e \ \subseteq \ c \uplus d \\ b \ \subseteq \ e \ \subseteq \ c \uplus d \end{array}\right. \wedge \left\{\begin{array}{l} c \ \subseteq \ e \ \subseteq \ a \uplus b \\ d \ \subseteq \ e \ \subseteq \ a \uplus b \end{array}\right.$$

But we cannot deduce the equivalence between a and c or b and d. We argue that such restrictions one profitable in practice as they allow local reasoning about the properties of expressions, instead of memory grounded relations between the variables, that are fragile if allocations change.

Disjunction and inclusion simplification Except for operator instantiation (rule I-Inst), the rules for inclusion extractions only build disjoint unions. In terms of memory locations, an inclusion $x \ \subseteq \ x_1 \uplus x_2$ means that there exists a correct allocation such that the locations of x_1 and x_2 are disjoint.[5] This property gives a starting point for detecting conflicting initializations (see below). However, disjunction is not preserved by transitivity. To see this, consider the expression:

$$[\lvert\ a,\ b,\ c\ \rvert] = t;\ u = [\lvert\ a,\ b\ \rvert];\ v = [\lvert\ b,\ c\ \rvert]$$

[5] Some allocations do not fulfill this property. For instance, the trivial allocation that maps all elements to a single location.

It is characterized by the following equivalences: $t = a \cup b \cup c \ \wedge \ u = a \cup b \ \wedge \ v = b \cup c$. By transitivity, we have $t \subseteq u \cup v$, even though u and v are obviously not disjoint. However, we must use transitivity so as to build a simple interface for operators that related only input and output flows. This part of structural inclusion needs more investigation.

Initialization correctness We prototype a syntactical check for initialization correctness that builds on structural inclusion. Our check collects the inclusion relations and marks the flows $\mathcal{F}_{\texttt{last}}$ that appear in `last` constructs and the flows $\mathcal{F}_{\texttt{init}}$ that are used in `init` constructs. The following properties are then established.

- *Completeness.* A flow used in a `last` must be included in a set of initialized flows:

$$\forall x \in \mathcal{F}_{\texttt{last}}.\ \exists y_1 \ldots y_k \in \mathcal{F}_{\texttt{init}},\ x \subseteq y_1 \cup \ldots \cup y_k$$

- *Uniqueness.* Initialized flows may not be included in other initialized flows except if they are siblings, i.e., used as decomposition of a single flow (we denote with S and T set of flow identifiers):

$$\forall x, y \in \mathcal{F}_{\texttt{init}},\ (\exists S.\ x \subseteq y \cup S) \implies (\exists z\, T.\ z \subseteq x \cup y \cup T)$$

The completeness check is sound but not complete. This property implies that the location of the `last` flow is included in the locations of the `init` ones, but the coarse approximation of locations may produce spurious errors. The uniqueness check is neither complete, for the same reason, nor sound, because location annotations may introduce sharing between variables that are unrelated in the data-flow graph, for instance (*x* `init` 0) `at` *'l*, (*y* `init` 1) `at` *'l*. These additional requirements depends intrinsically on allocation and must be checked after location inference. In our experimentation, the structural checks seem to detect the most common errors, while allowing complex initialization. Here are two trivial examples, more advanced ones will be introduced with iteration in Chapter 5.

```
1 let nat () returns (n):
2   n = last n + 1;
```

```
Error: Uninitialized last
```

```
1 let nat () returns (n):
2   n init 0 = last n + 1 init 1;
```

```
Error: Conflicting initialization
```

```
Note: Initialized also here
```

So as to illustrate the consequences of structural inclusion for initialization checking, let us again implement the `fib` function, by aggregating the state in an array of size 2. We use concrete syntax here so as to render compiler output in case of error.

```
1 let fib () returns (n):
2   [|a init 0, b init 1|] = [|c, d|];
3   n = !last a;
4   c = !last b;
5   d = n + c;
```

```
1 let fib () returns (n):
2   [|a, b|] = [|c init 0, d init 1|];
3   n = !last a;
4   c = !last b;
5   d = n + c;
```

Both definitions are accepted by the compiler. Whereas the first one is straightforward, the second one is less trivial. The initialization correctness of *a* and *b* stem from the initialization of *both* *c* and *d*. Indeed, interleaving initializations leads to errors:

```
1 let fib () returns (n):
2   [|a init 0, b|] = [|c, d init 1|];
3   n = !last a;
4   c = !last b;
5   d = n + c;
```

```
Error: Uninitialized last
```

```
Error: Conflicting initialization
```

```
Note: Initialized also here
```

```
1 let fib () returns (n):
2   [|a, b init 1|] = [|c init 0, d|];
3   n = !last a;
4   c = !last b;
5   d = n + c;
```

```
Error: Uninitialized last
```

```
Error: Conflicting initialization
```

```
Note: Initialized also here
```

For these definitions, the compiler complains twice. The use of `last` that refers to the indirectly initialized variable, e.g., *b* for the left version, is rejected because structural inclusion captures a

coarse relation, e.g., $b \subseteq c \cup d$. Thus the variable is initialized only if both c and d have initial values. Moreover, because of a similar structural inclusion, e.g., $a \subseteq c \cup d$ for the left version, initializations are considered overlapping and hence conflicting.

Conclusion

The MADL language conciliates a declarative description of programs with a specification of memory. Allocation is constrained by annotations that relate the locations of various expressions. More importantly, the language draws a crucial distinction between the *structural* and *computational* parts of programs. The former describe complex data-structures and their parts without any duplication of the underlying data, while the latter are the operations that need computation, including copying.

The structural expressions encompass two usages: (i) compound data-structures are constructed or destructed without introducing any computations and they can be viewed with another structure, e.g., a transposed matrix, without duplicating the data; (ii) the language describes low-level transition functions, where the state is explicit and subject to the same rules as other memory accesses. We propose a *two-sided* semantics that reflects this no-implicit-copy behavior by providing a guarded imperative execution. It ensures that structural expressions do not need computations while checking that memory accesses are consistent with the data-flow dependencies.

Structural constructs have a profound consequence. Several expressions, in particular variables, designate the same value or part of it, at a unique memory location. The language takes advantage of this by propagating the properties of flows, e.g., initialization, to their aliases.

MADL programs only constrain memory: the specification of allocation is indirect. For that reason, memory allocation has no semantic meaning, hence the language should be understood without memory locations, at a purely data-flow level. While this may seem to contradict our preceding remark, we think that *syntactical inclusion* gives a data-flow approximation of memory locations that is both easier to understand and precise enough to encompass the flexibility the underlying allocation offers. The study of the functional side of MADL is at an early stage. Besides giving a precise purely functional semantics, we anticipate that structural inclusion might be useful to build a coarse analysis of schedulability.

At this point, arrays are barely usable: the language must provide some way to iterate on them. Instead of the usual `map` and `fold` iterators, Chapter 5 defines a dedicated iteration construct, which takes advantage of the synchronous nature of MADL.

5 Iterating Over Arrays with Streams

Introduction

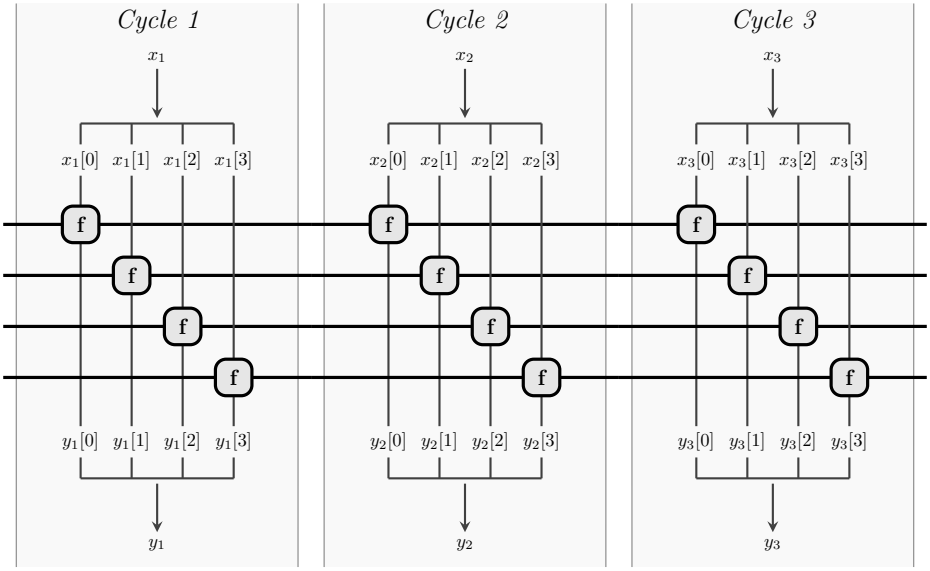
Arrays contain homogeneous data that are naturally manipulated by *iterating* computations over their elements. Such array operations may be partitioned into two classes:

- Some iterations are *unordered*: each computation is independent of the others. The iterations may be performed in any order, possibly in parallel. In functional languages, such operations are usually provided with a `map` operator.
- Other iterations are *ordered*: some information flows from one iteration to the other, thus elements must be processed sequentially. This is typically provided with `fold` primitives that fix the order of iteration (e.g., `fold_left` and `fold_right` in OCaml[1]).

Like general-purpose functional languages, the Scade language provides second-order operators that are variations of `map` and `fold` [CPP17]. These constructs need an *ad hoc* treatment since the language is first order. Among others, the Sisal language [FCO90], which stands for *Streams and Iteration in a Single Assignment Language*, proposes an alternative approach: arrays are handled as finite streams that are manipulated with dedicated first-order constructs.

The Scade language manipulates infinite streams. The approach of Sisal could benefit from the sequential constructs of Scade (**`pre`**, automata, *etc*) to iterate over arrays. The distinction between parallel and sequential iterations takes on a novel significance by viewing arrays as either *collections* of streams or as faster streams *folded* into arrays.

Arrays as collections of streams At first glance, arrays may be handled like any other data structures, e.g., tuples: they gather independent streams. In this context, iterating over arrays with transition function amounts to using different instances, i.e., states, for each element. This is represented by the following diagram.



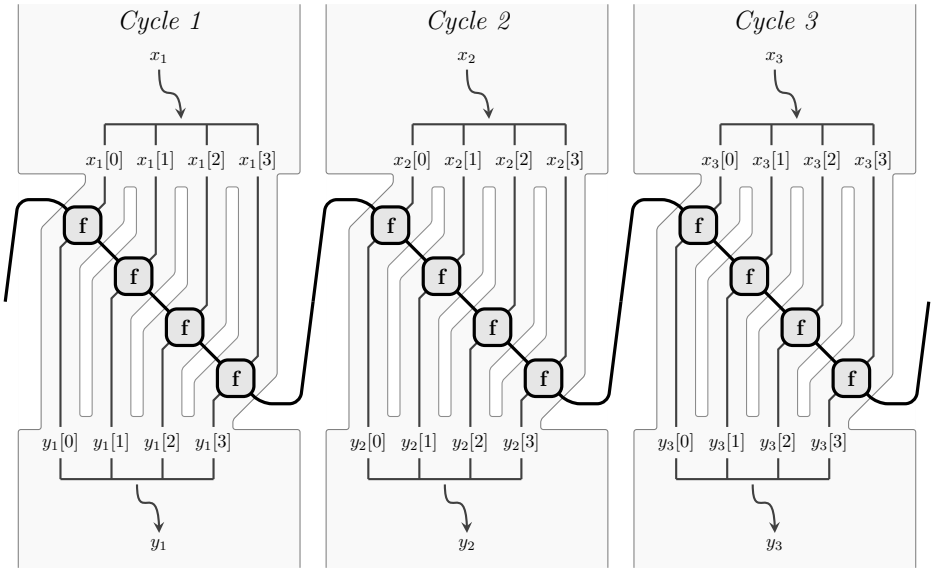
[1] https://v2.ocaml.org/api/List.html

At every cycle, the input array x of size η is decomposed in η elements that are given to the `f` transition functions. Each computation uses its own state, which is depicted by the horizontal lines. Hence, η instances of the `f` operator progress once at each cycle. Moreover, no information flows between computations in the same synchronous cycle. The results are aggregated to build the output array y, of size η as well. We call this form of iteration *multi-instantiation*.

The second-order iterators of SCADE (`map`, `fold`, *etc*) are inspired by the work of Morel [Mor05] and Maraninchi and Morel [MM04]. They have a multi-instantiation semantics: the above operation could be written `y = (map f «4») (x)`. Interestingly, the `fold` iterator combines multi-instantiation with sequential iteration. Each element is processed with its own instance, but the outputs are transmitted to distinguished inputs to allow accumulation. These iterators are well suited for block diagram representations.

In SISAL, parallel iteration is provided by the `for-loop` construct, which iterates a combinational expression over the elements of an input array. The results are either *aggregated* with the `array of` return clause to produce an array or *reduced*, for instance with the `value of sum` clause.

Arrays as folded streams In a stream-based language like SCADE, another view is possible by handling the elements of an array as the consecutive values of a *unique* stream, that comes in bundles. In this context, applying a stream function `f` to an array of size η amounts to computing η cycles of a unique instance of `f`. Each of these micro-steps uses an element of the input array and produces an element in the output array, but the state is passed from one iteration to the next. We call such iteration *multi-execution*. It is depicted below.



The flow of arrays is viewed as a faster flow made of the concatenated sequence of array elements. In the figure, these folded cycles are separated by the thin white gaps between iterations inside each synchronous cycle. The state is transmitted from one iteration to the other inside the cycle. It is a natural candidate to express accumulation.

The `for-initial` loop of SISAL provides this kind of iteration. Unlike for the `for-loop` construct, the iterated expression is sequential. It uses a state that is initialized before iterating (hence the name `for-initial`) and can be returned at the end of iteration.

This iteration scheme does not exist in SCADE. We prototyped a **`forward`** construct for the MADL language, with a view to conciliating the SISAL iteration with SCADE streams. This is the central topic of the present chapter. The idea of faster flows that unfold during synchronous cycles has been explored [Mik05; Pas13; Gua16] independently of array iterations.

Dedicated constructs or second-order iterators? General purpose functional programming languages, e.g., OCAML or HASKELL, tend to provide second-order iterators, that take the function to be iterated as an argument. Since these languages are higher order, that is, since functions are values like any other, dedicated constructs are not needed.

Since SCADE is a first-order language, higher-order constructs are confined to a separate language, namely the *operator expressions*, since the language is first order. For MADL, we argue in favor of a dedicated **`forward`** construct for three reasons:

- Accumulation is expressed using state and inherits the memory behavior of the `last` and `init` constructs. Notably, accumulation does not introduce copies, which otherwise, for fold-based accumulation, would need dedicated rules.
- Multi-dimensional iteration, which is common in data-intensive computations, can be tedious with second-order iterators, by requiring the introduction of intermediate operators to be used with `mapi` for instance. Our `forward` syntax extends naturally to iterate over multiple dimensions.
- The `forward` construct can be easily extended to recursive definitions of arrays, i.e., computing an element using already computed ones. This extends the expressiveness of the current iterators of SCADE, conciliating a declarative definition with efficiency for algorithms such as the Cholesky decomposition.

5.1 Simple Iterations

To start with, we only consider unidimensional iteration. The few syntactical additions for iteration are summarized in Figure 5.1 and detailed below. (i) The expressions are extended with *aggregation* — `[`e`]` — and *accumulation* — `{`e`}` — constructs, to be used in the interface of *forward iterators* — `forward` i`:` c `[`r`]` — that resemble local blocks. (ii) Interfaces are extended with *index capture* — `[`x `: size` x`]` — that is only legal in `forward` interfaces. (iii) The reset conditions of scopes are extended with unconditional *restarting* — `restart` — or *resuming* — `resume` — modalities. These conditions are not restricted to `forward` scope, although they are useless in non iterated scopes.

e	::=	...	***Expressions***	d	p	c
	|	`[·]`	aggregation	×	✓	✓
	|	`{·}`	accumulation	×	×	✓
	|	`forward` i`:` c `[`r`]`	forward iteration	×	×	✓
r	::=	`restart` | `reset` c | `resume`	***Reset conditions***			
i	::=	a ... a `«`$s, \ldots, s$`»` `[[`x `: size` x`]]` `(`p`)` `returns` `(`c`)`	***Interface***			

Figure 5.1: *Syntax of the MADL language: array iteration*

The interface of `forward` scopes may have at most one size parameter, that specifies the number of iterations. If omitted, the iteration size is inferred from the traversed or constructed arrays.

5.1.1 Structure and Scopes

The forward construct iterates with some expression, i.e., a transition function, over arrays. We distinguish two forms of result: (i) *aggregation* — `[·]` — builds an array from the collection of generated values, while (ii) *accumulation* — `{·}` — returns the final value of a flow that is part of the state. The examples will use the `nat` operator that we introduced in Section 4.3.1. We recall its definition:

```
let nat () returns (n):
  n = !last p;
  p = n + 1 init 0;
```

Structure of iteration Let us start with two simple examples. The first one — `iota` — builds an array of consecutive natural numbers by aggregating the results of `nat`. The second one — `dot` — implements the scalar product of two vectors. It returns the sum of the point-to-point products of the elements.

```
let iota «n» () returns (a):
  forward «n» returns (a = [i]):
    i = nat();
  reset true
```

```
let dot «n» (u, v) returns (s):
  forward «n» ([ui] = u) ([vi] = v)
  returns (s = {ls}):
    ls init 0 = last ls + ui * vi;
  reset true
```

As for local blocks in Chapter 4, these examples use the equation-like syntax for **forward** interfaces. It is more readable but gives the illusion that arguments and results are part of the scope. The following versions make scopes clear: i, ui, vi and ls are variables of the inner scopes whereas a, u, v and s are part of the operators scope.

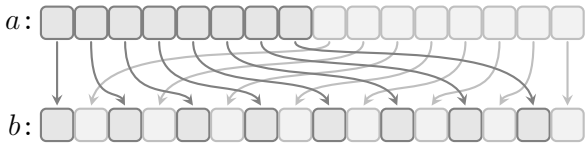
```
let iota «n» () returns (a):
  a = (forward «n» returns ([i]):
         i = nat()
       reset true)
```

```
let dot «n» (u, v) returns (s):
  s = (forward «n» ([ui], [vi])
       returns ({ls}):
         ls init 0 = last ls + ui * vi
       reset true) (u, v)
```

The size parameter — «n» — specifies the number of iterations. If omitted, it is inferred from the size of the read or written arrays. The iterator in the **iota** operator *aggregates* — [i] — the successive values of i: the result has type [n]int. The iterator in the **dot** operator returns an *accumulator* — {ls} — of type int. Its content — ls — is part of the state. In the **dot** operator, the aggregation construct is used in the arguments — [ui], [vi] — to name *current elements*. In the i^{th} iteration, the variable ui will denote the i^{th} element of array u. In contrast, the aggregation construct cannot appear in patterns.

In both cases, we specified the reset condition — **reset true** — that inserts an initialization at each synchronous cycle before starting iteration. Hence, the instantiation **iota** «4»() defines a constant stream whose value is the array [| 0, 1, 2, 3 |]. With the opposite reset condition — **reset false** —, it would produce the stream [| 0, 1, 2, 3 |], [| 4, 5, 6, 7 |], [| 8, 9, 10, 11 |], ... because the final state of **nat** at a given cycle would be the initial state for the next cycle. Likewise, the summation would run from one cycle to the next with a **false** reset condition in the **forward** iterator of **dot**.

Beside aggregation and accumulation, the **forward** interface may use any structural constructs. In particular, structural operators (**view**) may be instantiated in the interface, as the following example shows. The **shuffle** operator implements a perfect 'shuffling' of a deck of cards: it cuts the deck into two equally sized parts and interleaves them, as depicted below.




```
let shuffle <<n>> (a) returns (b):
  forward <<n>> (concat ([left_i], [right_i]) = a)
  returns (b = zip ([even_i], [odd_i])):
    even_i = !left_i;
    odd_i = !right_i;
  reset false
```

```
val shuffle: size i. type a. «i» → [2*i]a → [2*i]a
```

The **forward** interface is complex: at each iteration, it reads an element from both left — *left_i* — and right — *right_i* — parts of array a, and produces two elements *even_i* and *odd_i*. The aggregated arrays are zipped together (the definition of **zip** can be found in Section 4.3.3). We 'move' the cards by copying them from the a deck to the b one. For this operator, the reset condition makes no difference: the iterated expression is stateless.

The iteration frontier The iterated scope is separated from the enclosing one by accumulation and aggregation constructs. These expressions relate the type and memory location of the inner variables, i.e., the elements for aggregation, to the ones of outer variables, i.e., the array for aggregation.

The aggregation construct — [·] — whose arity is $1 \to 1$ has type $\tau \longrightarrow [\eta]\tau$ where τ is the type of an element and η is the number of iterations. Hence all the aggregated arrays, e.g., [*left_i*] in the **shuffle** example, have the same size, which explains the above signature. So as to support recursive definitions of arrays, this typing condition will be slightly modified in Section 5.2.1.

The typing relation for accumulation — { · } — is simpler. It has type $\tau \longrightarrow \tau$, that is, the result has the same type as the value computed in the body. Its main role is to ensure that the accumulators are part of the state.

A matter of scopes In the `forward` outputs, the aggregation and accumulation constructs connect the values produced by the inner expression to the iteration results. However, what would be the semantics of a returned value that does not pass through the `{·}` of `[·]` constructs, as in the following operator?

```
1 let get_last (a) returns (e):
2   forward ([ai] = a) returns (e = ai)
```

```
Error: Undefined variable ai
```

The above program is rejected. Before explaining how, let us explain why. The result looks similar to an accumulation: the value of e would be the value of ai at the last iteration. But e would be ill-defined in the case of iterations over empty arrays, where the iterated expression is never evaluated. That is the purpose of the `{·}` construct: it points out the returned accumulator, which must be initialized. Accumulators thus have a value even if iterations are empty. Indeed, the compiler complains for an uninitialized `last` (that should be reported more precisely) on the accumulation {ai}:

```
1 let get_last (a) returns (e):
2   forward ([ai] = a) returns (e = {ai})
```

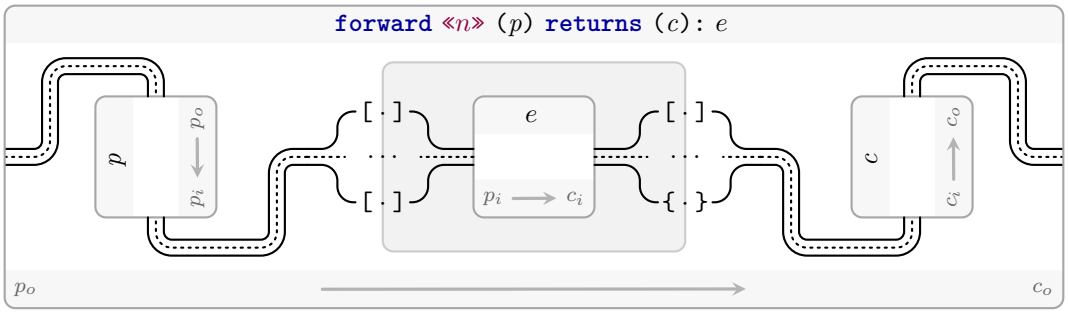
```
Error: Uninitialized last
```

To force returned flows to pass through accumulation or aggregation, the `forward` construct introduces two nested scopes instead of one. This is the reason for the error in the first version of `get_last`: *Undefined variable ai*. The innermost scope contains the iterated computation and the parts of the interface that are inside aggregation — `[·]` — or accumulation — `{·}` — constructs, while the outer one contains the rest of the interface. By making the implicit scope visible, our `iota` and `dot` examples are rewritten:

```
let iota «n» () returns (a):
  a = (forward «n» returns ([·]):
         block returns (i):
           i = nat()
         reset false
       reset true)
```

```
let dot «n» (u, v) returns (s):
  s = (forward «n» ([·], [·])
       returns ([·]):
         block (ui, vi)
         returns (sl):
           ls init 0 = last ls + ui * vi
         reset false
       reset true) (u, v)
```

In this form, we view the `[·]` and `{·}` constructs as wires (i.e., of arity $1 \to 1$), in a similar way as the other expressions, e.g., `!`$\cdot$ or $\cdot$ `as` p. In the `dot` operator, the `forward` argument thus has arity $2 \to 2$ and the result has arity $1 \to 1$. The iterated expression is a block whose interface is compatible with the input arities of the `forward` interface. In the `dot` operator, it is $2 \to 1$. Whatever the `forward` reset condition, the innermost block is never reset as this would initialize the expression at each iteration. The connection between the interface and the body of iteration is depicted below:



Because of the additional scope, the variables defined in the iteration, i.e., in the `block` scope, cannot be used in the `forward` scope, i.e., outside of aggregation and accumulation constructs.

The `forward` construct connects a pattern p of arity $p_i \to p_o$ and a computation c of arity $c_i \to c_o$ to an inner expression e of arity $p_i \to c_i$. The overall iteration has arity $p_o \to c_o$. The iterated part, depicted by the inner rectangle is guarded by the `[·]` and `{·}` expressions. These arity relations extend the ones of blocks, as summarized in Figure 5.2.

	Argument	Body	Pattern	Total
`block`	$0 \to p$	$0 \to 0$	$0 \to c$	$p \to c$
`forward`	$p_i \to p_o$	$p_i \to c_i$	$c_i \to c_o$	$p_o \to c_o$

Figure 5.2: *Comparison of arity constraints between* `block` *and* `forward`

To illustrate how the input and output arities of arguments and results may differ, we rephrase our deck shuffling example by making the inner block explicit. Here, both argument and result have arity $1 \to 2$, so the iterated block has arity $2 \to 2$ while the overall iteration has arity $1 \to 1$:

```
let shuffle «n» (a) returns (b):
  forward «n» (concat ([·], [·]) = a) returns (b = zip ([·], [·])):
    block (left_i, right_i) returns (even_i, odd_i):
      even_i = !left_i;
      odd_i  = !right_i;
    reset false
```

Constant arrays Let us introduce a convenient notation for constant arrays — `[`n`]`e — that corresponds to the expression `forward «`n`» returns ([`v`]):` v `=` e. If the expression e has some state, this iteration does not build a constant array: as for the `iota` example, it produces the n first values of the stream defined by e. This would be a typical use-case for multi-instantiation, that has not been studied yet. In the coming examples, constant arrays are only used with immediate integers, i.e., stateless expressions, where the semantics of multi-instantiation and multi-execution coincide.

Chapter 3 introduced a `repeat` projection function that allows to view a single value as a constant array. Contrary to iteration (`[_]`e), the expression `repeat «_»` (`e`) does not duplicate any value: it maps all the elements to the same memory location: it cannot initialize an array used as an accumulator.

This constant syntax is not yet available in our prototype because our AST is in a transitional state: It still separates equations, blocks and iterations from the other expressions. Hence, in the concrete syntax examples, we replace an expression `[`s`]`e by a variable u and an equation like the iteration `forward «`s`» returns (`u `= [`ui`]):` ui `=` e.

5.1.2 Restarting Nodes

Using state for accumulation is handy for at least two reasons. (i) It avoids using predefined iteration schemes, e.g., `fold` that may be contrived, e.g., by inserting unneeded results to use local accumulators. (ii) It benefits from the memory properties of sequential constructs — `last` and `init`— in particular, accumulators do not introduce copies. However, we may wonder: *is the above* `dot` *operator stateful?* Semantically, the answer is no. The state is reset at each synchronous cycle, before iteration. But syntactically, the definition has a state. This situation is depicted in Figure 5.3.

The state is transmitted from one cycle to the next even though it is reset before the first iteration at each instant (depicted by the $\star$). This has two undesirable consequences. (i) The `dot` operator may not be instantiated in a stateless operator (`func`). (ii) The output has a fixed location because it is part of the state. Thus, the results of `dot` operator cannot be accumulated and the following implementation of matrix-vector multiplication is incorrect:[2]

```
let mat_vec (a, u) returns (v):
  forward ([ai] = a) returns (v = [vi]):
    vi = dot (ai, u)                    Incorrect program:
  reset true                            Initialization of aggregated elements
```

[2] The verification is handled by the initialization check, that does not yet support instantiations. This is the reason for using the abstract syntax of MADL here.

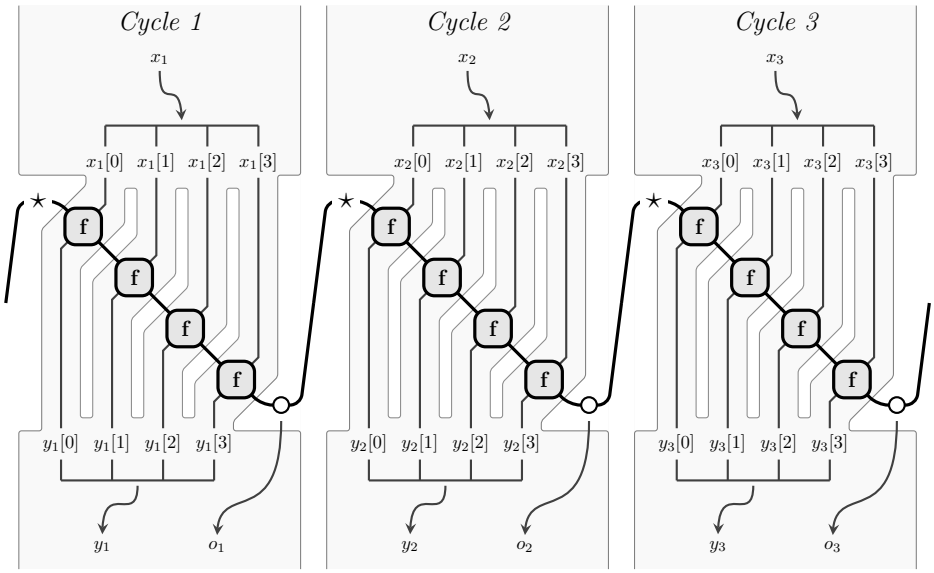


Figure 5.3: *Accumulation with a* `reset true` *reset condition*

This program requires to build an array whose elements are all stored at the same location, since the result of `dot` is part of the state. This is impossible. To circumvent this error, the result of `dot` must be copied. This allows a correct implementation of `mat_vec` at the price of an extra memory footprint and more computations:

```
let mat_vec (a, u) returns (v):
  forward ([ai] = a) returns (v = [vi]):
    vi = !(dot (ai, u))
  reset true
```

We emphasize the impact of copies in Section 4.3.1. They allow to dissociate the state from the output locations. In the previous implementations of the `iota` operator, we were careful to use the `nat` operator instead of the `nat'` one, whose output is the state. Indeed, using this second version would lead to the same issue as for the `mat_vec` operator: a copy is needed. To show the error, we inlined the definition of `nat'`.

```
1 let iota <<n>> () returns (a):
2   forward <<n>> returns (a = [i]):
3     i init (-1) = last i + 1
```

```
Error: Initialization of aggregated element
```

```
Note: Aggregated here
```

Local state The `dot` operator should be stateless. Since the iteration is reset at each cycle, the internal state need no longer be transmitted from one cycle to the next and can instead be allocated as a local value. To express this, we introduce two additional reset conditions (r) whose syntax is given in Figure 5.4.

$$r \quad ::= \quad \texttt{restart} \mid \texttt{reset}\ c \mid \texttt{resume} \qquad \textit{Reset conditions}$$

Figure 5.4: *Extended reset conditions*

The `restart` condition indicates an *unconditional reset*. The state is initialized at each synchronous cycle. As depicted in Figure 5.5, restarting blocks or iterators allows to disconnect the state used in each cycle (represented by the thick line). Symmetrically, the `resume` condition indicates that the scope is never reset. These conditions are distinguished from `reset true` and `reset false` because they are handled specifically.

These extended reset conditions impact the language and its semantics in two ways:

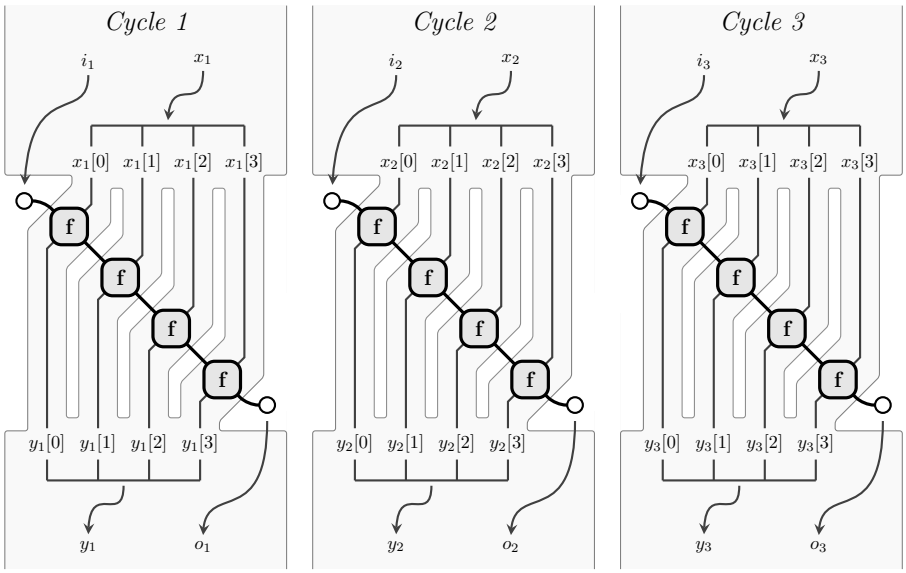


Figure 5.5: *Accumulation with a* `restart` *reset condition*

- For a scope that is *always* reset — `restart` —, the state need not be stored from one cycle to the next, since its initial value is computed at each instant. Hence, the construct (`block` and `forward`) is *combinational*. Moreover, since the initialization is computed at each synchronous cycle, it is now a flow, as depicted in Figure 5.5.
- For a scope that is *never* reset — `resume` —, the initial value of the state may be specified out of the scope, because it is not needed for reset. Section 5.1.4 discusses a direct consequence of this principle.

Scopes with an arbitrary reset condition — `reset` c — cannot benefit from either of these possibilities. They must be fully initialized and their state must be transmitted from one cycle to the next. By specifying a `restart` for the dot function, the matrix-vector multiplication can now be defined without copies. Because `mat_vec` is combinational as well, the `forward` iterator of its body is also restarted.

```
let func dot (u, v) returns (s):
  forward ([ui] = u) ([vi] = v) returns (s = {si}):
    si init 0 = last si + ui * vi;
  restart

let func mat_vec (a, u) returns (v):
  forward ([ai] = a) returns (v = [vi]):
    vi = dot (ai, u);
  restart
```

Default conditions The MADL language provides default reset conditions. If unspecified, blocks (either local or function bodies) are *resumed*, whereas `forward` iterators are *restarted*. As a final version of our examples, we may now drop the `reset true` conditions, effectively inserting a `restart`. Both operators are now stateless (`func`):

```
let func iota «n» () returns (a):
  forward «n» returns (a = [i]):
    i = nat()
```

```
let func dot «n» (u, v) returns (s):
  forward «n» ([ui] = u) ([vi] = v)
  returns (s = {ls}):
    ls init 0 = ls + ui * vi
```

Initialization To summarize the effect of extended reset conditions on initialization, we implement our favorite node `nat` by pushing one of the two equations into a separate block. The following table present variations on two orthogonal choices: (i) the reset condition and (i) the place where init is inserted, either on the inner variable (q) or on the outer one (p).

	Inner initialization	Outer initialization

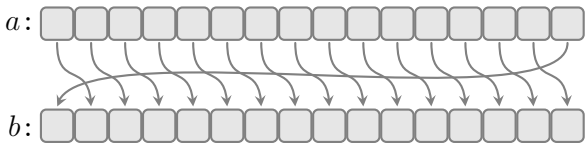
resume	`let nat () returns (n):`
	`  n = !last p;`
	`  block (p = q):`
	`  q init 0 = last q + 1`
	`  resume`
	`val nat: int`
		`let nat () returns (n):`
		`  n = !last p;`
		`  block (p init 0 = q):`
		`  q = last q + 1`
		`  resume`
		`val nat: int`

reset	`let nat (c) returns (n):`
	`  n = !last p;`
	`  block (p = q):`
	`  q init 0 = last q + 1`
	`  reset c`
	`val nat: bool → int`
		`let nat (c) returns (n):`
		`  n = !last p;`
		`  block (p init 0 = q):`
		`  q = last q + 1`
		`  reset c`
		`Error: Uninitialized last`

restart	`let nat () returns (n):`
	`  n = !last p;`
	`  block (p = q):`
	`  q init 0 = last q + 1`
	`  restart`
	`Error: Uninitialized last`
		`let nat () returns (n):`
		`  n = !last p;`
		`  block (p init 0 = q):`
		`  q = last q + 1`
		`  restart`
		`Error: Uninitialized last`

The left column shows that the state is shared unless the scope is restarted (`restart`). The right column illustrates that the inner scope must contain the `init` construct if it may be reset (`reset` or `resume`).

Example (rotation) With local state and the relaxed initialization condition, we can illustrate other uses of stateful expressions in iteration. The following operation defines an in-place array rotation, as depicted below.




```
let rotate_right «n» (a at 'l) returns (b at 'l):
  concat (_, [| l |]) = a;
  forward «n» ([ai] = a) returns (b = [bi]):
    bi = !last (!ai init !l)
```

The first equation — `concat (_, [| l |]) = a` — allows to extract the last element of the array a. The content of the `forward` iterator resemble the `pre` operator discussed in Section 4.3.2. It is a unit delay whose previous and next values are copied so that the location of the state is independent of the location of these values. Because `forward` scopes are restarted by default, the initialization may use variable l. The value must be copied too. Otherwise, i.e., `!`ai `init` l, the state would be located at the place of the last element of a, hence it would conflict with the last read of ai.

Example (insertion) The `insert_max` operator defined below adds a value v to an array whose elements are supposed to be sorted in decreasing order. To keep the size constant, the minimal value is eliminated. Hence `insert_max «4» ([| 6, 4, 2, 1 |], 3)` returns `[| 6, 4, 3, 2 |]`.

```
let insert_max «n» (v, a) returns (b):
  forward «n» ([ai] = a) returns (b = [bi]):
    ni = min (last ni, ai) init v;
    bi = max (last ni, ai);
```

Here as well, the ni variable is a register that contains the minimal value that has been encountered up to the current index. No copies are necessary because `min` and `max` are computations:

they write their result in an arbitrary location. This operator may be used to maintain the n maximal values of a flow.

Example (exponentiation) The `forward` iterator allows to encode imperative algorithms in an easier way than second-order iterators do. To illustrate this, we implement an operator `exp` that computes x^e, where e is a positive integer, using the *exponentiation by squaring* method. It relies on the binary decomposition of the exponent e and the following observation:

$$x^e = \prod \left(x^{2^i}\right)^{e_i} \quad \text{where} \quad e = \sum e_i 2^i$$

This identity induces an iterative algorithm by using the equality $q\,p^n = q\,p^{n\%2}\,(p^2)^{n/2}$ where % and / denote integer modulo and division. It is turned into the following streams:

$$\begin{cases} q_{i+1} = q_i\, p_i^{n_i\%2} \\ p_{i+1} = p_i^2 \\ n_{i+1} = n_i/2 \end{cases} \quad \text{with} \quad \begin{cases} q_0 = 1 \\ p_0 = x \\ n_0 = e \end{cases}$$

The following invariants immediately hold for all i: $n_i = e/2^i$ and $x^e = q_i\, p_i^{n_i}$. If the exponent is represented by a k-bit value, i.e., $\log_2(n) \leq k$, we have $n_k = 0$. Hence $x^e = q_k$. A MADL definition follows, assuming a branching construct — `if` e `then` e `else` e — is available.

```
let exp «k» (x, e) returns (r):
  forward «k» returns (r = {q} init 1):
    p = last (p * p init x);
    n = last (n / 2 init e);
    q = last (if n % 2 = 1 then q * p else q);
```
Yet unsupported

The local accumulators — p and n — do not appear in the interface of the iterator. In contrast, a second-order-based definition would have the following form assuming a SCADE-like `fold` primitive:

```
let exp_i (p, n, q) returns (p', n', q'):
  p' = p * p;
  n' = n / 2;
  q' = if n % 2 = 1 then q * p else q;

let exp «k» (x, e) returns (r):
  _, _, r = (fold exp_i «k») (x, e, 1)
```
Yet unsupported

In the `fold` version, all the accumulators — p, q and n — must be passed as inputs and outputs of the iterated function. The local ones are then ignored, which is error prone and less readable.

5.1.3 Capturing Indexes

General-purpose functional languages often provide some versions of iterators that give access to the index of the current iteration, e.g., `mapi` in OCAML. In MADL, this index cannot be used to access arrays since the language does not provide such a construct.[3] Indexes are rarely needed because arrays are traversed by naming their elements, i.e., using aggregation in the arguments of iterators. However, we found two use-cases that need to capture indexes:

- The iterated computation may depend on the value of the index. For instance, the matrix of *twiddle factors* used in the Fast Fourier Transforms is defined by $M_{jk} = e^{2ijk\pi/n}$.
- Some algorithms, such as the ones based on the computation of a pivot, uses indexes as sizes for slices, e.g., when reading triangular matrices.

These situations are very different. They lead to two ways of capturing indexes in `forward` iterators. They are introduced by an extension of the interface defined in Figure 5.6, that may only appear in `forward` arguments. Indexes are captured either as a term variable — `[`x`]` — or as a size variable — `[size` x`]` —. If both are needed, they can be combined in a single argument — `[`x `: size` x`]` — where the two variables may be identical, since they are taken from distinct name-spaces.

[3] Explicit accesses — `x.(i)` in OCAML— have been considered, but some memory location issues remain unsolved.

$$b \quad ::= \quad a \ \ldots \ a \ \text{«}s, \ldots, s\text{»} \ [[x : \texttt{size}\ x]] \ (p) \ \texttt{returns} \ (c) \qquad \textit{Interface}$$

Figure 5.6: *Extended interface arguments*

Indexes as variables The capture as a term variable — $[x]$ — introduces a variable x in the inner scope of the iteration, i.e., the one that contains the named elements and the iterated expression. For an iteration of size η, this variable is given type $[\eta]$, namely it has an index type. For instance, the `iota` function can be defined in a simpler way:

```
let func iota «n» () returns (a):
  forward «n» [i] returns (a = [ai]):
    ai = !i;
```

The copy may be surprising and the reader may wonder if it could be omitted. The answer is no. Intuitively, we expect `iota` to write a new value — an array — somewhere in memory. Thus it has some computational content. The effective reason for rejecting the above program without the copy lies in the memory location system that is detailed in Chapter 6. The memory location of the index is *existentially quantified*: it must not escape the scope of the iterator. For this reason, we must copy it.

Indexes as sizes So as to define arrays whose size depends on the iteration, indexes can be captured as sizes — `[size x]` — that name *existentially quantified sizes*. For instance, an identity matrix, with 1 on the diagonal and 0 elsewhere, may computed as:

```
let mId «n» () returns (m of [n][n]_):
  forward [size k] returns (m = [mi]):
    mi = [k]0 ++ [| 1 |] ++ [_]0
```

The notation e `++` f is a syntactical shortcut for `concat «_,_» (`e`,` f`)` and is left associative. In operator `mId`, each row is built out of three arrays: (i) `[`k`]0` is a constant array of size k, (ii) `[| 1 |]` is an immediate array of size 1 and (iii) `[_]0` is a constant array whose size $n-k-1$ is deduced by size inference.

The size k is existentially quantified: its value changes at each cycle (0, 1, *etc*). Hence, this size must neither appear in the type of the result nor a remaining constraint. In short, it cannot escape its scope. This ensures that we cannot produce irregular, i.e., non rectangular, multidimensional arrays. For instance, the following definition of a triangular array is rejected.

```
1 let triangular <<n>> () returns (t):
2   forward <<n>> [i:size i] returns (t = [ti]):
3     forward <<i>> [j] returns (ti = [tij]):
4       tij = i+j
```

```
Error: This expression has type [`i][`j]int
but an expression of type [`i][0]int was expected
```

```
Note: Size 1 is not compatible with size 0
```

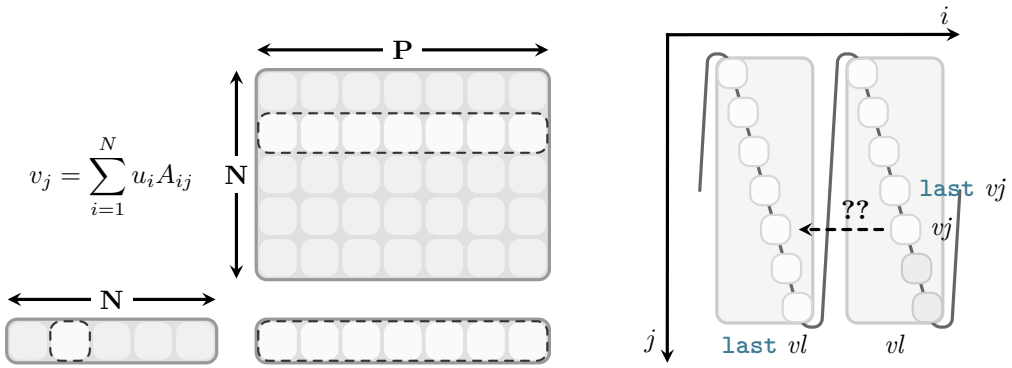
Error reports *Remark*

The error is cryptic. It complains about 1 not being equal to 0, in the type constraint `<'i> = <0>`. This comes from the strategy of resolution of size constraints, that uses abstract variables — *'i* — to decompose the polynomial constraints, as explain in Section 2.4.4. This situation needs a better diagnostic.

5.1.4 Referencing Through Multiple Dimensions

Algorithms on multidimensional arrays mix aggregation, accumulation and nested iterations. Referencing the previous value of the current element of an array used as accumulator in an outer iteration is challenging: By viewing arrays as streams, the `last` construct can only refer to the previous element of the stream, i.e., the previous element of the innermost iteration.

As an example, we define a vector-matrix multiplication in a vectorizable way (see Section 1.3.3), as depicted below. Starting from an initial vector of 0, each line of the matrix is multiplied by the corresponding element of the vector and added pointwise to the result. The inner iteration (over j), may be *vectorized*, that is, the operations can be computed in parallel.



A skeleton of a MADL version follows. It is structured as two nested iterators. The outer iteration accumulates the weighted lines into a vector. The inner one adds the weighted line to the current accumulator. For now, the reset conditions are omitted and both iterations are thus restarted. For the inner loop which computes a point-to-point sum, the reset condition makes no difference because it is state-less.

```
let func vec_mat (u, a) returns (v):
  forward ([ui] = u) ([ai] = a) returns (v = {vl}):
    forward ([aij] = ai) returns (vl = [vj]):
      vj = ?? + ui * aij;
```
Incomplete program

Two elements are missing: how shall we refer to the previous value of the element of vl, denoted by the ??, and where should we specify the initial zero-filled vector? The iteration is depicted above, where the line represents the flow of vj. Three options are possible.

1. Outer `init`, outer `last` The above diagram suggests a way to access the j^{th} element of the previous value of the accumulator. It consists in traversing the previous array — `[lvj] = last vl` — while producing the new one — `vl = [vj]`. Here, we initialized the accumulator at its use — `{vl} init [_]0` — instead of at its definition.

```
let func vec_mat (u, a) returns (v):
  forward ([ui] = u) ([ai] = a) returns (v = {vl} init [_]0):
    forward ([aij] = ai) ([lvj] = last vl) returns (vl = [vj]):
      vj = lvj + ui * aij;
```

2. Outer `init`, inner `last` For a memory-aware programmer who knows that vj are aggregated elements, it might be tempting to designate with `last vj` the last value the element had, i.e., the j^{th} element of `last vl`, instead of the value of the previous element, i.e., the $j-1^{\text{th}}$ element of v. Our `vec_mat` operator would look like:

```
let func vec_mat (u, a) returns (v):
  forward ([ui] = u) ([ai] = a) returns (v = {vl} init [_]0):
    forward ([aij] = ai) returns (vl = [vj]):
      vj = last vj + ui * aij;
    resume
```

Here, no name is needed for the elements of `last vl`, at the cost of the special semantics for the `last`. We added a `resume` reset condition. This is consistent with our relaxed initialization condition presented in Section 5.1.2. The initialization can be retrieved from the outer scope if the inner scope is resumed. This *distant* `last` does not require a dedicated compilation. As for a normal `last`, the result of `last vj` and vj are indeed at the same place.

3. Inner `init`, inner `last` Continuing further on this sliding slope, we may want to initialize the strange `last` *vj* inside the inner loop rather than initializing the array. The `vec_mat` operator would be:

```
let func vec_mat (u, a) returns (v):
  forward ([ui] = u) ([ai] = a) returns (v = {vl}):
    forward ([aij] = ai) returns (vl = [vj]):
      vj init 0 = last vj + ui * aij;          Incorrect program: Initialization of
    resume                                     aggregated elements (see Section 5.1.2)
```

The `init` construct would need a special treatment as well. It only computes an integer, whereas we would expect it to initialize the whole array. The second discrepancy is not just about semantics. If allowed, such *distant* `init`s would require a dedicated compilation because the `init` expressions that are 'attached' to aggregated flows must be duplicated. Moreover, this form conflicts with our discussion about aggregated state, presented in Section 5.1.2. It is not allowed.

Discussion We allow distant `last`s but we ban distant `init`s. Moreover, distant `last`s are guarded by the extra condition that they must be contained in resumed scopes. For instance, the second implementation of `vec_mat` would be rejected with a different reset condition. Beside aesthetic considerations, e.g., to avoid having to introduce variables for the last accumulator, we weighed up the provided flexibility against the introduced irregularity:

- Distant `last`s allows multi-dimensional iterations to be more widely applicable, by mixing aggregation and accumulation. This is illustrated in Section 5.3.
- The compilation of distant `last`s does not introduce any implementation complication. Unlike for distant `init`s, the compilation remains straightforward and comprehensible, i.e., initializations are not duplicated.
- Distant `last`s are consistent with our memory-aware semantics of initialization (see Section 4.3.2): initializations are propagated to flows that are structurally equivalent.

The above remark gives a reading grid for MADL programs: (local) states are made of a *unique* value per `init` expression contained in the scope. Limiting distant `last` to resumed blocks ensures that the semantics does not depend on the position of `init`. A distant `last` can be viewed as a syntactical convenience for the first version.

5.2 Recursive Array Aggregation

Some algorithms, such as the ones based on the computation of a pivot, use already computed elements to define subsequent ones. Efficient implementations are easily specified in an imperative language: an array is allocated without initial values and it is referred to as needed during the computation. In this case, it is the programmer's responsibility to ensure that the program only accesses values that have been computed.

In SCADE, where partially defined data may not exist, such algorithms are usually implemented by initializing the data-structure (e.g., vector, matrix, ...) and mimicking imperative loops with a `foldi` iterator, passing the partially computed result from one iteration to the next. At each iteration, one element is possibly modified with the *copy-and-modify* functional construct. The extra costs are twofold. The initialization code is actually useless, i.e., it is dead-code, and accumulation and functional modification of the data-structure introduce copies that are difficult to remove.

To avoid these penalties, we propose a safe way to refer to already constructed elements, which prevents uninitialized accesses. Sinkarovs et al. [Sin+17] present a general method to implement *recursive array aggregation*, i.e., self-referencing definition of arrays, by using lazily evaluated arrays. However, this is a dynamic mechanism. Elements are computed on demand, hence the evaluation order and the possible dead-locks are only discovered at runtime. Moreover such evaluation introduces extra execution costs to keep track of computed elements. We pursue a more modest objective: the iteration order is fixed by the `forward` construct so that it can be implemented efficiently.

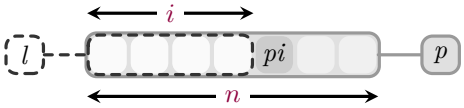
5.2.1 Referencing the Built Part

The introduction of a safe reference to the already computed elements of aggregated arrays is provided through the alias construct — $\cdot$ **as** p — that has a slightly different semantics in the interfaces of **forward** iterators than in normal expressions. As a simple example of a recursive array aggregation, we propose to compute an array of powers of 2 in a rather inefficient way, using the formula:

$$2^{n+1} = 1 + \sum_{k=0}^{n} 2^k$$

A MADL implementation follows, where iteration sizes —«n» and «i» — are explicit, even though they could have been omitted. The reset conditions are implicit, thus both iterations are restarted, i.e., a reset occurs at each cycle and at each iteration of the outer loop.

```
let pow2 «n» () returns (p):
  forward «n» [size i] returns (p = [pi] as l):
    forward «i» ([pk] = l) returns (pi = {s}):
      s init 1 = last s + pk;
```



Unlike normal aliases, the alias — [pi] **as** l — introduces a name — l — for the array of already computed elements only. The variables p and l are thus different for two reasons: (i) they have different scopes: p is part of the outermost scope while l is part of **forward** scope and (ii) their sizes differ: p has size n whereas l has size i, which is the existentially quantified size associated to the current index (see Section 5.1.3). Hence, the inner iteration, that sums the previous powers of two starting at 1, has size i.

The iteration frontier (bis) The typing rules for the aggregation constructs are slightly more complex than the ones we sketched in Section 5.1.1. In an iteration of size η and with an expression e of type τ, the aggregation [e] has type $[\iota]\tau$ (instead of $[\eta]\tau$), where ι is the existentially quantified size that corresponds to the index.

For the result to have the correct size, the aggregated array of type $[\iota]\tau$ must be turned into a value of type $[\eta]\tau$. This is all the more important as the existentially quantified size ι may not escape its scope, i.e., the **forward** construct. We express this size shift with a new *built-array* construct — $|\cdot|$ —, that is inserted around aliases constructs that contain aggregation, or aggregations if no aliases are present. In our prototype, they are inserted during name resolution. By making the built-array constructs explicit, the pow2 operator rewrites:

```
let pow2 «n» () returns (p):
  forward «n» [size i] returns (p = |[pi] as l|):
    forward «i» (|[pk]| = l) returns (pi = {s}):
      s init 1 = last s + pk;
```

Built-array constructs are introduced both for aggregation — |[pi] **as** l| — and current element naming — |[pk]| — but are unnecessary around accumulation — {s} — in which the inner and outer expressions have the same type. In an iteration of size η with index ι, the built-array construct — $|\cdot|$ — is given the type signature $\forall\, \kappa_1\, \kappa_2.\ [\kappa_1 * \eta + \kappa_2]\tau \longrightarrow [\kappa_1 * \iota + \kappa_2]\tau$. This size relation generalizes our initial need to turn ι into η, which is retrieved with $\kappa_1 = 1$ and $\kappa_2 = 0$.

These typing relations induce size constraints of the form $\eta_1 * \iota + \eta_2 = 0$. Such constraints may be simplified because the ι variable is existentially quantified as Section 2.4.4 explains. Abstract sizes allow to decompose polynomial constraints, i.e., $\eta_1 = 0$ and $\eta_2 = 0$, on the condition that η_2 cannot capture the abstract size ι. This is the case here since the coefficients κ_1 and κ_2 are part of the outer scope (they are used in the type of forward arguments or results).

Compared to the typing relations, finding memory locations for built-array constructs is much more involved and imposes some extra conditions on the structure of the arrays. These restrictions are presented in Section 6.2.5.

The typing constraints of built-arrays The built-array construct transforms the outermost dimension of the produced array. Hence, the structural expressions used inside built-array constructs may not produce multidimensional arrays whose inner dimensions depend on the existentially quantified size. To illustrate this, we propose contrived functions that compute vectors or matrices filled with 0.

```
let zero_vec () returns (m):
  forward
  returns (m = flatten ([mi]) as _):
    mi = [|0,0,0|]
```

```
val zero_vec: size i. [3*i]int
```

```
let zero_mat () returns (m):
  forward
  returns (m = transpose ([mi])):
    mi = [|0,0,0|]
```

```
val zero_mat: size i. [3][i]int
```

```
let zero_mat () returns (m):
  forward returns (m = transpose ([mi]) as _):
    mi = [|0,0,0|]
```

```
Error: This expression has type [0][3]int
but an expression of type [`i][3]int was expected
```

```
Note: Size 0 is not compatible with size 1
```

In the first operator, the partial array construct is introduced around a useless partial alias, i.e., `|flatten ([mi]) as _|`. It applies to a value of type `[3*ι]int` where ι is the existentially quantified size that corresponds to the current index. It has the expected form.

The second and third version are identical, except that the latter introduces a built array construct: `|transpose ([mi]) as _|`. In this case, the partial array has type `[3][ι]int`, hence the size ι is transmitted in the type of the result. This causes an error since the existentially quantified size would escape its scope.

The Cholesky Decomposition Among the motivations for recursive array aggregation was the pursuit of an efficient and declarative description of the Cholseky decomposition. Each element should be computed once, at its final destination, without losing the declarative aspects, i.e., all the accessible values should be fully and uniquely defined.

The Cholesky decomposition factorizes a matrix M into a product of a lower triangular matrices L and its conjugate transpose L^*, i.e., $L^*[i,j] = \overline{L[j,i]}$ where $\overline{z}$ denotes the conjugated complex of z and elements are referred with bracketed lists of indexes. M must be Hermitian, i.e., $M^* = M$, and definite positive, i.e., all its eigenvalues are (strictly) positive. This factorization finds L such that

$$M = LL^*$$

For real matrices, conjugation amounts to the identity, hence the above relation simplifies into $M = LL^T$ where L^T is the transpose of L, on the condition that M is symmetric positive definite. While the conditions on M might look restrictive, they are fulfilled in several stochastic contexts such as Monte-Carlo simulations or Kalman filters, because correlation matrices are actually symmetric and (semi)definite.

The elements can be computed with the following explicit formulas. The $\cdot$ operator denotes the scalar product and we use a PYTHON-like syntax for *index ranges* — $[a:b]$ — to denote the sub-vector starting at index a and spanning up to but not including the b index excluded and hence of length $b - a$:

$$L[i,j] = \begin{cases} \dfrac{M[i,j] - L[i,0:j] \cdot L[j,0:j]}{L[j,j]} & \text{if } j < i \\ \sqrt{M[i,j] - L[i,0:j] \cdot L[j,0:j]} & \text{if } j = i \\ 0 & \text{if } j > i \end{cases}$$

The computation of an element at index $[i,j]$ where $j \leq i$ uses the already computed elements from the rows i and j by evaluating the scalar product of the left parts: $L[i,0:j] \cdot L[j,0:j]$. These recursive access schemes are depicted in Figure 5.7 The computation of an element needs both the elements that are above and on its left. This turns out well, such an access scheme is compatible with both a row-by-row computation, entitled the *Cholesky–Banachiewicz* algorithm and a column-by-column computation, named the *Cholesky–Crout* algorithm.

A MADL implementation follows. It defines a `fuse` that builds an array by concatenating a left array, a central element and a right array. The implementation of the transformation closely follows its definition. The built-array constructs are omitted, they are introduced around aliases.

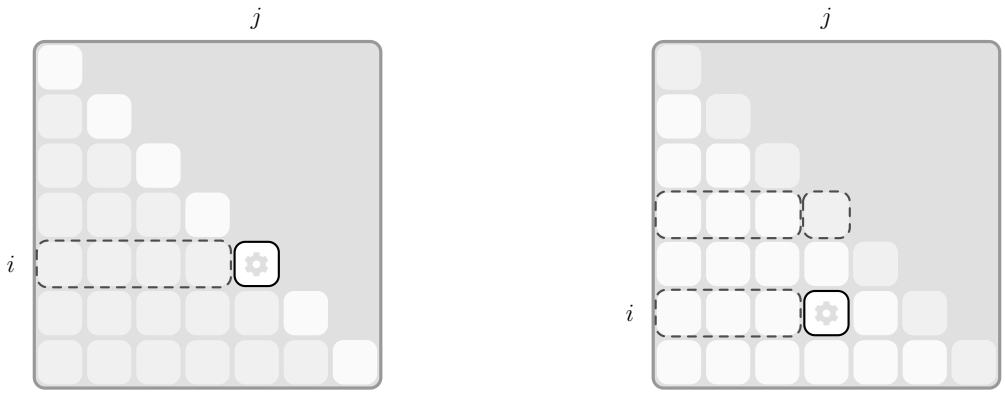


Figure 5.7: *Access scheme for the Cholesky decomposition computation*

```
// Fusion of left part + diagonal element + right part
let fuse (l,d,r) = (l ++ [| d |] ++ r)

// Cholesky-Banachievik algorithm: row by row
let cholesky (m of ['n]['n]_) returns (l of ['n]['n]_):

  forward ([fuse (miL, mii, miR)] = m)
  returns (l = [fuse (liL, lii, liR)] as c):

    // Definition of the left part
    forward ([mij] = miL) ([fuse (cjL, cjj, _)] = c)
    returns (liL = [lij] as ciL):
      lij = (mij - dot (cjL, ciL)) / cjj;
    end

    // Definition of the middle part
    lii = sqrt (mii - dot (liL, liL));

    // Definition of the right part
    forward ([_] = miR)
    returns (liR = [lij]):
      lij = 0;
    end
```

```
val fuse: size i j. type a. [-1+i]a, a, [j-i]a → [j]a
val cholesky: size i. [i][i]int → [i][i]int
```

5.2.2 Initialization

Recursive definitions often need some initial cases that are handled separately. As an example, we would like to build a rotated Pascal's triangle, i.e., a matrix M such that $M[i][j] = M[i-1][j] + M[i][j-1]$ for i and j strictly greater than 1 and $M[i][j] = 1$ elsewhere:

```
[| [| 1, 1, 1, 1 |]  ,
   [| 1, 2, 3, 4 |]  ,
   [| 1, 3, 6, 10 |] |]
```

Such a definition requires to initialize the first row and column with 1 before computing the other elements with the recursive definition. This is possible since we may use any structural expressions inside the built-array constructs. To define our Pascal's triangle, we use the `cons` and `snoc` operators that we recall here:

```
let cons (a,u) = (concat ([|a|], u))
and snoc (u,a) = (concat (u, [|a|]))
```

```
val cons: size i. type a. a, [-1+i]a → [i]a
val snoc: size i. type a. [-1+i]a, a → [i]a
```

Our implementation follows an imperative description. We define the first row, then we compute

the next ones by initializing the left-most element before computing the others. In the iteration that builds rows, we traverse the previous row to access the element above.

```
let pascal () returns (m):

  // Compute the first row
  forward returns (m0_ = [m0j]): m0j = 1 end  // m0_ = [_]1

  // Compute the next rows: r is the last computed row
  forward returns (m = cons (m0_, [mi_]) as snoc (_, r)):

    // Compute the first element
    mi0 = 1;

    // Compute the next elements: ci is the last computed element
    forward (cons (_, [rj]) = r)
    returns (mi_ = cons (mi0, [mij]) as snoc (_, ci)):

      mij = rj + ci
```

```
val pascal: size i j. [1+i][j]int
```

Since the type system does not ensure that sizes are positive, the inferred type scheme is equivalent to $\forall\, \nu\, \mu.\ () \longrightarrow [\nu][\mu]\texttt{int}$. The use of complex patterns — `cons (·, [·]) as ·` — in the partial aliases is crucial. It ensures that the two aliases — `· as snoc (_, ·)` — are valid.

5.3 Multi-dimensional Iteration

MADL supports uni-dimensional arrays. Multi-dimensional ones like matrices are handled as arrays of arrays. Likewise, the `forward` iteration allows for traversing one dimension. Hence a `mat_sum` operation that sums the elements of a matrix, could be defined as:

```
let mat_sum <<n, p>> (m) returns (s):
  forward <<n>> ([mi] = m) returns (s = {t} init 0):
    forward <<p>> ([mij] = mi) returns (t = {u} init last t):
      u = last u + mij;
```

```
val mat_sum: size i j. «i, j» → [i][j]int → int
```

The sizes — $«n»$ and $«p»$ — are stated explicitly, although they could be omitted. This implementation is valid and is given the expected type scheme. However, it is not as trivial as it might look: the state of the inner `forward` (of size $«p»$) is initialized with the previous value of the accumulator of the outer `forward` (of size $«n»$). Since `last` does not introduce copies, both t and u are stored at the same memory location. However, they do not conflict: `forward` scopes are restarted — `restart` —, hence the inner iterator state — u — is local.

To avoid this subtle two-fold initialization, the two iterations may use the same state, by specifying a `resume` reset condition:

```
let mat_sum <<n, p>> (m) returns (s):
  forward <<n>> ([mi] = m) returns (s = {t}):
    forward <<p>> ([mij] = mi) returns (t = {u}):
      u init 0 = last u + mij;
    resume
```

```
val mat_sum: size i j. «i, j» → [i][j]int → int
```

We move the initialization from the iteration interface to the iterated expression. Since the inner `forward`s is resumed, initialization would have been valid in both interfaces as well. This situation — perfectly nested iteration where inner `forward` are resumed — is expressed more succinctly by the *multi-level iteration* syntax, that we explain in Section 5.3.1:

```
let mat_sum <<n, p>> (m) returns (s):
  forward <<n>> ([mi ] = m ) returns (s = {t}),
          <<p>> ([mij] = mi) returns (t = {u}):
    u init 0 = last u + mij;
```

```
val mat_sum: size i j. «i, j» → [i][j]int → int
```

The two levels are separated by the comma, that ends the first line of the `forward` interface. Notice that the variables introduced in the first level are reused in the second one. The two levels

are now rather aligned and it is tempting to fuse them. This is the role of *multi-level interfaces* presented in Section 5.3.2. Our `mat_sum` example can be written:

```
1 let mat_sum <<n, p>> (m) returns (s):
2   forward <<n>>, <<p>> ([[mij]] = m) returns (s = {{u}}):
3     u init 0 = last u + mij;
```

```
val mat_sum: size i j. «i, j» → [i][j]int → int
```

The two iteration levels are still separated by a comma, inserted between iteration sizes «*n*» and «*p*». The aggregation and accumulation constructs are now nested to name the current element — `[[`*mij*`]]` — and return the accumulator — `{{`*u*`}}` —.

5.3.1 Multi-level Iteration

The **forward** iterator is inspired by the *for-loop* and *for-initial* loop constructs of the SISAL language [FCO90]. In addition to the integer — `i in 1, n` — and array — `ai in a` — range generators, SISAL provides *compound* and *multidimensional* range generators, introduced with the **dot** and **cross** operators that combine generators[4]

- The **dot** operator defines a range generator that traverses multiple ranges or arrays *simultaneously*. The MADL equivalent is the use of multiple *named current elements* — `[`*ai*`] = `*a* — with the difference that MADL forces equal sizes for the traversed arrays whereas SISAL uses the maximum valid size, i.e., the minimum of the generator sizes.
- The **cross** operator denotes a range generator that traverses multiple ranges *independently* by building all possible pairs. This corresponds to nested loops, which are described in MADL using multi-level iteration.

Instead of a single interface in **forward** scopes, multi-level iteration allows a comma separated list of interfaces, as depicted in the updated syntax below. The comma — `,` — plays the role of the **cross** keyword of SISAL.

			Expressions	d	p	c
e	::=	...				
	|	**forward** i, ..., i: c $[r]$	forward iteration	×	×	✓

Elaboration Multi-level iterations are eliminated early by elaborating them into nested simple iterations. In our prototype, the transformation is embedded in the name resolution pass, that converts a concrete syntax tree (CST) into an abstract syntax tree (AST) that does not contain multi-level iterations. The elaboration is defined by the following rewriting rule, that is applied repeatedly:

$$\textbf{forward } i_1, \ldots, i_n, i_{n+1}\text{: } c\ [r] \implies \begin{array}{l}\textbf{forward } i'_1, \ldots, i'_n\text{:}\\ \quad \textbf{forward } i'_{n+1}\text{:}\\ \quad\quad c;\\ \quad \textbf{resume}\\ [r]\end{array}$$

All nested iterations are resumed. The reset condition of the outermost iteration is that of the multi-level **forward**. Hence the state of the iterated expression is never reset during iteration. The *multi-level* interfaces $i_1, \ldots, i_n$ are turned into *unary* interfaces $i'_1, \ldots, i'_n$, a process that we describe in Section 5.3.2. In this section, we only use unary interfaces that are transmitted unaltered, i.e., such that $i_k = i'_k$.

We recall the third version of the `mat_sum` operator that we have introduced above. It computes the sum of the elements of a matrix.

```
let mat_sum «n, p» (a) returns (s):
  forward «n» ([ai] = a) returns (s = {t}),
          «p» ([aij] = ai) returns (t = {u}):
    u init 0 = u + aij
```

[4] The SISAL tutorial illustrates these generators. http://www2.cmp.uea.ac.uk/~jrwg/Sisal/10.More.loops.html

This example illustrates an important feature. The same variables, e.g., ai and t, may appear it several levels, as long as scoping rules are respected. This allows to chain the current element naming in arguments and accumulation or aggregation for results. We recall that the above implementation uses the *scope unfriendly* syntax for the interfaces. Indeed, the elaborated and expanded version (we do not separate the inner block for iterations, see Section 5.1.1) makes clear that the uses of variables are correct:

```
let mat_sum «n, p» (a) returns (s):
  s = (forward «n» ([ai]) returns ({t}):
         t = (forward «p» ([aij]) returns ({u}):
                u init 0 = u + aij
              resume) (ai)
       restart) (a)
```

Because scopes are different, we may reuse variable names for the inner and outer values in an interface. The following version of `mat_sum` is thus equivalent:

```
let mat_sum «n, p» (a) returns (s):
  forward «n» ([a] = a) returns (s = {s}),
          «p» ([a] = a) returns (s = {s}):
    s init 0 = s + a
```

Asymmetry between arguments and results Our unique example so far traverses and accumulates among two dimensions using a two-level iteration. Arrays may also be traversed using a subset of the iterated dimensions. For instance, the tensor product of two vectors u and v, denoted $u \otimes v$ is defined as: $(u \otimes v)_{i,j} = u_i v_j$. In MADL, it may be defined as a unique two-level iteration:

```
let tensor (a, b) returns (c):
  forward ([ai] = a) returns (c = [ci]),
          ([bj] = b) returns (ci = [cij]):
    cij = ai * bj
```

Contrary to the traversed arrays, the produced values, either accumulated or aggregated must cross all the iteration levels. Indeed, variables may not be used outside of the scope that defines them. This asymmetry is more visible in Sisal, where the multi-dimensional iterations have a single return clause, independently of the number of dimensions. Section 5.3.2 explains why we take a different approach.

Initialization of aggregated arrays We describe in Section 5.1.4 how MADL allows distant `last`s but bans distant `init`s. The former are only possible when nested scopes are resumed, which is the case for multi-level iterations. The constraint on initialization applies in a similar way, and the following implementation of matrix-vector product is rejected:

```
1 let not_mat_vec (a,u) returns (v):
2   forward ([ai] = a) returns (v = [vi]),
3           ([aij] = ai) ([uj] = u) returns (vi = {s}):
4     s init 0 = last s + aij * uj
```

```
Error: Initialization of aggregated element
```

```
Note: Aggregated here
```

The error comes from the second level of iteration where the corresponding `forward` scope is resumed, and hence the accumulation is part of the state of the first iteration. It cannot be aggregated. Moreover, the accumulator is not reset at each iteration as the operator would then not compute a matrix-vector product. This definition works fine with two nested `forward` iterators because the inner one is restarted:

```
1 let mat_vec (a,u) returns (v):
2   forward ([ai] = a) returns (v = [vi]):
3     forward ([aij] = ai) ([uj] = u) returns (vi = {s}):
4       s init 0 = last s + aij * uj
```

```
val mat_vec: size i j. [i][j]int, [j]int → [i]int
```

5.3.2 Multi-level Interfaces

To add dimensions to the `mat_sum` operator defined above, it suffices to duplicate the forward levels — «k» ([a] = a) returns (s = {s}) — and change the dimension k, or omit it. However, to avoid the cascade of identical names, each one used only once, we would like to benefit from a kind of referential transparency. This is provided by *multi-level interfaces*, which are nested aggregation and accumulation constructs. They do not need to extend MADL syntax. The two following versions are equivalent:

```
let mat_sum «n, p» (a) returns (s):
  forward «n» ([ai] = a) returns (s = {t}),
          «p» ([aij] = ai) returns (t = {u}):
    u init 0 = u + aij
```

```
let mat_sum «n, p» (a) returns (s):
  forward «n», «p» ([[aij]] = a)
  returns (s = {{u}}):
    s init 0 = s + a;
```

The version on the right is still a multi-level iteration since the comma separates the two size parameters «n» and «p», but the interface of the outer level (of size n) is empty. Compared to usual inlining, two differences are worth mentioning:

- Variables of the inner interface are inlined in the outer one, regardless of whether they are used or defined: t is defined in t = {u} but ai is used in [aij] = ai.
- The resulting arguments and patterns are moved downward to the inner levels. As a consequence, the inner variables aij and u still appear in the level that corresponds to their scope.

Elaboration We only give an informal description of interface normalization. Given the result p_i = c_i of the `forward` interface at level i, i.e., the iteration over the i^{th} dimension:

1. Compute its rank $r = rank(c_i)$, i.e., the maximum depth of nested iteration or accumulation.
2. If $r > 1$:
   (a) Define c'_i as the computation c where the sub-expression of the outer-most aggregation — [c] — or accumulation — {c} — is replaced by a fresh variable x, i.e., [x] or {x}.
   (b) Replace the result at level i by x = c.
   (c) Replace the result p_{i-k+1} = c_{i-k+1} at level $i - k + 1$ by p_{i-k+1}, p_i = c_{i-k+1}, c.
3. Repeat and proceed in a similar way for arguments until all interfaces only contain expressions of rank 1.

As an example, the implementation of the tensor product at the left below is elaborated to the version at the right by introducing a fresh (well-named) variable ci and splitting the result of the second level in two:

```
let tensor (a, b) returns (c):
  forward ([ai] = a), ([bj] = b)
  returns (c = [[cij]]):
    cij = ai * bj
```

```
let tensor (a, b) returns (c):
  forward ([ai] = a) returns (c = [ci]),
          ([bj] = b) returns (ci = [cij]):
    cij = ai * bj
```

In Sisal, *range generators* are separated from *return clauses*. The former are composed with `dot` and `cross` operators, while the latter are made of a unique `array of ...` or `value of ...` construct that specifies whether the produced values are aggregated or accumulated, for all the dimensions of the loop generator. In Sisal, the tensor product is written:

```
w := for ui in u cross vj in v                    (Sisal)
  returns array of ui * vj
  end for
```

It is impossible to mix the `array of` and `value of` return clauses in a single iteration construct. Contrary to Sisal, our approach treats traversed arrays and produced values on an equal footing. This symmetry allows to mix accumulation and aggregation in the returned values:

```
let vec_mat (u, a) returns (v):
  forward ([ui] = u), ([[aij]] = a) returns (v = {[vj]} init [_]0):
    vj = last vj + ui * aij;
```

```
let vec_vec (u, v) returns (s):
  forward ([ui] = u) ([vi] = v)
  returns (s = {s} init 0):
    s += ui * vi
```

(a) Scalar product

```
let vec_mat (u, b) returns (w):
  forward ([ui] = u), ([[bij]] = b)
  returns (w = {[wj]} init [_]0):
    wj += ui * bij
```

(b) Vector-matrix product

```
let mat_mat (a, b) returns (c):
  forward, ([[aik]] = a), ([[bkj]] = b)
  returns (c = [{[cij]}] init [_][_]0):
    cij += aik * bkj
```

(c) Matrix product

```
let mat_vec (a, v) returns (w):
  forward, ([[aij]] = a) ([vj] = v)
  returns (w = [{wi}] init [_]0):
    wi += aij * vj
```

(d) Matrix-vector product

Figure 5.8: *Linear algebra operators*

The returned value `v = {[vj]} init [_]0` is defined by accumulation on the outer dimension and aggregation on the inner one. Its elaborated form is obtain by introducing a variable w:

```
let vec_mat (u, a) returns (v):
  forward ([ui] = u) ([ai] = a) returns (v = {w} init [_]0),
          ([aij] = ai) returns (w = [vj]):
    vj = last vj + ui * aij;
```

Accumulation syntactic sugar In an attempt to attract C programmers, we provide an *assign and modify* syntax for scalar operator: `+=`, `-=`, `&=`, *etc.* For instance, these two versions of the `mat_sum` operator are equivalent:

```
let mat_sum «n, p» (a) returns (s):
  forward «n», «p» ([[aij]] = a)
  returns (s = {{t}}):
    t init 0 += aij
```

```
let mat_sum «n, p» (a) returns (s):
  forward «n», «p» ([[aij]] = a)
  returns (s = {{t}}):
    t init 0 = last t + aij
```

An assign operation p *op*= c is elaborated to p = `last` $\widetilde{p}$ *op* c, where the *computation* $\widetilde{p}$ is the pattern p with initialization and alias constructs removed. Using this syntax, Figure 5.8 gathers the definitions of basic linear algebra operations, to emphasize their similarities. In the `mat_mat` and `mat_vec` operators, the comma immediately follows the `forward` keyword since the interface of the first layer is empty.

Dimensions mix-up The correctness of multi-level interfaces relies on a consistent number of commas, square brackets and curly ones. Invalid programs give rise to two kinds of structural errors that are reproduced below:

```
1 let mat_add (a,b) returns (c):
2   forward, ([[aij]] = a) ([[bij]] = b)
3   returns (c = [cij]):
4     cij = aij + bij
```

```
Error: Undefined variable c
```

```
1 let mat_add (a,b) returns (c):
2   forward, ([aij] = a) ([[bij]] = b)
3   returns (c = [[[cij]]]):
4     cij = aij + bij
```

```
Error: To many iteration levels
```

On the left, an accumulation is missing. The variable c is introduced in the scope of the iteration over the first dimension, and is undefined in operator's scope. On the right, there is one accumulation too many an elaboration fails.

Distant initialization (bis) As explained for multi-iteration, initialization may not be declared on aggregated values, it must be moved out of the aggregation. Indeed the following implementation of matrix multiplication is rejected (Figure 5.8 gives the correct one):

```
1 let mat_mat (a,b) returns (c):
2   forward, ([[a]] = a), ([[b]] = b) returns (c = [{[c]}]):
3     c_init_0 += a*b
```

```
Error: Uninitialized last
```

```
Error: Initialization of aggregated element
```

```
Note: Aggregated here
```

Flowers... To illustrate how multi-dimensional iterations allows concise definitions, we implement a unidimensional convolution. It expects the size «*i*» of a kernel k and an array a on which convolution applies.

```
let convol_1D «i» (k, a) returns (b):
  forward, ([[kj]] = repeat «_» (k)) ([[aik]] = window «i» (a))
  returns (b = [{bi}] init [_]0):
    bi += kj * aij
```

The iteration has two levels, separated by the comma that follows the `forward` keyword. It traverses the slices of a, built using `window «k» (a)` and the constant matrix made of copies of k, constructed with the `repeat` operator. As Chapter 7 will explain, these operators are turned into index computations. This indirect definition can be easily extended to a bidimensional convolution:

```
let convol_2D «i, j» (k, a) returns (b):
  forward, ([[k]] = repeat «_» (k)) ([[a]] = window «i» (a)),
         , ([[k]] = repeat «_» (k)) ([[a]] = window «j» (a))
  returns (b = [{[{b}]}] init [_][_]0):
    b += k * a
```

We showed in Section 5.2.2 a possible definition of Pascal's triangle. We propose here a more elegant definition, in our opinion, that constructs the matrix a more regular way, following the symmetry of the result. It computes: (i) the top-left corners, (ii) the first row and column and (iii) the remaining parts of the matrix by traversing the already computed parts.

```
1 let pascal () returns (m):
2   d = 1;                                          // Top-left corner: m[0][0]
3   forward returns (c = cons (d, [i])): i = 1 end  // Left column    : m[0:n][0]
4   forward returns (r = cons (d, [i])): i = 1 end  // Top row         : m[0][0:n]
5   forward (cons (_, [c]) = c) returns (m = cons (r, [m]) as snoc (_, r)),
6           (cons (_, [r]) = r) returns (m = cons (c, [m]) as snoc (_, c)):
7     m = c + r
```

```
Error: Invalid locations
```

This program is correct but allocation inference fails, after initialization checking and type checking. We explain allocation inference in Chapter 6, where we show how to insert some annotations to guide the inference process and make this example pass allocation inference.

... and Monsters The landscape of multi-dimensional iteration provides concise and hopefully intuitive descriptions for typical algorithms. However, some monsters are hiding: programs that are well-defined, but barely understandable. Their commonality is the complex scoping rules that arise from nested accumulation and aggregation constructs.

Scope inversion. The two sides of equations in interfaces are part of different scopes, which allows for reusing names, but may lead to confusing definitions, in particular with multi-level interfaces. Once again, we illustrate this point with two equivalent definitions of matrix multiplication:

```
let mat_mat (a, b) returns (c):
  forward, ([[a]] = a), ([[b]] = b)
  returns (c = [{[c]}] init [_][_]0):
    c += a * b
```

```
let mat_mat (a, b) returns (c):
  forward, ([[b]] = a), ([[a]] = b)
  returns (c = [{[c]}] init [_][_]0):
    c += b * a
```

In the version on the right, the variable b introduced in the 2nd level — `[[b]] = a` — is not the one used in the definition of the variable b in the 3rd level — `[[a]] = b` — because this two-dimensional array traversal reads the outer value from the 1st scope.

Unbalanced multi aggregation. Some terms may have sub-expressions with different depths of nested accumulations or aggregations, e.g., `[`*ti*`] ++ [[`*bij*`]]` (recall that `++` denotes the application of the `concat` built-in operator). Here lies one of the arguments against the introduction of multi-level interfaces in the inner-most scope. This scope is consistent with the most deeply nested introduction only, i.e., *bij* in our example. As a consequence, the following definition is correct:

```
let half_mat () returns (m):
  forward, returns (ti = [tij]) (m = [ti] ++ [[bij]]):
    tij, bij = 1, 0
```

This operator builds a matrix whose top half contains 1 while the lower half is filled with 0. It might be surprising that the *ti* variable, defined by the second level — `ti = [`*tij*`]` —, hence in the scope of the first level, can be used in the definition of *m*: `m = [`*ti*`] ++ [[`*bij*`]]`. The reason becomes clear in the elaborated version:

```
let half_mat () returns (m):
  forward returns (m = [ti] ++ [bi]),
          returns (ti = [tij]) (bi = [bij]):
    tij, bij = 1, 0
```

These examples argue in favor of a more controlled iteration construct that restricts the possible interfaces. In particular, the complex scoping rules that arise from multi-level interfaces are only understandable through the elaboration process. We leave a restricted multi-dimensional iteration construct for future work.

Conclusion

The `forward` construct of MADL is inspired by the stream-based iterations that have been proposed in various declarative languages such as Lucid [WA+85] and Sisal [FCO90]. It aims at conciliating the view of arrays as finite streams with the infinite streams of Scade.

From a syntactical point of view, the `forward` iteration construct has at least two advantages over the second-order iterators (`map`, `fold`, *etc*). (i) Interfaces are clearer because they are free of local accumulators and the equation-like syntax allows to introduce results in separate clauses instead of decomposing a list of results (see Section 5.1.2). (ii) Multidimensional iterations are easily described with the `forward` construct. Nested second-order iterators are more limited, e.g., it is impossible to capture indexes.

Iterating with streams The `forward` construct is based on the idea that iteration bodies define faster flows, i.e., flows that are computed by groups of k where k is the iteration size. Accumulations are thus performed by accessing the previous values of the flows, i.e., reading the state. A particular use-case deserves dedicated handling. In order to use `forward` iterators in combinational operators, iterations that are reset at each synchronous cycle must be treated as stateless. This is introduced by the `restart` reset condition that allows for local states. Local states also relax the conditions imposed on initialization (constant values only) by allowing the use of the flows that are computed outside of the restarted scopes.

The various examples show that *iteration indexes are (almost) useless*. Indeed, we did even not prototype explicit projections.[5] Most of the accesses found in data-intensive applications follow regular patterns. Expressing them with a dedicated construct instead of general index manipulations improves readability of the model and static verification of accesses and code generation by providing strong structural properties to the compiler.

We proposed a novel extension, to our knowledge, of stream-based iteration to describe recursive array definitions in a declarative manner. Our *partial array aliases* enforce both a simple compilation scheme, i.e., the evaluation order is still fixed by the construct, and a safe definition, i.e., all values are fully defined.

An alternative approach would be to view recursive array aggregation as an accumulation that starts from an empty array and appends one element at each cycle. In this context, the already constructed array would be represented by the previous value (`last`) of the accumulator. We preferred a custom alias to a custom access to the state because the introduced typing and

[5] Explicit indexing would require some hard work. Instead, we preferred to explore in depth the possibilities of current element naming.

allocation exemptions are localized in the interface. Handling recursive array aggregation as special accesses to the state would require distinguishing the `last` constructs that apply to these recursively built arrays in the interface and throughout the iterated scope.

Our approach for multi-dimensional iteration is too basic. It only amounts to syntactic sugar over the unidimensional iterations. This may be sufficient for an intermediate language but it is probably not adapted for an input language. In particular, the exotic scoping rules that arise are hard to understand without elaborating them. A standalone multi-dimensional iteration construct would be more usable. We identified two restrictions that could help: (i) banning the intricate iteration clauses where a variable introduced in one iteration level is used in another, (ii) allowing a single return clause. We think that allowing for nested aggregations and accumulations in return clauses may be useful if a special rule for the `last` construct is provided.

Multi-instantiation We only studied a multi-execution iteration construct for now. As an alternative to the SCADE iterators, which have a multi-instantiation semantics, we would like to provide a `foreach` iterator, whose structure would resemble the one of `forward`, except that the state is duplicated.

Such a `foreach` iterator would not cover all the uses of SCADE iterators. Since we perform accumulation using the state that is duplicated in multi-instantiation iterations, `foreach` cannot describe accumulations, i.e., `fold`. However, it seems that most of the uses of `fold` do not introduce delays between the previous and new values of the accumulator. In such situations, iteration may be decomposed into a multi-instantiation part followed by a multi-execution one that accumulates. This claim requires confirmation.

The founding goal of MADL lies in providing a declarative way to describe most of the possible imperative implementations of algorithms. This aims at (i) allowing a precise control of the generated code from the language, and (ii) providing a solid basis to perform code transformations and optimizations at a declarative level. A complete separation of the multi-instantiation and multi-execution iteration constructs would prevent any loop fusion between them. This might be a reason for providing a common way of describing both iteration models in a single construct.

Multi-execution in a flow-based synchronous language To our knowledge, a stream based iteration over arrays has not yet been studied for LUSTRE-like languages. However, the idea of a locally faster time is not new. Mandel, Pasteur, and Pouzet [MPP15] propose *reactive domains* to allow a modular decomposition of clocked processes into faster processes. In the context of program refinement, Mikac and Caspi [MC05] extend this notion to stream functions by formalizing *temporal refinements*, that allow for faster clocks in nodes.

A multi-execution construct has been studied as an extension of the functional semantics proposed by Colaco et al. [Col+23] for a synchronous language that contains the principal constructs of SCADE, e.g., automata. A similar `forward` construct has been prototyped in the companion interpreter.[6] Contrary to MADL, this language does not describe location. For that reason, accesses to the state are only determined by their scope and no distant `last`s are available.

[6] https://github.com/marcpouzet/zrun

6 Memory Allocation

Introduction

MADL is a *declarative* language. Variables thus only serve to describe the graph structure of data-flow dependencies in a linear text. Our language applies the same principle to memory specification. Program variables do not impact memory allocation, which is only determined by the structure of values and expressions. The MADL language gives a precise although indirect specification of memory allocation. This description is supported by (i) the distinction between the *structural* and *computational* expressions and (ii) additional *location annotations* used to require that multiple values share the same location. Altogether, these elements constrain the possible allocations.

Allocation is defined as a type system. We provide a deduction system that selects the valid allocations and an inference algorithm that derives an allocation for implicitly allocated programs. As for sizes, inference is incomplete. Location annotations then play the same role as type annotations: they insert extra constraints that may help allocation inference succeed. The analogy goes beyond checking and inference. For code generation, the type information drives the representation of data in the generated code. Similarly the allocation information controls in a prescriptive way how memory is used.

Memory, accesses and computations To understand how allocation is inferred from MADL programs, let us examine the structure of data-flow programs paying careful attention to memory. Data-flow programs are usually thought of in terms of connected *operations* — the data-flow graph — where the result of an operation is directly linked to its uses. In a naive implementation, memory is ubiquitous: by considering that operations are performed at different moments, each result must be stored somewhere so as to be available for future operations. Schematically, the nodes of the data-flow graph can be decomposed into a compute part and a storage part, as depicted by the puzzle pieces in Figure 6.1.

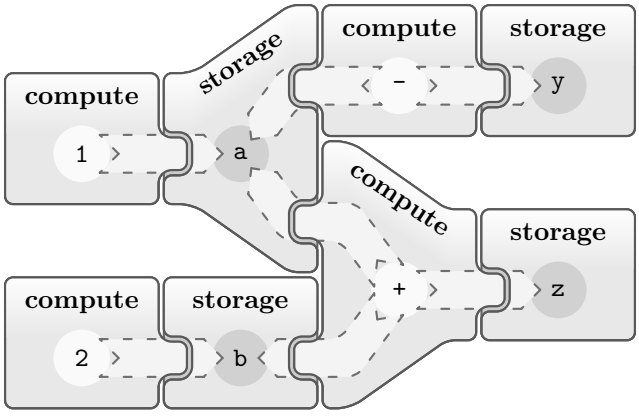


Figure 6.1: *Anatomy of a data-flow program:* `a = 1 ; b = 2 ; y = -a ; z = a+b`

For simplicity, we assume that a computation produces a single value, but may use multiple ones. Conversely, the storage nodes are accessed by multiple computations, but they are written once. The data flows from left to right. The large inner arrows represent accesses: storage is *passive*, i.e., it is written or read, while computations are *active*, i.e., they read or write. The inner

arrows thus always flow from computations to storage. As the shapes of the puzzle pieces suggest, computations may not be connected without an intermediate memory.

These commonplace remarks prepare the handling of the central element of MADL: *views*, i.e., the structural expressions. These computation-less operations eliminate the need to store intermediate values by providing complex *accesses*, that may be though as statically managed indirections. Accesses are inserted either in reading positions (i.e., between a storage and a computation) or in writing positions (i.e., between a computation and the storage). With such accesses, the compute-storage alternation is completed with additional elements, as depicted in Figure 6.2.

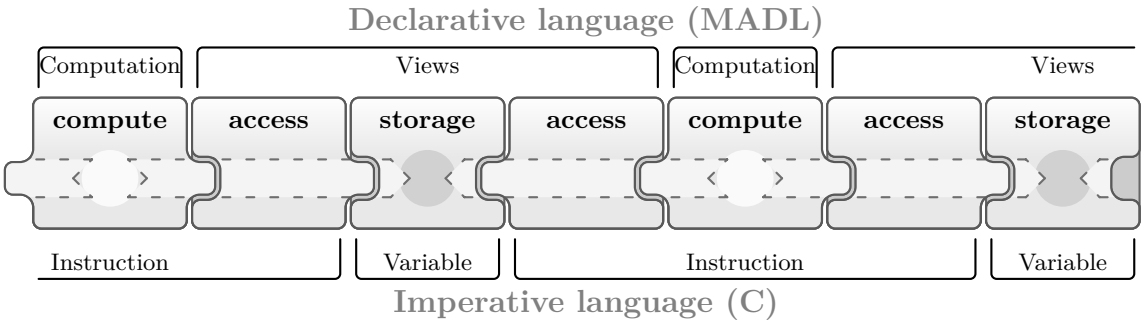


Figure 6.2: *Anatomy of MADL programs: memory, accesses and computations*

MADL programs only manipulate views. The choice of the values that are directly stored in memory is left to the compiler. Inferring allocation amounts to finding some valid position of the storage pieces (that are implicit in the program) so as to complete the puzzle. Contrary to the pure data-flow case where it suffices to insert storage for each result, the task is non-trivial. Indeed, as the shapes of access pieces suggests, the accesses that are used for reading and writing are incompatible: they must be separated by a storage.

Compilation This representation also gives a glimpse of what the compiler does: it translates a declarative form into an imperative form. The source language (MADL) distinguished between the *computational* and *structural* expressions. The former are the active parts of programs, that create new value. The latter are the passive parts, they define multiple views of the memory locations, without modifying them. A different partition applies for the target language (C) where storage is identified with *variables*, while computations and accesses are gathered in *instructions*. Here the accesses result in complex projections when reading or writing the variables.

In short, allocation inference identifies the expressions that must be stored in memory, i.e., it choose the place of the storage pieces. Scheduling and code generation then transform the computation-centered description, where views encompass both storage and accesses to a memory-centered description, where instructions compute and access. Accesses are statically managed indirections that can be combined, moved around, duplicated or eliminated, but they cannot exist in the generated code without computations. This is the reason for providing an explicit copy construct. It allows to turn accesses into a concrete operations.

Type and allocation checking Our description of allocation is based on a type system that discriminates between correctly and incorrectly allocated programs. Because our memory locations are typed, well-allocated programs are also well-typed. In the formalization, we present only the rules to derive allocation correctness, which ensure, by the way, type correctness.

The inference process, though, is split into two phases: types and sizes are reconstructed first, independently of allocation, then memory locations are derived. This stratification is grounded in several observations:

- The language should have a memory-independent semantics, that will exist under typing assumptions. This memory-agnostic level is easier to understand. In particular, reports for typing errors are much more intelligible than allocation errors.

- Memory allocation should act as additional constraints that do not have any semantic content. In particular, since the semantics depend on size, allocation shall not introduce extra size

relations. As a consequence, it might be necessary to add explicit size constraints so that allocation inference succeeds.

Our memory location system ensures the consistency of flow allocations. Each value is written and read at the same location. However, for this specification to be turned into a sequential implementation, additional properties are required. Some of them are unavoidable, e.g., writes must occur at *injective* locations, while others are implementation dependent, e.g., how views are represented at run-time. These aspects are checked independently of the allocation correctness.

6.1 A Location Type System

Allocations are described by associating to each flow a *memory location*, called *data* in the syntax of MADL. Locations represent typed chunks of memory accompanied with a description of how they can be accessed. They are independent from the flows of the program and multiple flows may be associated to the same location.

Before digging into the details of our location type system, we briefly review some simple examples of allocation description to guide the reader through the subsequent forest of Greek letters.

Sharing memory locations As a warm up, let us start with the identity function. This structural, bijective operator can be defined in MADL as:

```
let patt id (x) returns (x)
```

since this term contains no copies, the argument and the result must be allocated at the same place. Operator `id` is given the *allocation scheme*:

$$\forall\,\alpha.\ \forall\,\delta{:}\alpha.\ \delta \longrightarrow \delta$$

It reads: (i) $\forall\,\alpha.$ — given a type α and (ii) $\forall\,\delta{:}\alpha.$ — given a location δ of type α, (iii) $\delta \longrightarrow \delta$ the operator expects 1 argument at location δ and it 'produces' 1 result at location δ. This last part is called *location signature*. Because the location variable δ is typed, this allocation scheme incorporates the usual type scheme: $\forall\alpha.\ \alpha \to \alpha$.

Several outputs can also have the same location, as in the following `double` operator, whose allocation scheme is given on the right:

```
let patt double (x) returns (x, x)
```

$$\forall\,\alpha.\ \forall\,\delta{:}\alpha.\ \delta \longrightarrow \delta,\ \delta$$

Views and projection functors The above schemes illustrate simple memory locations. For more complex structural operators, the memory locations are built using projection functors (see Chapter 3). For instance, the allocation scheme of the `transpose` primitive pattern is:

$$\forall\,\nu\,\mu.\ \forall\,\alpha.\ \forall\,\delta{:}\,[\nu]\,[\mu]\,\alpha.\ \delta \longrightarrow \delta.\texttt{transpose}\ \nu\ \mu\ \alpha$$

It is universally quantified on sizes ν and μ, a type α and a location δ of type $[\nu]\,[\mu]\,\alpha$. The operator expects one argument at location δ and 'produces' its results at location $\delta.\texttt{transpose}\ \nu\ \mu\ \alpha$ where $\texttt{transpose}\ \nu\ \mu\ \alpha$ is the projection functor presented in Section 3.3.2. Extracting the type scheme is more involved: it depends on the type of the `transpose` projection functor, $[\nu]\,[\mu]\,\alpha \to [\mu]\,[\nu]\,\alpha$. Applied to a memory location of type $[\nu]\,[\mu]\,\alpha$, it allows to build a view of type $[\mu]\,[\nu]\,\alpha$. We recover the expected type scheme: $\forall\nu{\cdot}\mu{\cdot}\alpha.\ [\nu]\,[\mu]\,\alpha \to [\mu]\,[\nu]\,\alpha$.

Non-representable views The transpose operator is a direct embedding of the projection functor in the language. However, we argued in Section 3.1.3 that some operations cannot be represented as projection functors. For instance, this is the case for the operator `concat` that concatenates two arrays. Its allocation scheme is:

$$\forall\,\nu\,\mu.\ \forall\,\alpha.\ \forall\,\delta{:}\,[\nu+\mu]\,\alpha.\ \text{«}\nu,\mu\text{»}\ \delta.\texttt{cut}\ \nu\ \mu\ \alpha.\texttt{(1/2)},\ \delta.\texttt{cut}\ \nu\ \mu\ \alpha.\texttt{(2/2)} \longrightarrow \delta$$

A novel component appears here: the size parameters — «ν, μ» — reflect the sizes that are optionally specified when instantiating this operator: `concat` «s_1, s_2» `(a, b)`. Unlike the `transpose` operator, the location of the result is simply δ while the locations of arguments are defined using the `cut` ν μ α projection functor and tuple projections `(1/2)` and `(2/2)` respectively. In short, argument locations are derived from the result location with the inverse projection functor.

Stateful operators To conclude this tour, let us see how the allocation signature describes stateful operators. We recall the two implementations of the counter operator `nat` introduced in Section 4.3.4 and give their respective allocation schemes.

```
let node nat () returns (n):
  n = !last p;
  p = n + 1 init 0;
```

$$\forall \delta_1 : \texttt{int}, \delta_2 : \texttt{int}.\ () \xrightarrow{\delta_1} \delta_2$$

```
let node nat' () returns (n):
  n init −1 = last n + 1;
```

$$\forall \delta : \texttt{int}.\ () \xrightarrow{\delta} \delta$$

The memory location of the state appears above the arrows. The claim of Section 4.3.4 is now explicit: in the `nat'` version, the result is indeed at the location of the state, whereas the copy in the `nat` version allows to separate them.

Summary of allocation schemes Operators are generic transition functions with size parameters. Hence, allocation schemes have the following structure:

$$\forall \vec{\nu}.\ \forall \vec{\alpha}.\ \forall \overrightarrow{\delta : \tau}.\ \text{«}\vec{\eta}\text{»}\ \vec{\varepsilon_i} \xrightarrow{\vec{\varepsilon_s}} \vec{\varepsilon_o}$$

where

- $\forall \vec{\nu}.$ are universally quantified size variables
- $\forall \vec{\alpha}.$ are universally quantified type variables
- $\forall \overrightarrow{\delta : \tau}.$ are universally quantified typed location variables
- $\text{«}\vec{\eta}\text{»}$ is a list of size parameters, that may be specified at instantiation
- $\theta := \vec{\varepsilon_i} \xrightarrow{\vec{\varepsilon_s}} \vec{\varepsilon_o}$ is a *location signature*, where
  - $\vec{\varepsilon_i}$ is the list of input locations
  - $\vec{\varepsilon_s}$ is the list of locations of the state
  - $\vec{\varepsilon_o}$ is the list of output locations

Location signatures describe the memory constraints of monomorphic transition functions, i.e., expressions. In the following, the universal quantification of sizes, types and locations will be gathered in a unique *generalization* object $\mathcal{G}$. Location signatures then look like:

$$\forall \mathcal{G}.\ \text{«}\vec{\eta}\text{»}\ \theta$$

6.1.1 Fully Typed MADL

Before presenting how to check or infer allocations, we formalize the meta-information (size, type and location) and present how programs are supplemented with them.

Formal Sizes, Types and Locations The static analysis of MADL programs is supported by three entwined type systems that describe *sizes*, *types* and *locations*. They are checked together with a single set of deduction rules.

Figure 6.3 recalls the syntax of sizes (η), data types (τ) and projection functors (φ), that are introduced in more detail in Chapter 3. It also defines the syntax of *locations* (ε), that are either *location variables* —δ— or *views* —$\varepsilon.\varphi$— of a location ε through a projection functor φ.

The composition of projection functors induces a natural normalization of locations as a location variable coupled with a unique projection functor, e.g., $\delta.\varphi$, because multiple views may be combined by composing functors:

$$\varepsilon.\varphi.\psi = \varepsilon.(\varphi \gg \psi)$$

Polymorphism Sizes, types and locations possess free variables that are needed to derive the memory allocation relation. As for simple types in ML-like type systems, these variables are introduced and substituted with polymorphism, that allows to make operator signatures generic in size, type and location:

η	::=	***Sizes***	φ	::=	***Projection transformers***
	$\mid \nu,\ \mu,\ \ldots$	variable		$\mid \mathbf{1}_\tau$	identity
	$\mid 0,\ 1,\ \ldots$	constant		$\mid$ `@[`e`<`η`]`$.\ \varphi$	array application
	$\mid \eta + \eta$	sum		$\mid$ `@(k/m)`$.\ \varphi$	tuple application
	$\mid \eta * \eta$	product		$\mid \lambda$`[`i`<`η`]`$.\ \varphi$	array abstraction
				$\mid \lambda$`(1/m)`$.\ \varphi$	tuple abstraction
				$\mid \ \ldots$	
				$\mid$`(m/m)`$.\ \varphi$	
τ	::=	***Types***	ε	::=	***Locations***
	$\mid \alpha,\ \beta,\ \ldots$	variables		$\mid \delta$	variable
	$\mid [\eta]\tau$	array		$\mid \varepsilon.\varphi$	view
	$\mid \tau * \ldots * \tau$	tuple			

Figure 6.3: *Syntax of meta-objects that are manipulated by the compiler*

- *Generalization* — $\mathcal{G}$. Abstraction points quantify size, type and location variables that should considered abstract. A generalization $\mathcal{G}$ is a triplet $(\mathcal{G}^\eta, \mathcal{G}^\tau, \mathcal{G}^\varepsilon)$ where $\mathcal{G}^\eta$ (resp. $\mathcal{G}^\tau$) is a set of size (resp. type) variables and $\mathcal{G}^\varepsilon$ is a map from location variables to types, e.g., a set of typed location variables, denoted $\{\delta_1 : \tau_1, \ldots \delta_n : \tau_n\}$. These types $\tau_1, \ldots, \tau_n$ may use the abstract type and size variables introduced in $\mathcal{G}^\eta$ and $\mathcal{G}^\tau$. For instance, the generalization $(\{\nu\}, \{\alpha\}, \{\delta : [\nu]\alpha\})$ is valid.

- *Instantiation* — $\mathcal{I}$. Application points require concrete sizes, types and locations to instantiate the abstract ones. An instantiation $\mathcal{I}$ is a triplet $(\mathcal{I}^\eta, \mathcal{I}^\tau, \mathcal{I}^\varepsilon)$ where $\mathcal{I}^\eta$, $\mathcal{I}^\tau$ and $\mathcal{I}^\varepsilon$ are respectively size, type and location substitutions, i.e., mappings from variables to sizes, types and locations.

Generalizations serve as contexts for the terms or annotations that use sizes, types and locations. Multiple generalizations may be gathered using disjoint union — $\mathcal{G}_1 \uplus \ldots \uplus \mathcal{G}_n$ — provided that they do not intersect. If $\delta : \tau \in \mathcal{G}^\varepsilon$, we write $\mathcal{G}^\varepsilon(\delta)$ for the type τ of the location variable δ in $\mathcal{G}$. Similarly, given an instantiation $\mathcal{I}$, $\mathcal{I}^\eta(\nu)$, $\mathcal{I}^\tau(\tau)$ and $\mathcal{I}^\varepsilon(\delta)$ denote respectively the size, type and location associated to variable.

Location and types Since locations are typed, well-allocated programs are also well-typed. For that reason, we do not define a standalone type system. For locations to be well typed, they must be constructed from compatible projection functors: the location $\varepsilon.\varphi$ is valid only if ε has type τ and φ has type $\tau \to \tau'$. The *location typing* relation — $\mathcal{G} \vdash \varepsilon : \tau$ — formalizes these constraints:

Location type $\quad \mathcal{G} \vdash \varepsilon : \tau$

$$\textsc{L-Var}\ \frac{\mathcal{G}^\varepsilon(\delta) = \tau}{\mathcal{G} \vdash \delta : \tau} \qquad \textsc{L-Proj}\ \frac{\mathcal{G} \vdash \varepsilon : \tau \qquad \mathcal{G}^\eta\ ;\ \mathcal{C} \vdash \varphi :\ \tau \to \tau'}{\mathcal{G} \vdash \varepsilon.\varphi : \tau'}$$

The projection functor typing relation — $\Gamma\ ;\ \mathcal{C} \vdash \varphi :\ \tau \to \tau'$ — defined in Chapter 3 uses an environment that consists of a set of size variables, $\mathcal{G}^\eta$ in the present context, accompanied by a set of polynomial constraints $\mathcal{C}$, that guarantee the correctness of array accesses. Because these polynomial constraints are useless for allocation formalization, they are dropped and the constraint set $\mathcal{C}$ is existentially quantified in the L-Proj rule.

Substitutions The substitution of generalized variables $\mathcal{G}$ with an instantiation $\mathcal{I}$ whose domains match is defined as usual. For an object o (size, type, location, *etc*), it is denoted $o\,\{\mathcal{I}/\mathcal{G}\}$. An extra typing assumption is required: in an outer context $\mathcal{G}_o$, the type of substituted location variables must be consistent:

$$\forall\ \delta : \tau \in \mathcal{G}^\varepsilon,\ \mathcal{G}_o \vdash \mathcal{I}^\varepsilon(\delta)\ :\ \tau$$

Location information The syntax of MADL is implicitly typed: size, type and location annotations only introduce constraints. To present type and location checking, programs have to be supplemented with some typing and allocation information. For convenience, we do not introduce an explicitly typed and allocated version of MADL, but we provide the needed information through a typing oracle $\mathcal{O}$ whose contents is summed up in Figure 6.4.

The typing oracle associates typing information to some elements of programs: scopes, instantiations and declarations. It corresponds to the information that allocation inference retrieves. This monolithic view help to separate the syntactical elements, in particular the annotations, from the statically manipulated descriptions: sizes, types and locations. In the internal representation of our prototype, the typing and allocation information are stored locally, in a typed AST, which is convenient for compilation.

The typing oracle is a triplet $(\mathcal{O}_{inst}, \mathcal{O}_{decl}, \mathcal{O}_{scope})$ where $\mathcal{O}_{decl}$ associates each operator declaration with a generalization ($\mathcal{G}_d$), $\mathcal{O}_{inst}$ associates each instance of operator with an instantiation ($\mathcal{I}_i$) and $\mathcal{O}_{scope}$ contains the typing information for each scope (detailed below).

Typing Oracle ($\mathcal{O}$)				
	Operator Declaration ($\mathcal{O}_{decl}$)		Generalization ($\mathcal{G}_d$)	Size variables ($\mathcal{G}_d^\eta$)
				Type variables ($\mathcal{G}_d^\tau$)
				Loc. variables ($\mathcal{G}_d^\varepsilon$)
	Operator Instantiation ($\mathcal{O}_{inst}$)		Instantiation ($\mathcal{I}_i$)	Size substitution ($\mathcal{I}_i^\eta$)
				Type substitution ($\mathcal{I}_i^\tau$)
				Loc. substitution ($\mathcal{I}_i^\varepsilon$)
	Scope ($\mathcal{O}_{scope}$)	**Local Abstraction**	Instantiation ($\mathcal{I}_l$)	Size substitution ($\mathcal{I}_l^\eta$)
				Type substitution ($\mathcal{I}_l^\tau$)
				Loc. substitution ($\mathcal{I}_l^\varepsilon$)
			Generalization ($\mathcal{G}_l$)	Size variables ($\mathcal{G}_l^\eta$)
				Type variables ($\mathcal{G}_l^\tau$)
				Loc. variables ($\mathcal{G}_l^\varepsilon$)
		Existential Quantification	Generalization ($\mathcal{G}_e$)	Size variables ($\mathcal{G}_e^\eta$)
				Type variables ($\mathcal{G}_e^\tau$)
				Loc. variables ($\mathcal{G}_e^\varepsilon$)
		Variables ($\mathcal{X}$)	Annotations	Size var. def. ($\mathcal{X}^\eta$)
				Type var. def. ($\mathcal{X}^\tau$)
				Loc. var. def. ($\mathcal{X}^\varepsilon$)
			Expressions	Term var. def. ($\mathcal{X}^e$)

Figure 6.4: *The contents of the typing oracle*

The scopes introduce local size, type and location variables and they give to term and annotation variables their formal description. The $\mathcal{O}_{scope}$ field of the oracle defines for each scope a quadruplet $(\mathcal{I}_l, \mathcal{G}_l, \mathcal{G}_e, \mathcal{X})$ that contains:

- *Local abstraction.* Local abstract size, type and location variables stand for concrete sizes, types and locations, but ensure that the content of the scope does not rely on their actual values (see Section 2.4.4). Local abstraction may be understood as local polymorphism, where a term is generalized and immediately instantiated. Hence, it is defined by a generalization ($\mathcal{G}_l$) coupled with a compatible instantiation ($\mathcal{I}_l$).

- *Existential quantification.* Existentially quantified size, type and location variables resemble locally abstract ones. They are variables that are considered generic inside the scope. However, these variables are given no value outside the scope and hence must not appear in the interface. They are useful to describe varying length (e.g., using the iteration index as a size) or local memory locations. They are defined with a unique generalization ($\mathcal{G}_e$).

- *Program's variable typing.* Last, the scope defines the sizes, types and locations of meta-variables (used in annotations), both declared (x) and undeclared ($'x$). It also gives the locations of term variables.

For each scope, the various set of introduced variables, i.e., $\mathcal{G}_l$, $\mathcal{G}_e$ and $\mathcal{X}$, are ordered. Local abstract size and type variables ($\mathcal{G}_l$) may appear in the type of existentially quantified location variables ($\mathcal{G}_e^{\varepsilon}$) and all of them may be used in the declaration of program variables ($\mathcal{X}$).

Accesses to the oracle are uniformly denoted by the component, the related syntactical object and the desired information: $\mathcal{O}_{scope}(b).\mathcal{X}^e(x)$ designates the location of the variable x in block b, similarly $\mathcal{O}_{inst}(\texttt{f}).\mathcal{I}_i$ denotes the instantiation of the type scheme of operator `f`.[1]

Annotations Let us emphasize the distinction between the source *annotations* and the *formal* meta-information. The former describe partial constraints on sizes, types and locations, in particular, some part may remain unspecified, using a placeholder (_). The latter are complete descriptions of this meta-information. To recall the difference, annotations are denoted with Latin letters while formal sizes, types and locations uses Greek letters.

In both worlds, the *abstract* sizes, types and locations refer to generic variables that are universally quantified, while the *concrete* sizes, types and locations are the objects of the size, type and location language (either annotations or formal ones). We summarize these elements in the following table.

	Annotation	Formal Representation
Abstract (variables)	n a v	ν α δ
Concrete (objects)	$2 * s$ $[s]\,t$ `transpose` (d)	$2 * \eta$ $[\eta]\,\tau$ ε.`transpose` $\eta\ \eta\ \tau$

Size and type annotations have the same structure as their formal counterparts. The situation is different for locations: annotations use operators instances whereas formal locations are made of projection functors.

6.1.2 A Type System for Locations

In the following discussion, we use the vector notation — $\vec{o}$ — to denote multiple sizes, types or locations. We recall that vectors can be concatenated and extended. This is denoted in a unified way: $\vec{o}, p, \vec{q}$ is the vector $o_1, \ldots, o_n, p, q_1, \ldots, q_m$.

Expressions describe transition functions that expect multiple arguments and produce multiple results using an internal state. The state can be viewed as implicit argument (the current state) and an implicit result (the new state). This highlights a key restriction: although handled locally, the state cannot capture any of the existentially quantified variables, since they would escape their scope. For this reason, locations that are part of the state appear in the location signature of expressions.

Definition 6.1 (Location signature)**.** A *location signature* θ is a triple $\vec{i} \xrightarrow{\vec{s}} \vec{o}$ where $\vec{i}$, $\vec{s}$ and $\vec{o}$ are respectively the input, state and output location vectors — $\varepsilon, \ldots, \varepsilon$ —.

The case of empty inputs or outputs are denoted with `()`. By contrast, inexistant state locations are simply omitted. A stateless expression that expects no arguments will have a signature of the form `()` $\longrightarrow \varepsilon, \ldots, \varepsilon$ and likewise for an empty output list: $\varepsilon, \ldots, \varepsilon \longrightarrow$ `()`.

Definition 6.2 (Allocation scheme)**.** An *allocation scheme* σ is a triple $\forall\, \mathcal{G}.\ «\vec{\eta}»\ \theta$ where $\mathcal{G}$ is a generalization, $«\vec{\eta}»$ are size parameters and θ is a location signature.

Allocation schemes are polymorphic location signatures. The additional size parameters — $«\vec{\eta}»$ — are used to insert constraints on abstract sizes that could not be deduced otherwise. These allocation schemes are instantiated by providing concrete sizes, types and locations for the abstract ones and the size parameters.

[1] We suppose that the multiple occurrences of the same operator are distinguished in some way.

Definition 6.3 (Scheme instantiation)**.** An allocation scheme σ is *instantiated* — $\theta := \sigma\,[\mathcal{I}]\,\langle\!\langle \vec{\eta} \rangle\!\rangle$ — with an instantiation $\mathcal{I}$ and sizes $\langle\!\langle \vec{\eta} \rangle\!\rangle$ to build a location signature θ that is defined by the following rule:

$$\textsc{Inst}\ \frac{\vec{\eta'} = \vec{\eta}\,\{\mathcal{I}/\mathcal{G}\} \qquad \theta' = \theta\,\{\mathcal{I}/\mathcal{G}\}}{\theta' := \left(\forall\,\mathcal{G}.\ \langle\!\langle \vec{\eta} \rangle\!\rangle\ \theta\right)\,[\mathcal{I}]\,\langle\!\langle \vec{\eta'} \rangle\!\rangle}$$

We recall that the substitution imposes that instantiation and generalization be compatible. Allocation scheme instantiation succeeds only if the provided sizes match the declared size parameters.

Environment Annotations and expressions are typed and located in a context $\mathcal{G}, \Gamma$ that consists of the bound size, type and location variables — $\mathcal{G}$ — alongside an *environment* Γ that describes identifiers and their scopes. Their syntax is given below:

			Environment
Γ	::=		
		$\mathcal{F}$	Operators
	|	$\Gamma \cdot \mathcal{X}$	Resumed scope
	|	$\Gamma \diamond \mathcal{X}$	Restarted scope

Environments thus consist of a list $\mathcal{F} : \mathcal{X} : \ldots : \mathcal{X}$ where ':' is either '$\diamond$' or '$\cdot$'. They start with a description of available operators — $\mathcal{F}$ — that maps operator identifiers to allocation schemes: $\{\texttt{f1} : \sigma_1, \ldots\}$. This set of operators is completed by *scopes* — $\mathcal{X}$ — as defined in the content of the oracle, that associate size, type, location or term variables with a size, type or location. Restarted scopes are separated from the outer ones with a '$\diamond$' marker whereas other scopes are separated with a '$\cdot$' marker.

The various accesses to the environment are denoted in a similar way to accesses to the typing oracle: $\Gamma^{\texttt{f}}(\texttt{f})$ is the signature of the operator `f`, $\Gamma^{e}(x)$ is the location of the variable x. For annotation variables, this notation is extended to meta-variables (v) by returning an arbitrary size, type and location for anonymous variables, as follows:

$$\Gamma^{\eta}(\_) = \eta \qquad \Gamma^{\tau}(\_) = \tau \qquad \Gamma^{\varepsilon}(\_) = \varepsilon \qquad \text{(anonymous)}$$
$$\Gamma^{\eta}('x) = \Gamma^{\eta}(x) \qquad \Gamma^{\tau}('x) = \Gamma^{\tau}(x) \qquad \Gamma^{\varepsilon}('x) = \Gamma^{\varepsilon}(x) \qquad \text{(undeclared)}$$

Annotation checking All the correctness judgments assume that expressions or annotations are well-formed, i.e., that their free variables are part of the environment. To relate annotations to their formal description, a size relation — $\mathcal{G}, \Gamma \vdash s : \eta$ — associates each size annotation with a formal size. Similarly, a type relation — $\mathcal{G}, \Gamma \vdash t : \tau$ — links type annotations and formal types. They are defined in Figure 6.5: the rules are unsurprising and mirror the structure of annotations in the formal sizes and types.

Size annotation checking — $\mathcal{G}, \Gamma \vdash s : \eta$

$$\textsc{S-DataV}\ \frac{\Gamma^{\eta}(v) = \eta}{\mathcal{G}, \Gamma \vdash v : \eta} \qquad \textsc{S-Imm}\ \frac{}{\mathcal{G}, \Gamma \vdash n : n} \qquad \textsc{S-Add}\ \frac{\mathcal{G}, \Gamma \vdash s_1 : \eta_1 \qquad \mathcal{G}, \Gamma \vdash s_2 : \eta_2}{\mathcal{G}, \Gamma \vdash s_1 + s_2 : \eta_1 + \eta_2}$$

$$\textsc{S-Sub}\ \frac{\mathcal{G}, \Gamma \vdash s_1 : \eta_1 \qquad \mathcal{G}, \Gamma \vdash s_2 : \eta_2}{\mathcal{G}, \Gamma \vdash s_1 - s_2 : \eta_1 + (-1) * \eta_2} \qquad \textsc{S-Mul}\ \frac{\mathcal{G}, \Gamma \vdash s_1 : \eta_1 \qquad \mathcal{G}, \Gamma \vdash s_2 : \eta_2}{\mathcal{G}, \Gamma \vdash s_1 * s_2 : \eta_1 * \eta_2}$$

Type annotation checking — $\mathcal{G}, \Gamma \vdash t : \tau$

$$\textsc{T-DataV}\ \frac{\Gamma^{\tau}(v) = \tau}{\mathcal{G}, \Gamma \vdash v : \tau} \qquad \textsc{T-Int}\ \frac{}{\mathcal{G}, \Gamma \vdash \texttt{int} : \texttt{int}} \qquad \textsc{T-Bool}\ \frac{}{\mathcal{G}, \Gamma \vdash \texttt{bool} : \texttt{bool}}$$

$$\textsc{T-Array}\ \frac{\mathcal{G}, \Gamma \vdash s : \eta \qquad \mathcal{G}, \Gamma \vdash t : \tau}{\mathcal{G}, \Gamma \vdash [s]t : [\eta]\tau} \qquad \textsc{T-Tuple}\ \frac{\mathcal{G}, \Gamma \vdash t_1 : \tau_1 \qquad \ldots \qquad \mathcal{G}, \Gamma \vdash t_n : \tau_n}{\mathcal{G}, \Gamma \vdash t_1 * \ldots * t_n : \tau_1 * \tau_n}$$

Figure 6.5: *Annotation checking*

Expression checking So as to handle array iteration, the rules for expression checking require an additional bit of information: the current context of iteration, denoted k. It is either $\varnothing$, i.e., no iteration, or $\iota < \eta$ to indicate an iteration of size η where the existentially quantified size

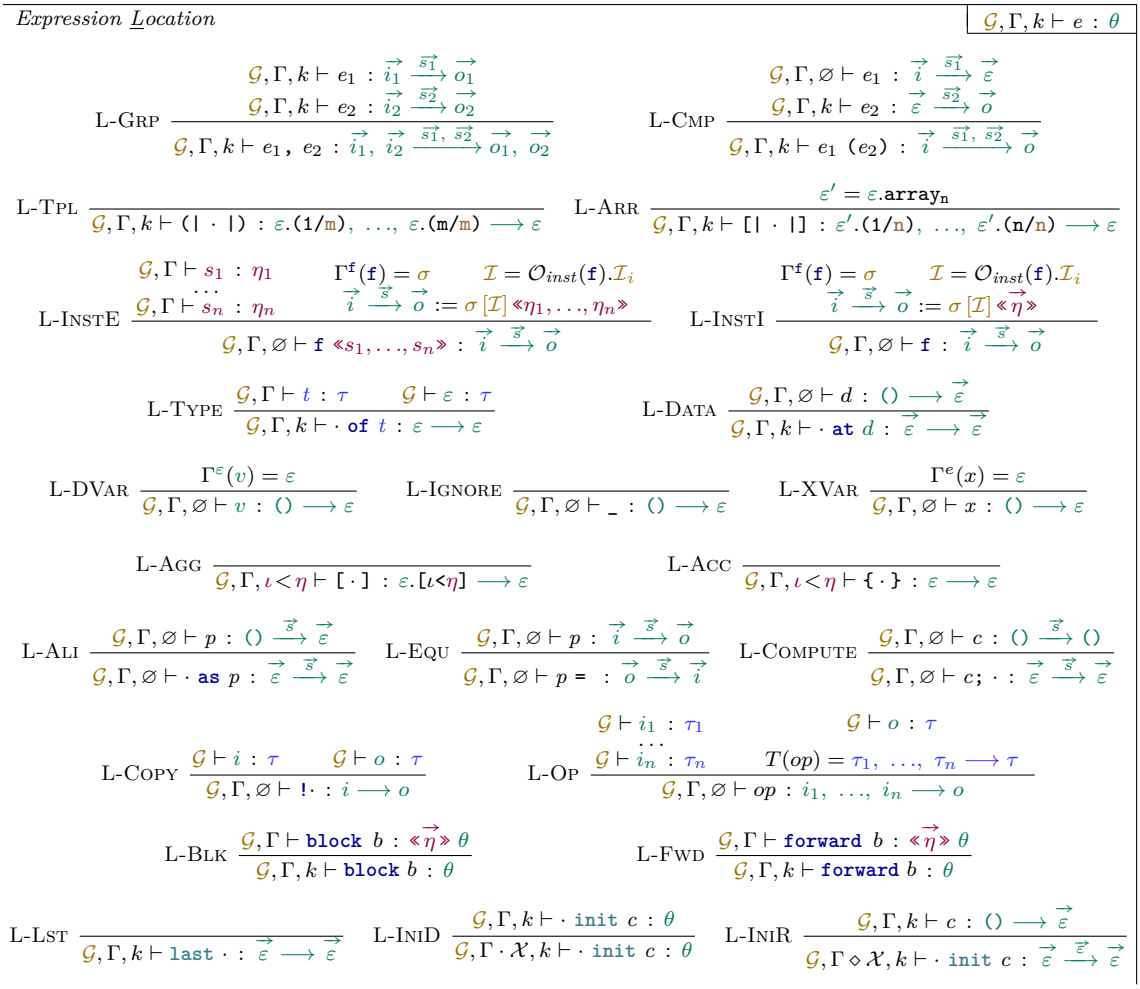


Figure 6.6: *Expression signature checking*

variable that correspond to the index is ι (see Section 5.1.3). The expression checking judgment —$\mathcal{G}, \Gamma, k \vdash e : \theta$— specifies the location signature for expressions, which are either patterns, computations or location annotations. It reads: *with formal variables* $\mathcal{G}$*, in environment* Γ *and iteration context* k*, expression* e *has location signature* θ.

The definition of the relation is given in Figure 6.6. We have separated the hairy rules for the block construct —$\mathcal{G}, \Gamma \vdash \texttt{block}\ b : \sigma$— and the forward construct —$\mathcal{G}, \Gamma \vdash \texttt{forward}\ b : \sigma$—, that are given below.

For the groups construct, rule L-Grp only gathers the inputs, the outputs and the states. For composition (rule L-Cmp), the locations of connected flows must match. Because the iteration context describe only one dimension of iteration, it is only forwarded to the second expression: this prevents connected aggregation or accumulation, e.g., `[·] ([·])`, that correspond to nested aggregations or accumulations, e.g., `[[·]]`. Multi-level clauses are elaborated to simple iteration clauses prior to allocation inference (see Section 5.3.2).

The rule L-Tpl relates the memory location of each component to that of the result: we use the abbreviation $\varepsilon.\texttt{(k/m)}$ to denote the location $\varepsilon.\big(@\texttt{(k/m)}.\ \mathbf{1}_\tau\big)$, i.e., the projection of δ on the $\texttt{k}^{\text{th}}$ component. We provide a similar rule L-Arr for the immediate array construct by viewing the array as a tuple with the $\texttt{array}_\texttt{n}$ projection functor defined below. This functor allows to view an array of fixed size `n` (an integer value) as a tuple of arity `n`.

$$\texttt{array}_\texttt{n}\ \alpha = \begin{array}{l} \lambda\texttt{(1/n)}.\ @\texttt{[1-1<n]}.\ \mathbf{1}_\alpha \\ \texttt{|}\quad\quad\quad\quad \ldots \\ \texttt{|\,(n/n)}.\ @\texttt{[n-1<n]}.\ \mathbf{1}_\alpha \end{array}$$

At instantiation, the size parameters of operators are either specified (rule L-InstE) or implicit (rule L-InstI). In both cases, the allocation scheme of the operator is instantiated with the instantiation given by the oracle: $\mathcal{O}_{inst}(\texttt{f}).\mathcal{I}_i$.

The aggregation (rule L-Agg) and accumulation (rule L-Acc) constructs are only possible in iteration contexts ($k = \iota < \eta$). For aggregation, the location of the input — $\varepsilon.[\iota<\eta]$ — is the ι^{th} element at the location of the output (the aggregated array), where ι denotes an abstract size that correspond to the index. As for the rules L-Tpl and L-Arr, $\varepsilon.[\iota<\eta]$ denotes the location $\varepsilon.\big(@[\iota<\eta]\,.\,\mathbf{1}_\tau\big)$.

Copies (rule L-Copy) and operations (rule L-Op) write new values. Their location is only constrained by the types: $T(op)$ denotes the type of operation op.

The last three rules deal with the state. The inputs and outputs of `last` are at the same locations: this will induce a scheduling constraint so that the old values must be read before writing the new ones (see Section 7.1). The handling of `init` has two stages: first the non-restarted scopes are dropped in the L-IniD rule. Once a restarted scope has been found, the initialization expression is checked with the rule L-IniR. It must produce its results, without any arguments or state, at the locations of the expression that will be connected to the `init`. This rule introduces the locations of the state in the signature.

Scope checking Scopes — $i\colon\ c\ r$ — are introduced by operator bodies, local blocks and `forward` iterators. They are associated with a reset condition — r. The *reset checking* judgment — $\mathcal{G}, \Gamma \vdash r$ — states the correction of reset conditions. It is trivial for the `resume` and `restart` conditions. For the remaining case, it ensures that the dynamic condition has a boolean type:

Reset condition checking — $\mathcal{G}, \Gamma \vdash r$

$$\text{R-ESUME}\ \frac{}{\mathcal{G}, \Gamma \vdash \texttt{resume}} \qquad \text{R-ESET}\ \frac{\mathcal{G}, \Gamma, \varnothing \vdash c : () \longrightarrow \varepsilon \qquad \mathcal{G} \vdash \varepsilon : \texttt{bool}}{\mathcal{G}, \Gamma \vdash \texttt{reset}\ c} \qquad \text{R-ESTART}\ \frac{}{\mathcal{G}, \Gamma \vdash \texttt{restart}}$$

Scope interfaces — $i = a_1\ \ldots\ a_p\ \text{«}s_1, \ldots, s_n\text{»}\ (p)\ \texttt{returns}\ (c)$ — introduce abstract parameters: $a_1\ \ldots\ a_p$. These scoped annotation variables must be associated with size type and location variables of the same scope. This is formally defined by the *abstract parameter checking* judgment — $\mathcal{G}_l, \mathcal{G}_e, \mathcal{X} \vdash a$ — below.

Abstract parameter checking — $\mathcal{G}_l, \mathcal{G}_e, \mathcal{X} \vdash a$

$$\text{A-S}\ \frac{x_1 : \nu_1 \in \mathcal{X}^\eta \quad \nu_1 \in \mathcal{G}_l^\eta \quad \ldots \quad x_n : \nu_n \in \mathcal{X}^\eta \quad \nu_n \in \mathcal{G}_l^\eta}{\mathcal{G}_l, \mathcal{G}_e, \mathcal{X} \vdash (\texttt{size}\ x_1\ \ldots\ x_n)} \qquad \text{A-T}\ \frac{x_1 : \alpha_1 \in \mathcal{X}^\tau \quad \alpha_1 \in \mathcal{G}_l^\tau \quad \ldots \quad x_n : \alpha_n \in \mathcal{X}^\tau \quad \alpha_n \in \mathcal{G}_l^\tau}{\mathcal{G}_l, \mathcal{G}_e, \mathcal{X} \vdash (\texttt{type}\ x_1\ \ldots\ x_n)}$$

$$\text{A-D}\ \frac{x_1 : \delta_1 \in \mathcal{X}^\varepsilon \quad \delta_1 : \tau_1 \in \mathcal{G}_e^\varepsilon \quad \ldots \quad x_n : \delta_n \in \mathcal{X}^\varepsilon \quad \delta_n : \tau_n \in \mathcal{G}_e^\varepsilon}{\mathcal{G}_l, \mathcal{G}_e, \mathcal{X} \vdash (\texttt{data}\ x_1\ \ldots\ x_n)}$$

The rule A-S (resp. A-T) ensures that abstract sizes (resp. types) actually represent local abstract sizes (resp. types) of the scope ($\mathcal{G}_l$) while the A-D rule ensures that abstract locations represent existentially quantified locations ($\mathcal{G}_a$). This difference is due to the usage of this meta-information: abstract sizes and types represent sizes and types that we do not need to know about, while abstract locations intuitively represent local memory, which should not be accessed outside its scope. Existentially quantified sizes are useful as well: they account for a size whose value is not defined by a polynomial, e.g., the current index.

Blocks We can now define the *block checking relation* $\mathcal{G}, \Gamma \vdash \texttt{block}\ i\colon\ c\ r\ :\ \text{«}\vec{\eta}\text{»}\ \theta$. It associates to blocks some size parameters and a location signature.

Scope checking (`block`) — $\mathcal{G}, \Gamma \vdash \texttt{block}\ i\colon\ c\ r\ :\ \text{«}\vec{\eta}\text{»}\ \theta$

$$i = a_1\ \ldots\ a_p\ \text{«}s_1, \ldots, s_n\text{»}\ (p)\ \texttt{returns}\ (c) \qquad \mathcal{G}, \Gamma \vdash r$$

$$(\mathcal{I}_l, \mathcal{G}_l, \mathcal{G}_e, \mathcal{X}) = \mathcal{O}_{scope}(b) \qquad \Gamma' = \begin{cases} \Gamma \diamond \mathcal{X} & \text{if } r = \texttt{restart} \\ \Gamma \cdot \mathcal{X} & \text{otherwise} \end{cases}$$

$$\mathcal{G}' = \mathcal{G} \uplus \mathcal{G}_l \uplus \mathcal{G}_e$$

$$\text{Blk}\ \frac{\mathcal{G}_l, \mathcal{G}_e, \mathcal{X} \vdash a_1 \quad \ldots \quad \mathcal{G}_l, \mathcal{G}_e, \mathcal{X} \vdash a_p \qquad \mathcal{G}', \Gamma' \vdash s_1 : \eta_1 \quad \ldots \quad \mathcal{G}', \Gamma' \vdash s_n : \eta_n \qquad \mathcal{G}', \Gamma', \varnothing \vdash p : () \xrightarrow{\vec{s_i}} \vec{i} \quad \mathcal{G}', \Gamma', \varnothing \vdash c : () \xrightarrow{\vec{s_o}} \vec{o} \quad \mathcal{G}', \Gamma', \varnothing \vdash c' : () \xrightarrow{\vec{s}} ()}{\mathcal{G}, \Gamma \vdash \texttt{block}\ i\colon\ c'\ r\ :\ (\text{«}\eta_1, \ldots, \eta_n\text{»}\ \theta)\,\{\mathcal{I}_l/\mathcal{G}_l\}}$$

$$\theta = \begin{cases} \vec{i} \longrightarrow \vec{o} & \text{if } r = \texttt{restart} \\ \vec{i} \xrightarrow{\vec{s_i},\ \vec{s},\ \vec{s_o}} \vec{o} & \text{otherwise} \end{cases}$$

The reset condition — r — is checked in the outer context. The inner context of the block — $\mathcal{G}', \Gamma', \varnothing$ — is built by adding the local abstract variables — $\mathcal{G}_l$ —, the existentially quantified ones — $\mathcal{G}_e$ — and registering the new program variables and meta-variables — $\mathcal{X}$ —, separated by a restart marker $\diamond$ if applicable. The abstract parameters — $a_1\ \ldots\ a_n$ — must be associated to abstract variables, as defined above. One condition is missing: they must represent distinct size,

type and location variables. The size parameters — $\text{«}\eta_1, ..., \eta_n\text{»}$ —, the argument — p —, the result — c — and the internal computation — c' — are checked in the extended context.

The location signature θ depends on the reset condition: if restarted, only the argument and result locations appear, i.e., the state is dropped, otherwise, the state locations are collected. To build the final signature, the local abstract variables are substituted with their instantiation in $\text{«}\eta_1, ..., \eta_n\text{»}\ \theta$. The existentially quantified variables cannot appear in these sizes or in the signature. In particular, unless the block is restarted, they may not be used in the state.

Iteration The *iteration checking relation* — $\mathcal{G}, \Gamma \vdash \texttt{forward}\ i\texttt{:}\ c\ r\ \texttt{:}\ \text{«}\vec{\eta}\text{»}\ \theta$ — is similar. For simplicity, we suppose that the interface of the `forward` iterator is complete, i.e., it has a single size parameter — $\text{«}\eta\text{»}$ — and a double index capture — $[ix\ \texttt{:}\ \texttt{size}\ is]$ — as a term variable (ix) and a size variable (is).

Scope checking (`forward`) — $\mathcal{G}, \Gamma \vdash \texttt{forward}\ i\texttt{:}\ c\ r\ \texttt{:}\ \text{«}\vec{\eta}\text{»}\ \theta$

$$i = a_1\ \ldots\ a_p\ \text{«}s\text{»}\ [ix\ \texttt{:}\ \texttt{size}\ is]\ (p)\ \texttt{returns}\ (c) \qquad \mathcal{G}, \Gamma \vdash r$$

$$(\_, \mathcal{G}_l, \mathcal{G}_e, \mathcal{X}) = \mathcal{O}_{scope}(b) \qquad \Gamma' = \begin{cases} \Gamma \diamond \mathcal{X} & \text{if } r = \texttt{restart} \\ \Gamma \cdot \mathcal{X} & \text{otherwise} \end{cases} \qquad is : \iota \in \mathcal{X}^\eta \qquad \iota \in \mathcal{G}_e^\eta$$

$$\mathcal{G}' = \mathcal{G} \uplus \mathcal{G}_l \uplus \mathcal{G}_e \qquad ix : \delta \in \mathcal{X}^e \qquad \delta : [\eta] \in \mathcal{G}_e^\varepsilon$$

$$\text{FWD}\ \frac{\begin{array}{c}\mathcal{G}_l, \mathcal{G}_e, \mathcal{X} \vdash a_1 \\ \ldots \\ \mathcal{G}_l, \mathcal{G}_e, \mathcal{X} \vdash a_p\end{array} \qquad \mathcal{G}', \Gamma' \vdash s\ \texttt{:}\ \eta \qquad \begin{array}{l}\mathcal{G}', \Gamma', \iota < \eta \vdash p\ \texttt{:}\ \vec{i_b} \xrightarrow{\vec{s_i}} \vec{i} \\ \mathcal{G}', \Gamma', \iota < \eta \vdash c\ \texttt{:}\ \vec{o_b} \xrightarrow{\vec{s_o}} \vec{o} \\ \mathcal{G}', \Gamma', \varnothing \vdash c'\ \texttt{:}\ \vec{i_b} \xrightarrow{\vec{s}} \vec{o_b}\end{array} \qquad \theta = \begin{cases} \vec{i} \longrightarrow \vec{o} & \text{if } r = \texttt{restart} \\ \vec{i} \xrightarrow{\vec{s_i}, \vec{s}, \vec{s_o}} \vec{o} & \text{otherwise} \end{cases}}{\mathcal{G}, \Gamma \vdash i\texttt{:}\ c'\ r\ \texttt{:}\ (\text{«}\eta\text{»}\ \theta)\ \{\mathcal{I}_l/\mathcal{G}_l\}}$$

Let us review the differences between the rules for `block` and `forward`. The size index — is — is associated to an existentially quantified size — $\iota \in \mathcal{G}_e$ — and the expression index — ix — is associated to an existentially quantified location — $\delta : [\eta] \in \mathcal{G}_e$ — with type $[\eta]$, i.e., an index whose size is the number of iterations.

Contrary to blocks, the arguments and results of `forward` iterations are checked in an iteration context — $\iota < \eta$ —, that allows for accumulation and aggregation constructs. The arguments and results may have inputs: they define the signature of the internal computation c'.

The rules for accumulation (L-Acc) and iteration (Fwd) do not account for the partial aliases presented in Section 5.2. We sketch how partial aliases are supported in Section 6.2.5.

Operators Lastly, the *top-level declaration* judgment — $\mathcal{F} \vdash \texttt{let}\ \ td \rightsquigarrow \mathcal{F}$ — relates a potentially recursive definition of operators to some preceding and resulting operator contexts. It is defined as follows:

Top-level declaration checking — $\mathcal{F} \vdash \texttt{let}\ \ td \rightsquigarrow \mathcal{F}$

$$\mathcal{F}'' = \begin{cases} \mathcal{F}' & \text{if } \texttt{rec} \\ \mathcal{F} & \text{otherwise} \end{cases}$$

$$\frac{\begin{array}{ll}\mathcal{G}_1 = \mathcal{O}_{decl}(\texttt{f}_1).\mathcal{G}_d & \mathcal{G}_1, \mathcal{F}'' \diamond \emptyset \vdash \texttt{block}\ b_1\ \texttt{:}\ \text{«}\vec{\eta_1}\text{»}\ \theta_1 \\ \ldots & \ldots \\ \mathcal{G}_n = \mathcal{O}_{decl}(\texttt{f}_n).\mathcal{G}_d & \mathcal{G}_n, \mathcal{F}'' \diamond \emptyset \vdash \texttt{block}\ b_n\ \texttt{:}\ \text{«}\vec{\eta_n}\text{»}\ \theta_n\end{array} \qquad \mathcal{F}' = \mathcal{F} \left\{ \begin{array}{l}\texttt{f}_1 \mapsto \forall \mathcal{G}_1.\ \text{«}\vec{\eta_1}\text{»}\ \theta_1 \\ \ldots \\ \texttt{f}_n \mapsto \forall \mathcal{G}_n.\ \text{«}\vec{\eta_n}\text{»}\ \theta_n\end{array} \right\}}{\mathcal{F} \vdash \texttt{let}\ [\texttt{rec}]\ [k_1]\ \texttt{f}_1\ b_1\ \texttt{and}\ \ \ldots\ \ \texttt{and}\ [k_n]\ \texttt{f}_n\ b_n \rightsquigarrow \mathcal{F}'}$$

The typing of declaration bodies is initiated with an empty restarted environment — $\mathcal{F} \diamond \emptyset$ —. This allows for initialization in the node, since their is a restarted marker in the environment. However, it imposes that initialization expression to be given a location in an empty environment.

Do scalar values need locations?

Indeed, scalar value are mostly free to copy (this will be managed by C compiler register allocation). However, they can be part of compound data-structures that must be tracked.

6.1.3 Examples

Built-ins We gave in the preamble the location scheme of some built-in operators. There are two kinds of primitives, whether they can be described directly with the projection functors introduced in Section 3.3.2 or not. The first kind take an argument at an unconstrained location and build a view for their result:

$$
\begin{aligned}
\texttt{reverse}: &\ \forall\nu.\,\forall\alpha.\,\forall\delta{:}[\nu]\alpha.\,\delta \longrightarrow \delta.\texttt{reverse}\ \nu\ \alpha\\
\texttt{transpose}: &\ \forall\nu\,\mu.\,\forall\alpha.\,\forall\delta{:}[\nu]\,[\mu]\alpha.\,\delta \longrightarrow \delta.\texttt{transpose}\ \nu\ \mu\ \alpha\\
\texttt{window}: &\ \forall\nu\,\kappa.\,\forall\alpha.\,\forall\delta{:}[\nu+\kappa-1]\alpha.\,\text{«}\kappa\text{»}\ \delta \longrightarrow \delta.\texttt{window}\ \nu\ \kappa\ \alpha\\
\texttt{sample}: &\ \forall\nu\,\kappa.\,\forall\alpha.\,\forall\delta{:}[\nu*\kappa-\kappa+1]\alpha.\,\text{«}\kappa\text{»}\ \delta \longrightarrow \delta.\texttt{sample}\ \nu\ \kappa\ \alpha\\
\texttt{split}: &\ \forall\nu\,\kappa.\,\forall\alpha.\,\forall\delta{:}[\nu*\kappa]\alpha.\,\text{«}\kappa\text{»}\ \delta \longrightarrow \delta.\texttt{split}\ \nu\ \kappa\ \alpha\\
\texttt{repeat}: &\ \forall\nu.\,\forall\alpha.\,\forall\delta{:}\alpha.\,\text{«}\nu\text{»}\ \delta \longrightarrow \delta.\texttt{repeat}\ \nu\ \alpha
\end{aligned}
$$

For the second kind, operator type schemes use projection functors that describe the inverse operation. Then, the location of the output is unconstrained while the locations of inputs are built out of views:

$$
\begin{aligned}
\texttt{concat}: &\ \forall\nu\,\mu.\,\forall\alpha.\,\forall\delta{:}[\nu+\mu]\alpha.\,\text{«}\nu,\mu\text{»}\begin{cases}\delta.\texttt{cut}\ \nu\ \mu\ \alpha.(1/2)\\ \delta.\texttt{cut}\ \nu\ \mu\ \alpha.(2/2)\end{cases}\longrightarrow\delta\\
\texttt{flatten}: &\ \forall\nu\,\kappa.\,\forall\alpha.\,\forall\delta{:}[\nu*\kappa]\alpha.\,\text{«}\kappa\text{»}\ \delta.\texttt{sample}\ \nu\ \mu\ \alpha\longrightarrow\delta
\end{aligned}
$$

This dichotomy is central for the inference process, which is presented in Section 6.2.

Data structures Data structures (tuples and arrays) impose conditions on the locations of their components. The following operator aims at building an array of size 2 that contains x for each of its elements:

```
1 let dupl (x) returns (a): a = [| x,x |]
```

```
Error: Invalid locations
```

This program is incorrect because it induces the following location constraint: $\delta.[0<2] = \delta.[1<2]$. Depending on the objectives, this error can be solved in various ways:

```
1 let dupl (x) returns (a):
2   a = [| x,!x |]
```

```
val dupl: type a. a → [2]a
```

$$\forall\alpha.\,\forall\delta{:}[2]\alpha.\,\delta.[0<2] \longrightarrow \delta$$

```
1 let dupl (x) returns (a):
2   a = [| x,x |] at repeat (_)
```

```
val dupl: type a. a → [2]a
```

$$\forall\alpha.\,\forall\delta{:}\alpha.\,\delta \longrightarrow \delta.\texttt{repeat}\ 2\ \alpha$$

The left version introduces a copy — `!`· —, hence, the location of the second element of the immediate array is no longer equal to the location of the first one. In the right version, we add a location annotation — · `at repeat (_)` — that specifies that the array is indeed made of a unique element.

The type schemes of both versions are identical, but the location schemes are different.[2] For the copy version (on the left), the argument must be the first element of the array of size 2 that contains the result. This operator simulates in a declarative way a function that completes a partially initialized value. For the annotated version, the returned location describes a fictional array: all its elements are mapped to the argument. Combining the two solutions would lead to an error of another kind, that we discuss in Section 7.1.

Incompleteness of location inference The inference of locations is not complete, as the above example shows: adding a location annotation makes the inference succeed. Here is another example where the location constraints cannot be solved without help:

```
1 let parts (a,b,c) returns (a ++ b, b ++ c)
```

```
Error: Invalid locations
```

The troublesome constraint arises from the use of b in both (unrelated) concatenations:

$$\delta_1.\texttt{cut}\ \nu_1\ \nu_2\ \alpha.(2/2) = \delta_2.\texttt{cut}\ \nu_2\ \nu_3\ \alpha.(1/2)$$

[2] The compiler does generate location signatures, but they are barely readable. Hence we here transcribe manually the compiler's output.

We present in Section 6.2 how annotations may be inserted to guide the constraint solving process. For this operator, we would expect the following location scheme, that embeds a, b and c in a unique array whose size is the sum of the sizes of the arguments.

$$\forall\,\nu_1\,\nu_2\,\nu_3.\ \forall\,\alpha.\ \forall\,\delta : [\nu_1+\nu_2+\nu_3]\,\alpha.\ \begin{cases}\delta.(\lambda[i{<}\nu_1].\,@[i{<}\nu_1+\nu_2+\nu_3].\,\mathbf{1}_\alpha)\\ \delta.(\lambda[i{<}\nu_2].\,@[i+\nu_1{<}\nu_1+\nu_2+\nu_3].\,\mathbf{1}_\alpha)\\ \delta.(\lambda[i{<}\nu_3].\,@[i+\nu_1+\nu_2{<}\nu_1+\nu_2+\nu_3].\,\mathbf{1}_\alpha)\end{cases} \longrightarrow \begin{cases}\delta.(\lambda[i{<}\nu_1+\nu_2].\,@[i{<}\nu_1+\nu_2+\nu_3].\,\mathbf{1}_\alpha)\\ \delta.(\lambda[i{<}\nu_2+\nu_3].\,@[i+\nu_1{<}\nu_1+\nu_2+\nu_3].\,\mathbf{1}_\alpha)\end{cases}$$

Abstract sizes and locations Scopes contain abstract size, type and location variables. In well-typed programs, these variables cannot be used outside of their scope. We refer the reader to Section 5.1.3 for an example of an operator (`triangular`) that is ill-typed because a size escape its scope through a type. For locations, similar errors arise in two ways: either by making location variables escape, or by making size variables escape through the locations. These errors happen on well-typed terms. For instance, the `iota` operator presented in Section 5.1.3 would produce an error without copy:

```
let func iota «n» () returns (a):
  forward «n» [i] returns (a = [i])
```

Incorrect program:
abstract location would escape its scope

The intuition is that since `iota` produces a value, it contains some computations. The formal reason comes from the location of the index i. As specified in the FWD rule, the index must be given a location δ_i such that $\delta_i : [\eta] \in \mathcal{G}_e^{\varepsilon}$, i.e., δ_i is an existentially quantified location variable and hence cannot escape the scope of the iteration, as it would through a. A correct implementation can be found in Section 5.1.3.

There is a second cause of errors in such an operator. To illustrate it, we propose to redefine a `repeat` operator. It builds an array as a view of a single element, i.e., without duplicating its value.

```
let func my_repeat «n» (x) returns (a):
  forward «n» returns (a = [x])
```

Incorrect program:
abstract size would escape its scope

By denoting δ_x and δ_{ai} the locations of x and ai, respectively, ν the value of the size parameter «n» and ι the existentially quantified size that represents the index, the following constraint arises from the above definition: $\delta_x = \delta_a.[\iota{<}\nu]$. However, the index size ι would escape its scope since ι is part of the scope of the `forward` iterator whereas δ_x is part of the outermost scope. As presented in Section 6.1.3, adding a location annotation — a `at repeat (_)` — would allow allocation to succeed.

6.2 Elaboration

In MADL, a direct description of memory locations is impossible because projection functors have no concrete syntax — a deliberate limitation that ensures that the available functors are correct. Memory allocation must thus be deduced from terms.

Inference and elaboration In the context of ML, Pottier [Pot14] clearly stated the distinction between type *inference* and type *elaboration*. The former aims at deciding whether a typing derivation exists, whereas the latter reconstructs missing type information. For instance, the untyped λ-calculus — $M, N \;::=\; x \mid \lambda x.\ M \mid M\ N$ — may be elaborated into its explicitly typed version: — $M, N \;::=\; x \mid \lambda x : \tau.\ M \mid M\ N$ — where type annotations are only added to abstractions.

Similarly, the explicit type information of MADL programs is confined to a few constructs — the domain of the typing oracle: instantiations, declarations and scopes — that are arbitrarily scattered in the terms. The reconstruction algorithms are structured as follows:

1. *Constraint generation.* The terms are traversed to allocate formal variables that represent the information to be reconstructed (e.g., types, sizes, *etc*), as well as constraints between these variables (e.g., equality, sub-typing, *etc*).

2. *Constraint solving.* The collected constraints are partially or fully solved. This procedure depends on the formal representation of types and constraints only. In particular, it is independent of the terms from which the constraints are extracted.

3. *Term elaboration.* The solution of the system must be injected in the terms to add the inferred information. This step requires to traverse terms again to produce an annotated term with a similar structure.

The two term traversals have the same structure but they need to run separately, as because constraint solving is a global process, it needs all the constraints. To avoid code duplication, Pottier [Pot14] proposed an elegant solution that combines these two traversals while keeping the modularity of constraint solving. In short, a single traversal builds at the same time the constraints and a continuation that will produce the reconstructed term from a solution of the constraints. This style greatly clarifies the reconstruction implementation. Moreover, since this proposal, the *binding operator* extension of OCaml[3] allows to eliminate most of the administrative code, by taking advantage of the applicative structure of the traversals.

Constraint collecting Allocation constraints are gathered under the form of a tree of constraint scopes that has the same structure as the tree of scopes, i.e., local blocks and iterations. Each constraint scope $\mathcal{S}$ is a quadruple $(\mathcal{I}_l, \mathcal{G}_l, \mathcal{G}_e, \mathcal{C})$. They resemble the description of typing scopes: $\mathcal{G}_e$ are the existentially quantified variables of the scope, $\mathcal{G}_l$ are the local abstract ones, whose outer definition is given by the local instantiation $\mathcal{I}_l$. The constraint $\mathcal{C}$ is a triple $(\mathcal{C}^\eta, \mathcal{C}^\tau, \mathcal{C}^\varepsilon)$ where $\mathcal{C}^\eta$ is a set of size equalities, $\eta_1 = \eta_2$, $\mathcal{C}^\tau$ is a set of sub-typing constraints, $\tau_1 <: \tau_2$ and $\mathcal{C}^\varepsilon$ are location constraints, $\varepsilon_1 = \varepsilon_2$. All these constraints may use free variables, gathered in $\mathcal{G}_f$, i.e., variables that are not part of any existentially quantified or local abstract variables sets.

A solution is a substitution $\mathcal{I}_l$, i.e., an instantiation, of the size, type and location free variables such that all the constraints are fulfilled. To formalize this, we suppose a *constraint checking* judgment $— \mathcal{G} \vdash \mathcal{C} —$ (not detailed here) that ensures that size, type and location constraints are well-formed in $\mathcal{G}$ and fulfilled, using size equality, sub-typing and location equality respectively.

We define a *constraint tree checking* relation $\mathcal{G} \vdash \mathcal{T} \dashv \{\mathcal{I}'/\mathcal{G}'\}$ whose unique deduction rule makes precise how abstract variables may be used. It reads: *in a size, type and location environment $\mathcal{G}$, the constraint tree $\mathcal{T}$ is fulfilled and defines the concrete sizes, types, and locations of local abstract variables with the substitution $\{\mathcal{I}'/\mathcal{G}'\}$.* A constraint tree $— \mathcal{T} = \mathcal{S}\,; \overrightarrow{\mathcal{T}'} —$ is made of a constraint scope $\mathcal{S}$ with sub-trees $\overrightarrow{\mathcal{T}'}$.

Constraint Tree Checking $\quad \mathcal{G} \vdash \mathcal{T} \dashv \{\mathcal{I}'/\mathcal{G}'\}$

$$\frac{\begin{array}{ccc} \mathcal{G} \uplus \mathcal{G}_l \uplus \mathcal{G}_e \vdash \mathcal{T}_1 \dashv \{\mathcal{I}_1/\mathcal{G}_1\} & \{\mathcal{I}'/\mathcal{G}'\} \uplus \{\mathcal{I}''/\mathcal{G}''\} = \{\mathcal{I}_1/\mathcal{G}_1\} \uplus \ldots \uplus \{\mathcal{I}_n/\mathcal{G}_n\} & \\ \ldots & & \\ \mathcal{G} \uplus \mathcal{G}_l \uplus \mathcal{G}_e \vdash \mathcal{T}_n \dashv \{\mathcal{I}_n/\mathcal{G}_n\} & \mathcal{I}'_l = \mathcal{I}_l\,\{\mathcal{I}', \mathcal{I}''/\mathcal{G}', \mathcal{G}''\} & \mathcal{FV}(\,\mathcal{I}'_l\,) \# \mathcal{G}_l \uplus \mathcal{G}_e \\ \mathcal{G} \uplus \mathcal{G}_l \uplus \mathcal{G}_e \vdash \mathcal{C}\,\{\mathcal{I}'/\mathcal{G}'\} & \mathcal{I} = \mathcal{I}'\,\{\mathcal{I}'_l/\mathcal{G}_l\} & \mathcal{FV}(\,\mathcal{I}\,) \# \mathcal{G}_e \end{array}}{\mathcal{G} \vdash (\mathcal{I}_l, \mathcal{G}_l, \mathcal{G}_e, \mathcal{C})\,; \mathcal{T}_1, \ldots, \mathcal{T}_n \dashv \{\mathcal{I}/\mathcal{G}'\} \uplus \{\mathcal{I}'_l\,\{\mathcal{I}/\mathcal{G}'\}/\mathcal{G}_l\}}$$

This rule maintains the following invariants. If $\mathcal{G} \vdash \mathcal{T} \dashv \{\mathcal{I}'/\mathcal{G}'\}$, then (i) $— \mathcal{G} \# \mathcal{G}' —$ the local abstract variables are not part of the environment, and (ii) $— \mathcal{FV}(\,\mathcal{I}'\,) \subset \mathcal{G} —$ their definition is well-formed in the environment. These invariants are preserved thanks to the two disjunction conditions: (i) $— \mathcal{FV}(\,\mathcal{I}_l\,\{\mathcal{I}'/\mathcal{G}'\}\,) \# \mathcal{G}_l \uplus \mathcal{G}_e —$ the definition of local abstract size, type and location may not capture the abstract (local or existentially quantified) variables, (ii) $— \mathcal{FV}(\,\mathcal{I}\,) \# \mathcal{G}_e —$ the part of abstract variables that are forwarded to the outer scope must not capture the existentially quantified variables of the scope.

In short, local abstract variables may be used outside of their scope, and are then substituted with their concrete values, while existentially quantified variables must appear only in their scope. Hence, a solution for a constraint tree may (and sometimes must) use local abstract sizes for free variables that appear outside of their scope. To illustrate this situation, we consider the constraint tree:

$$(\emptyset, \emptyset, \emptyset, 42 = \nu); (\iota, 42, \emptyset, \iota = \nu);$$

It represents two nested scopes ($\emptyset$ denotes empty generalizations and instantiations), where ν is a free size variable and ι is a local abstract size of the inner scope. A correct substitution must replace the free variable ν by local abstract one ι, even if ν appears in the outer scope. Substituting ν with 42 would not solve the constraint $\iota = \nu$.

[3] https://v2.ocaml.org/manual/bindingops.html

Disentangling types and locations We presented a single memory location system and we claimed that well allocated programs are well typed. It is thus sufficient to solve location constraints to elaborate correct terms. However, we adopt two-phase constraint solving by inferring first sizes and types before finding a correct allocation. There are several reasons for doing so.

- Values at equal locations must have equal types. Solving type constraints and more particularly size ones is thus a prerequisite to allocation inference.
- More importantly, we would like MADL to have a *functional* semantics independent of allocations. Its existence should be guaranteed by a location independent type checking. Moreover, sizes must not be determined by location constraints, since sizes affect the semantics.
- Diagnostics for allocation errors are less readable than type error reports. All the program defects that can be detected as type errors thus benefit from simpler explanations.

Scope of this presentation We did not formalize our inference algorithm and several points are still not working in our prototype. Hence we only focus on the resolution of a set of location constraints.

6.2.1 Allocation Inference by Examples

Allocation constraints are equalities between memory locations: $\varepsilon_1 = \varepsilon_2$. Using the normal form of locations, they reduce to equations of the form $\delta_1.\varphi_1 = \delta_2.\varphi_2$. A solution is a location variable δ and a pair of projection functors ψ_1 and ψ_2 such that:

$$\delta.\psi_1.\varphi_1 = \delta.\psi_2.\varphi_2$$

Even with our restricted language of projection functors, there is little chance of solving such equations. To explain our approach, let us first examine how possible solutions are constrained by the expressiveness limits of projection functors.

Matrix reshaping As an example, we implement *matrix reshaping*. This operation transforms a matrix of size (n, p) into a matrix of size (n', p') where $n * p = n' * p'$, such that the row major order of elements is preserved:

$$\texttt{reshape}_{3,4} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

At first glance, this operation is parametrized by four sizes, with a constraint: products of matrix dimensions are equal. In MADL, this constraint is encoded by factorizing the dimensions into a product of sizes:

```
let reshape «a, b, c, d» (i of [a * b] [c * d] _)
returns (o of [a * d] [c * b] _):
  o = split «_» (flatten «_» (i));
```

We recall the location schemes of `split` and `flatten` operators, which are obviously mutual inverses when instantiated with equal sizes:

$$\texttt{split} : \forall\, \nu\, \mu.\ \forall\, \alpha.\ \forall\, \delta : [\nu * \mu]\alpha.\ \delta \longrightarrow \delta.\texttt{split}\ \ \nu\ \ \mu\ \ \alpha$$
$$\texttt{flatten} : \forall\, \nu\, \mu.\ \forall\, \alpha.\ \forall\, \delta : [\nu * \mu]\alpha.\ \delta.\texttt{split}\ \ \nu\ \ \mu\ \ \alpha \longrightarrow \delta$$

In our `reshape` example, the location schemes of `split` and `flatten` are instantiated with fresh variables δ_s and δ_f, respectively. Moreover, we denote by δ_i and δ_o the locations of the argument and result. The constraint collection builds the following system of constraints:

$$\begin{cases} \delta_i = \delta_s.\texttt{split}\ \ \nu_1 * \mu_1\ \ \nu_2 * \mu_2\ \ \alpha \\ \delta_s = \delta_f \\ \delta_o = \delta_f.\texttt{split}\ \ \nu_1 * \mu_2\ \ \nu_2 * \mu_1\ \ \alpha \end{cases}$$

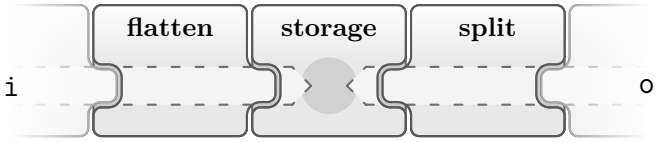
Because the products of `split` sizes are identical, the types of δ_s and δ_f match, this system has a straightforward solution. Given a location variable $\delta : [\nu_1 * \mu_1 * \nu_2 * \mu_2]\alpha$, the above variables are defined by:

$$\begin{cases} \delta_i := \delta.\texttt{split}\ \ \nu_1 * \mu_1\ \ \nu_2 * \mu_2\ \ \alpha \\ \delta_s := \delta \\ \delta_f := \delta \\ \delta_o := \delta.\texttt{split}\ \ \nu_1 * \mu_2\ \ \nu_2 * \mu_1\ \ \alpha \end{cases}$$

Hence the `reshape` operator has the following location scheme, where both the argument and the result are views on a single location:

$$\texttt{reshape} : \forall\, \nu_1\, \mu_2\, \nu_2\, \mu_2.\ \forall\, \alpha.\ \forall\, \delta : [\nu_1 * \mu_1 * \nu_2 * \mu_2]\,\alpha.\ \text{«}\nu_1, \mu_1, \nu_2, \mu_2\text{»}\ \ \delta.\texttt{split}\ \ \nu_1 * \mu_1\ \ \nu_2 * \mu_2\ \ \alpha \longrightarrow \delta.\texttt{split}\ \ \nu_1 * \mu_2\ \ \nu_2 * \mu_1\ \ \alpha$$

In this location scheme, the location variable δ only appears in views: the value that is associated is a local result of the operator. This intuition is made clearer by our puzzle depiction (the arguments and results are represented by faded-out pieces): the memory piece lies in the middle of accesses.



Shapes of puzzle pieces This puzzle highlights that the `flatten` and `split` operations cannot be welded together, a memory storage must be inserted in between. But why do these operators have these shapes? Given an expression, the shape of the associated piece is defined by:

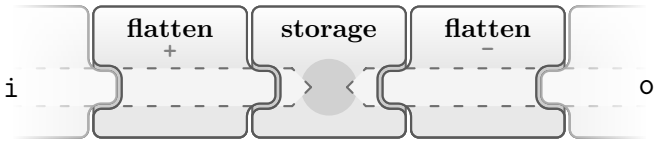
1. *The location signature.* The flows whose locations are variables, i.e., δ, are represented with plugs, while the ones with view locations, i.e., $\delta.\varphi$, are depicted with slots.

2. *The expression use.* The positions of plugs and slots are defined by data-flow direction rather than interface position, (in arguments or results). For operations used in computations (right-hand-side expressions), where data flows from arguments to results, arguments are on the left and results are on the right. On the contrary for operations that appear in patterns, results are on the left and arguments on the right. In short, data always flows from left to right in our puzzles.

To illustrate the second point, consider an alternative definition of the reshape operator, using only `flatten`:

```
let reshape «a, b, c, d» (i of [a * b] [c * d] _)
returns (o of [a * d] [c * b] _):
  flatten «_» (o) = flatten «_» (i)
```

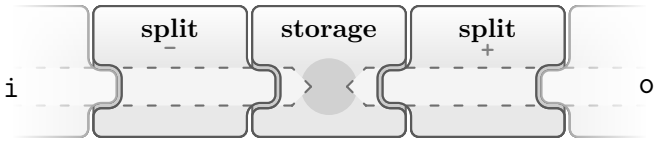
The underlying memory allocation is reassuringly identical. To distinguish the two instances of `flatten`, we add a sign `-` (resp. `+`) to denote the one that appears in a pattern (resp. computation):



In this second implementation, the inserted memory corresponds to the value that passes through the `=` sign. As a last example of multiple terms leading to identical allocations, a `split`-based implementation follows.

```
let reshape «a, b, c, d» (split (u) of [a * b] [c * d] _)
returns (split (u) of [a * d] [c * b] _)
```

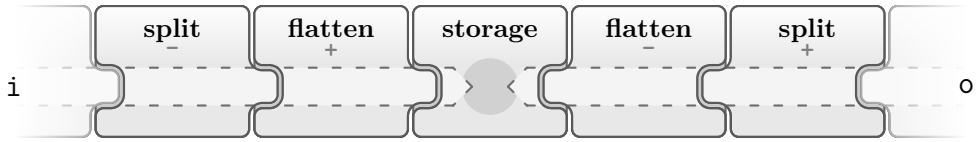
The underlying scheme is unsurprisingly identical (we keep the above names for argument and result although they are not explicit in the term). Notice here that the first instance of `split` is in a pattern. In this case, the memory is held by the variable u.



To conclude this tour, the following operator implements a three-dimensional version of reshaping, by combining the above implementations.

```
let reshape_3D «a, b, c, d, e, f, g, h, i» (split (a) of [a * b * c] [d * e * f] [g * h * i] _)
returns (split (b) of [a * e * i] [d * h * c] [g * b * f] _):
  flatten «_» (b) = flatten «_» (a)
```

We omit its location scheme that would be barely readable, but its puzzle depiction highlights how structural operations may fuse without intermediate memory:



6.2.2 Pull and Push Accesses

Allocation inference identifies some places of the data-flow graph where memory must be introduced between operations that cannot be welded together. This is in strong connection with fusion optimizations that are found in the compilation of functional languages for GPUs such as Futhark [Hen+17]. However, operator fusion is directed in MADL: only equally oriented operations can be fused.

Demand-driven or data-driven? A similiar oriented fusion exists for the Obsidian DSEL: Claessen, Sheeran, and Svensson [CSS12] proposed two complementary representations of arrays so as to generate more regular GPU kernels with operator fusion. They observed that certain operations are more efficiently described in a demand-driven way, using *pull arrays*, while others require a data-driven representation, provided by *push arrays*. The difference between these kinds of operations lies in the regularity of array accesses. For instance, in a demand-driven style, accessing the result of a concatenation requires to select the correct part, i.e., a conditional, whereas it only amounts to an index shift in a data-driven style. We refer the reader to Section 1.3.2 for more details on pull and push arrays.

Our situation resembles that of Obsidian although some differences are worth mentioning:

- We are concerned with fusing *structural* operators at the level of memory locations, whereas Obsidian fuses general computations at the level of GPU kernels, i.e., source code. In MADL fusion is represented by projection functor composition.
- In patterns, data flows in the opposite direction, from results to arguments. Hence, the data-driven or demand-driven nature depend on the operation and its position. A data-driven operation in computations is thus demand-driven when used in patterns, and conversely.
- The choice between one representation or the other is not a matter of efficiency, but rather imposed by the language of projection functors. Moreover, it is managed by the compiler, whereas pull and push arrays are specified by the programmer in Obsidian.

In Obsidian, going from a pull array to a push array, the `push` operation, is an easy task: it consists in reading elements through the demand-driven array (pull) and passing them to the push array. This requires no extra memory or synchronization. In MADL, it is the equivalent of a copy operation. The way back is costly: Obsidian's `sync` operation writes the entire push array in memory, synchronizing threads, before reading from it, as a pull array. This is the exact counterpart of our implicit storage pieces.

In MADL, we fuse memory locations and hence projection functors. As summarized in Figure 6.7, *pull accesses* appear in reading position, i.e., between a storage and a computation. They are represented with a plug at left and a slot at right while *push accesses* appear in writing position i.e., between a computation and a storage. they are represented with a slot at right and a plug at left. From the point of view of the context in which operations are used:

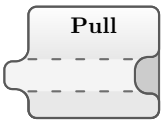
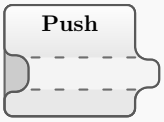
	Style	Position	Signature	Puzzle
Pull access	Demand-driven	Reading	$\delta \longrightarrow \delta.\varphi$	
Push access	Data-driven	Writing	$\delta.\varphi \longrightarrow \delta$	

Figure 6.7: *Pull and push accesses*

- Slots represent flows whose locations are views ($\delta.\varphi$). These flows are stored by the operation. The context is thus free to access them fully or partially, either for reading or writing.
- Plugs depict flows whose locations are variables (δ). These flows are accessed by the operation. The context is thus in charge of storing them and giving full access.

Bidirectional operators Some projection functors are invertible, e.g., `transpose` and `reverse`. Operations whose locations are defined with these invertible functors may be used for both push and pull accesses. In our favorite puzzle depiction, they are represented as a piece with a floating central part. For instance, the `transpose` operator may be given the following location schemes:

$$\forall\, \nu\mu.\ \forall\, \alpha.\ \forall\, \delta : [\nu]\,[\mu]\,\alpha.\ \delta \longrightarrow \delta.\texttt{transpose}\ \ \nu\ \ \mu\ \ \alpha$$
$$\forall\, \nu\mu.\ \forall\, \alpha.\ \forall\, \delta : [\mu]\,[\mu]\,n.\ \delta.\texttt{transpose}\ \ \mu\ \ \nu\ \ \alpha \longrightarrow \delta$$

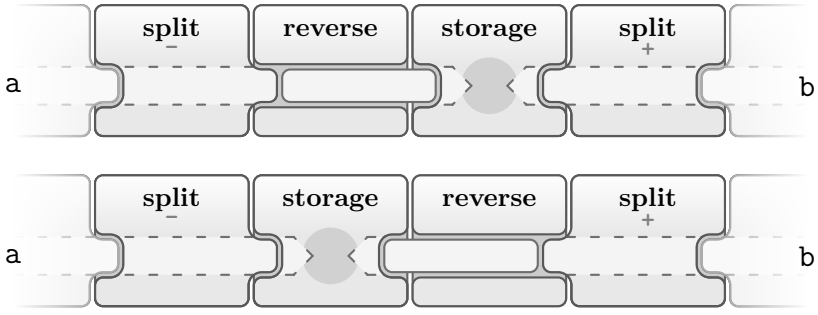
This kind of operator introduces some flexibility in the allocation. As an example, we implement a matrix rotation operator. It constructs a 180° rotated matrix, that has thus the same shape.

$$\texttt{rot}_{3,4} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix} = \begin{pmatrix} 12 & 11 & 10 & 9 \\ 8 & 7 & 6 & 5 \\ 4 & 3 & 2 & 1 \end{pmatrix}$$

This operation can be implemented by inserting an array reversal in the above `reshape` operator. Because sizes are simpler here, size parameters can be deduced from the argument types.

```
let rot (split (u) of ['n]['p]_)
returns (split (v) of ['n]['p]_):
  v = reverse (u)
```

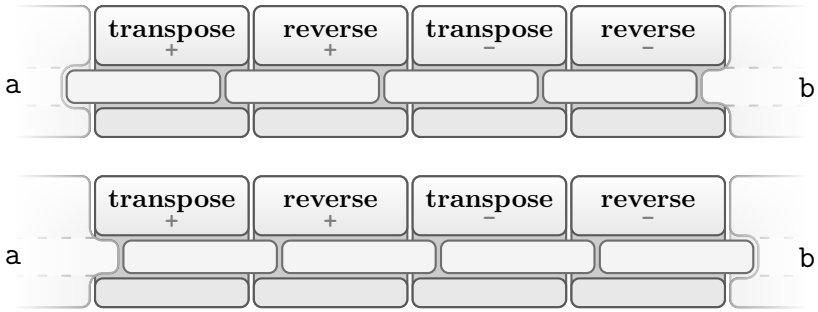
As depicted below, the necessary storage can be introduced at two places:



The above implementation is contrived and matrix rotation is better defined in terms of `reverse` and `transpose` operations:

```
let rotate_mat (a) returns (b):
  transpose (reverse (b)) = reverse (transpose (a))
```

This second implementation has an enjoyable property: all the underlying projection functors are invertible. Hence, the overall operator is invertible as well and it may be instantiated in both reading and writing contexts:



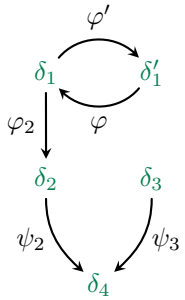
6.2.3 The Constraint Graph

So as to take into account the direction of location constraints, we express them as a directed graph. To do this, general constraints of the form $\delta_1.\varphi_1 = \delta_2.\varphi_2$ are normalized by introducing a new location variable δ and adding the constraints $\delta = \delta_1.\varphi_1$ and $\delta = \delta_2.\varphi_2$.

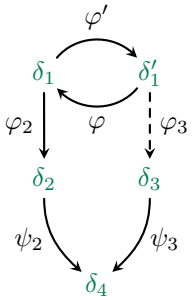
Structure of constraints Given an operator, the set of normalized location constraints has a natural structure of category. Objects are memory locations, represented by location variables, and morphisms are the projection functors that relate two locations. A constraint $\delta_1 = \delta_2.\varphi$ induces a morphism φ from δ_2 to δ_1. Morphisms compose with projection functor composition. Figure 6.8a gives an example of allocation constraints, whose associated category is depicted in Figure 6.8b.

$$\begin{cases} \delta_1 = \delta'_1.\varphi \\ \delta'_1 = \delta_1.\varphi' \\ \delta_2 = \delta_1.\varphi_2 \\ \delta_4 = \delta_2.\psi_2 \\ \delta_4 = \delta_3.\psi_3 \end{cases}$$

(a) Allocation constraints



(b) Allocation relations

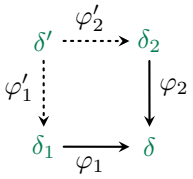


(c) A possible completion

Figure 6.8: *Structure of location constraints*

The targets of morphisms are locations defined as a view of their source through the associated projection functor. Valid memory allocations require consistent definitions of all the locations, this is ensured by the following additional properties on the associated category.

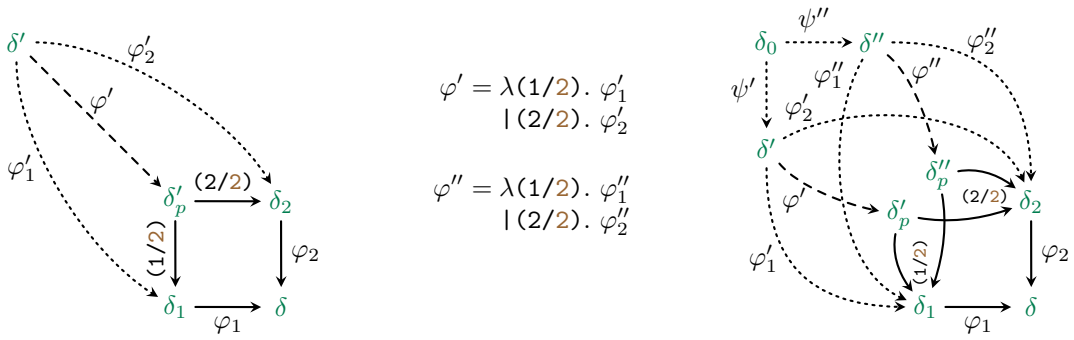
- It is *thin*, i.e., two morphisms with identical sources and targets are equal. Equivalently, all diagrams commute. In particular, all endomorphisms are identities, thus φ and φ' are mutual inverses in the example of Figure 6.8.
- If two morphisms $\varphi_1 : \delta_1 \to \delta$ and $\varphi_2 : \delta_2 \to \delta$ have the same target, there exists an object δ' and two morphisms $\varphi_1 : \delta' \to \delta_1$ and $\varphi_2 : \delta' \to \delta_2$ with the same source whose targets are the sources of φ and φ' respectively such that the diagram at right commutes (which must hold anyway because of the thinness property).



The first property induces a check: projection functors are derived from programs, they must match if they relate the same locations. The second property delineates inference work. It must find locations and projection functors to ensure that views are deduced from a single location. The categories that fulfill these requirements specify memory *up to an isomorphism*. One location must be selected for each class of isomorphic maximal objects, e.g., δ_1 or δ'_1 in the example of Figure 6.8c. An object δ is maximal if all its incoming morphisms are isomorphisms.

Pullbacks The diagram of the second property is the same as the one of pullbacks, but we do not assume universality. Pullbacks exists if we consider all possible locations. It suffices to build a view that is a pair whose components are the two views. However, these locations may not explicitly exist in programs, i.e., no values have such a location. This is why we did not state this property as a pullback.

To prove our claim, given a pair of objects δ_1, δ_2 and morphisms φ_1, φ_1 with a common target δ, we build the pullback using the second property, that gives an object δ' and the dotted morphisms φ'_1 and φ'_2 as depicted in Figure 6.9a. We construct the pullback δ'_p by viewing δ' though the projection functor φ' defined in Figure 6.9b. To prove universality, we must show that the constructed object is independent of the selected common source location δ'. Figure 6.9c presents our naming conventions.



(a) Existence *(b) Definitions* *(c) Uniqueness*

Figure 6.9: *Construction of the pullback*

To show that $\delta'_p = \delta''_p$, it suffices to establish that $\psi' \gg \varphi' = \psi'' \gg \varphi''$, since the definition for these locations would be identical. We have the following equalities

$$\psi' \gg \begin{matrix}\lambda(1/2).\ \varphi'_1 \\ |\,(2/2).\ \varphi'_2\end{matrix} = \begin{matrix}\lambda(1/2).\ \psi' \gg \varphi'_1 \\ |\,(2/2).\ \psi' \gg \varphi'_2\end{matrix} = \begin{matrix}\lambda(1/2).\ \psi'' \gg \varphi''_1 \\ |\,(2/2).\ \psi'' \gg \varphi''_2\end{matrix} = \psi'' \gg \begin{matrix}\lambda(1/2).\ \varphi''_1 \\ |\,(2/2).\ \varphi''_2\end{matrix}$$

The side equalities come from composition definition and the central one is a consequence of the thinness property. This proves the uniqueness of our definition, and hence that a pullback exists.

6.2.4 Implementation

Instead of looking for both locations and views, allocation inference only searches for missing views between existing locations, as represented by the φ_3 dashed morphism in Figure 6.8c. The intuition is that the memory location from which multiple views are built must be explicit in the program.

Constraint simplification Our resolution strategy has not been formalized yet, we present it succinctly. It is built on the following observations.

- The constraints of the form $\delta_1 = \delta_2.\varphi$ give an immediate substitution: $\delta_1 := \delta_2.\varphi$. If φ is invertible, the substitution $\delta_2 := \delta_1.\varphi^{-1}$ is also possible and inference has to select a direction.
- As a generalization, constraints of the form $\delta_1.\psi = \delta_2.\varphi.\psi$ may be treated in the same way, under the condition that ψ is surjective.

The surjectivity condition ensures that inference will not introduce unwanted sharing. Such extra location constraints could prevent the existence of a schedule (see Section 7.1). Otherwise, a trivial allocation is possible by mapping all the scalar locations of a given type to a single location variable with that type.

Our algorithm builds the oriented graph of constraints with distinguished bidirectional edges for the invertible projection functors. Then, this graph is reduced until reaching a tree, or more precisely a forest, with the following operations:

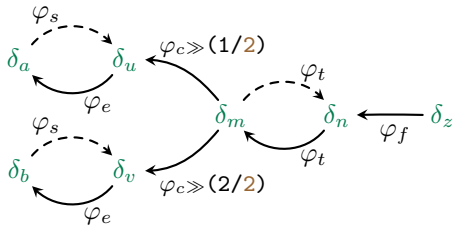
1. Orientate as many invertible projection functors as possible so that nodes have at most one incoming edge.

2. For the nodes with multiple incoming edges with no common ancestors, look for a substitution with the generalized rule above.

3. Orientate one of the remaining invertible projection functors in an arbitrary direction.

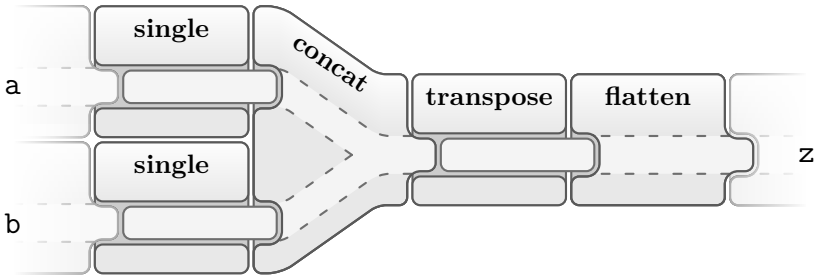
Examples To illustrate the choice of orientation, we implement a zip operator that interleaves two arrays of the same size. As discussed in Section 3.1.2, this operation can be decomposed into simple operators, as recalled below. Here, we name the intermediate result so that the location of a variable x is δ_x. The graph of location constraints is depicted alongside the operator. The projection functors are defined in Section 3.3.2.

```
let patt zip (a, b)
returns (z):
  u = [| a |];
  v = [| b |];
  m = concat (u, v);
  n = transpose (m);
  z = flatten «_» (n);
```



φ_s = `single` $[\nu]\alpha$
φ_e = `extract` $[\nu]\alpha$
φ_c = `cut` 1 2 $[\nu]\alpha$
φ_t = `transpose` 2 ν α
φ_f = `split` ν 2 α

Used in computations, the `flatten` and `concat` operators are both push accesses, whereas the other part of this definition introduces invertible projection functors. In this case, the orientation phase eliminates the dashed edges (morphisms) so that all the node (objects) have only one incoming edge. The puzzle depiction highlights a similar structure: the `zip` operator is a push access:

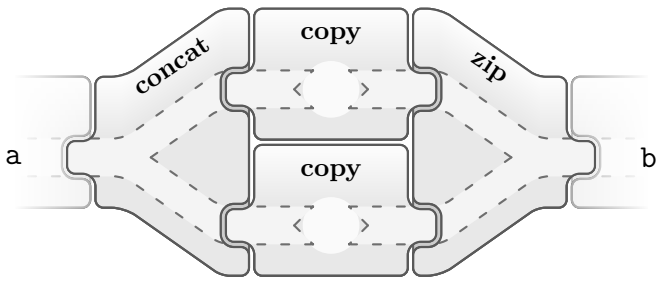


The `zip` operator has the following type scheme, where the projection functors that result from composition are given explicitly:

$$\texttt{zip} : \forall\, \nu.\ \forall\, \alpha.\ \forall\, \delta : [2 * \nu]\alpha.\ \begin{cases} \delta.\big(\lambda[i{<}\nu].\ @[2*i+0{<}2*\nu].\ \mathbf{1}_\alpha\big) \\ \delta.\big(\lambda[i{<}\nu].\ @[2*i+1{<}2*\nu].\ \mathbf{1}_\alpha\big) \end{cases} \longrightarrow \delta.$$

We use the `zip` function to implement the perfect `shuffling` example introduced in Section 5.1.1 in a slightly different way: the two half decks of cards are zipped with two copies that operate directly on arrays. Such copies cannot be implemented with a unique operation (e.g., `memcpy`), we detail this in Section 7.2. As the location signature shows, input and output arrays are located in different memory locations.

```
let shuffle (a)
returns (b):
  concat (l, r) = a;
  e, o = !l, !r;
  b = zip (e, o);
```



$$\texttt{shuffle} : \forall\, \nu.\ \forall\, \alpha.\ \forall\, \delta_1 : [2 * \nu]\alpha,\ \delta_2 : [2 * \nu]\alpha.\ \delta_1 \longrightarrow \delta_2$$

The explicit location requirement So as to illustrate the limits of location inference, we define an `inc_left` function. It increments the leftmost element of an array of integers. We recall that the `cons` pattern builds an array by gluing an element, the head to an array, the tail: `cons (0, [| 1, 2, 3 |]) = [| 0, 1, 2, 3 |]` (see Section 4.1).

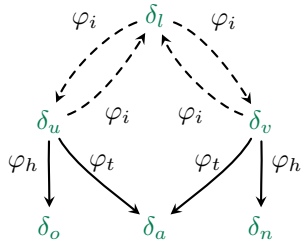
```
1 let inc_left (cons (o,a))
2 returns      (cons (n,a)):
3   n = o + 1
```

```
Error: Invalid locations
```

```
1 let inc_left (cons (o,a) at 'l)
2 returns      (cons (n,a) at 'l):
3   n = o + 1
```

```
val inc_left: size i. [i]int → [i]int
```

Because the same sub-array a is used in the input and output arrays, the result must be stored at the location of the argument. Thus, this operator implements an in-place modification. The version on the left is rejected with an uninformative error,[4] while the second version is accepted. To see why, we depict below the constraint graph that arises from this operator. The δ_u and δ_v are respectively the location of the input and output arrays.



$$\varphi_i = \mathbf{1}_{[\nu+1]\texttt{int}}$$
$$\varphi_h = \texttt{cut}\ 1\ \nu\ \alpha \gg (1/2) \gg \texttt{extract}\ \alpha$$
$$\varphi_t = \texttt{cut}\ 1\ \nu\ \alpha \gg (2/2)$$

The dashed morphisms represent the constraints that arise from the location annotations — · **at** $'l$ —. Without them, i.e., in the rejected version, the inference is unable to find a projection functor (the identity!) to relate δ_u and δ_v. The problem is that both φ_h and φ_t are non-surjective hence the generalized rule for solving multiple definitions of a variable — $\delta_u.\varphi_t = \delta_a = \delta_v.\varphi_t$ — cannot be used. The location annotations solve this issue by introducing a relation between δ_u and δ_v.

The same situation happens in the failing example `parts` we gave in Section 6.1.3. The rejected definition is equivalent to the fixed version given below without line 2. In this case, the location of variable b is constrained in two incompatible ways:

$$\begin{cases} \delta_b = \delta_u.\texttt{cut}\ \nu_1\ \nu_2\ \alpha.(2/2) \\ \delta_b = \delta_v.\texttt{cut}\ \nu_2\ \nu_3\ \alpha.(1/2) \end{cases}$$

```
1 let parts (a,b,c) returns (a ++ b at 'u, b ++ c at 'v):
2   a ++ b ++ c at 'u ++ _ at _ ++ 'v
```

```
val parts: size i j k. type a. [-i+j]a, [i]a, [k-i]a → [j]a, [k]a
```

To circumvent the issue, the body of the operator (line 2) builds an array that contains simultaneously a, b and c. This expression compute a useless value that introduces the missing location δ from which all the others can be deduced. The two location annotations $'u$ `++ _` and `_ ++` $'v$ link this missing location to the locations of the outputs. Once explicit, the location inference deduces:

$$\begin{cases} \delta_b = \delta.\texttt{cut}\ (\nu_1+\nu_2)\ \nu_3\ \alpha.(1/2).\texttt{cut}\ \nu_1\ \nu_2\ \alpha.(2/2) \\ \delta_b = \delta.\texttt{cut}\ \nu_1\ (\nu_2+\nu_3)\ \alpha.(2/2).\texttt{cut}\ \nu_2\ \nu_3\ \alpha.(1/2) \end{cases}$$

The two projection functors that result from the above composition turn out to be equal. This property could not be established without a formal representation of projection functors (and sizes) that allows symbolic computations and comparison.

In all our examples, the solutions to the location failures stem from the same principle, which is a sufficient condition for inference to succeed: *the flows that are part of a single value in memory must be linked to an explicit expression or annotation whose location is that of the overall value.* In most applications, this overall value already exists in the program: it is either decomposed or constructed, hence no annotations are needed.

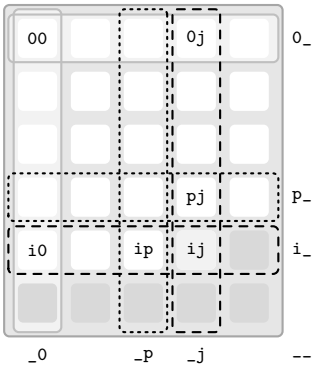
Location equations? *Remark*

The introduction of location constraints is contrived (see the `parts` example). A dedicated syntax would be helpful, maybe as equations on location annotations. Such enhancements have not been investigated for now.

[4] Displaying the full locations is not a solution either. Generating human readable reports is still an open question.

Back to Pascal's triangle In Section 5.2.2, we left the reader with a rejected definition of the computation of Pascal's triangle. We recall this operator below, with different variable names so as to help explanations. The picture at right explains our conventions: variables are named `m**` where `*` is either:

- `_` — a full row of column
- `0` — the first element of the row or column
- `i` or `j` — the current element of the row or column
- `p` — the previously computed element of the row or column

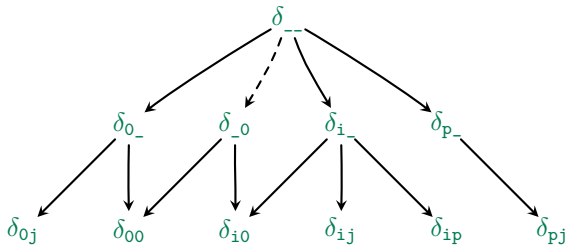


For example, `m_0` is the first column and `mp_` is the previously computed row.

```
1 let pascal () returns (m__):
2   m00 = 1;
3   forward [i] returns (m_0 = cons(m00,[mi0])): mi0 = 1 end
4   forward [j] returns (m0_ = cons(m00,[m0j])): m0j = 1 end
5   forward [i] (cons(_,[mi0]) = m_0) returns (m__ = cons(m0_,[mi_]) as snoc(_,mp_)),
6           [j] (cons(_,[mpj]) = mp_) returns (mi_ = cons(mi0,[mij]) as snoc(_,mip)):
7     mij = mip + mpj
```

```
Error: Invalid locations
```

The error message does not help much. Actually, the issue comes from the first column `m_0`, whose location cannot be derived from the location of the matrix `m__`. It results in two incompatible locations for both `m00` and `mi0`. The situation is more clearly visible with the constraint graph below (we omit the projection functors), where the missing relation is depicted with a dashed arrow. This explains why the conflict is reported on line 4: `m00` is used there.



To guide location inference, it suffices to relate the location $\delta_{\_\_}$ of the matrix to the location $\delta_{\_0}$ of the first column. This is the role of line 4 in the following definition. It uses an undeclared location variable $'c$ that also constrains the location of `m_0` (line 9). The second annotation $'r$ has no other role than to pursue our quest for regularity: the location $\delta_{0\_}$ already stems from $\delta_{\_\_}$. Inference succeeds and produces the expected location scheme $\forall\,\nu\mu.\ \forall\,\delta\colon[\nu]\,[\mu]\,\texttt{int}.\ ()\longrightarrow\delta$.

```
1  let pascal () returns (m):
2
3    // Location inference helpers: tell how to deduce r and c locations from m.
4    transpose (m) at cons ('c, _);
5                m  at cons ('r, _);
6
7    // Computations
8    d = 1;                                                  // Top-left corner: m[0][0]
9    forward returns (c at 'c = cons (d, [c])): c = 1 end  // Left column: m[0:n][0]
10   forward returns (r at 'r = cons (d, [r])): r = 1 end  // Top row    : m[0][0:n]
11   forward (cons (_, [c]) = c) returns (m = cons (r, [m]) as snoc (_, r)),
12           (cons (_, [r]) = r) returns (m = cons (c, [m]) as snoc (_, c)):
13     m = c + r
```

```
val pascal: size i j. [i][j]int
```

6.2.5 Partial Array Location

We rapidly sketched in Section 5.2.1 the typing constraints that apply for referencing the *built* part of arrays: for an expression `|e|`, e must have type $[\eta_1 * \iota + \eta_2]\tau$ where ι, the size variable that represents the index value, appears neither in τ, η_1 or η_2. This prevents, for example, expressions

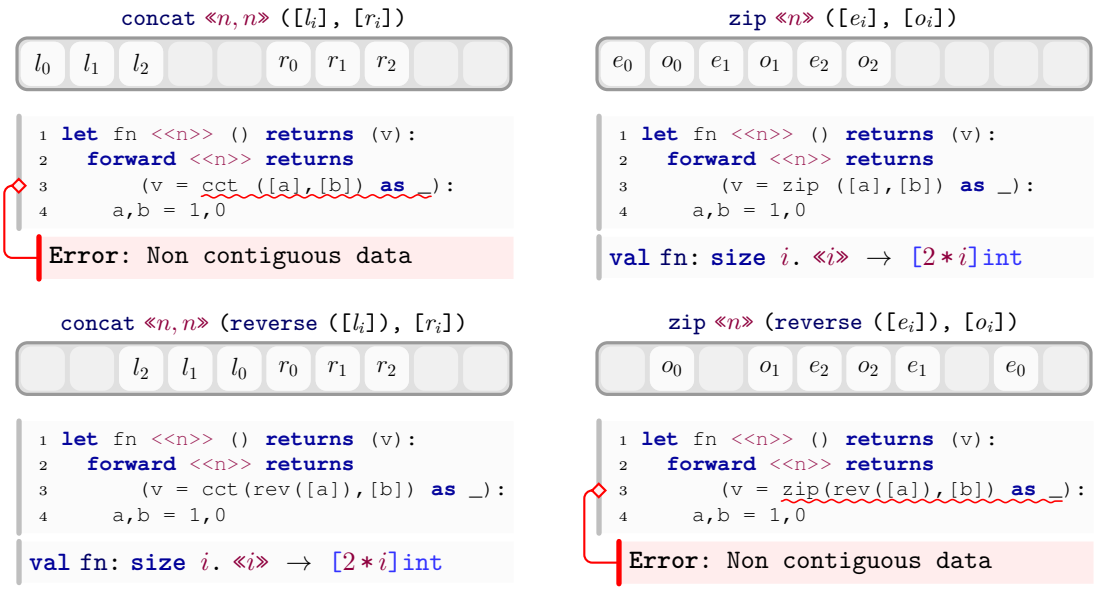


Figure 6.10: *The contiguity requirement (here, `concat` and `reverse` are renamed `cct` and `rev`).*

like `|transpose ([e])|` since `transpose ([e])` would have type $[\nu][\iota]\alpha$, hence the size variable ι would escape its scope.

Locations impose a second kind of constraints. The built parts must be given a location, which allows to treat them as normal arrays in the body of iteration. With complex clauses, such as those found in Pascal's Triangle example — `cons (r, [m])`, this is not always possible.

The contiguity constraint As an example, let us introduce (useless) partial aliases — `· as _` — in the `shuffle` example presented in Section 5.1.1. Following the principles of Section 5.2.1, the compiler introduces built-array constructs around the aliases: `|concat ([left_i], right_i) as _|` and `|zip ([even_i], odd_i) as _|`.

```
let shuffle (a) returns (b):
  forward (concat ([left_i], [right_i]) as _ = a)
  returns (b = zip ([even_i], [odd_i]) as _):
    even_i, odd_i = !left_i, !right_i
```

```
Error: Non contiguous data
```

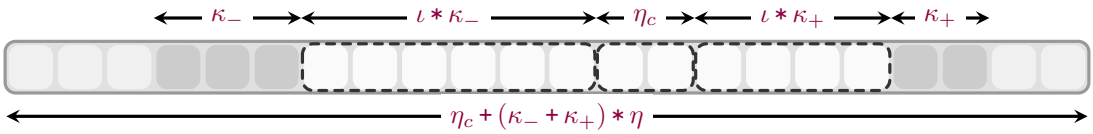
An error occurs for the first partial alias because it cannot give a location for the partial array. Indeed, the already traversed elements are not side by side. There is a hole between those of the left and right halves of a.

The situation is depicted in Figure 6.10 alongside various other schemes of complex aggregation. In each case, the operator builds an array of an even size, by producing two elements at each iteration. Only the top right and bottom left situations are suitable for partial aliases. The two others cannot be addressed with a simple projection functor because the constructed elements at a given iteration are not contiguous.

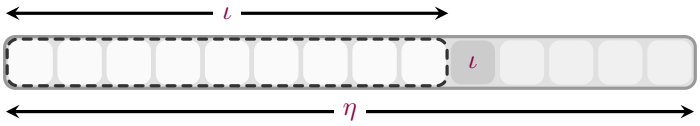
As a more general scheme, the partially aliasable arrays have the following structure. They grow on each side from a constant number of elements, starting from an iteration independent array. Such an aggregation scheme may be obtained with the expression below where c is an array defined outside of the iteration, κ_- and κ_+ are two sizes and l and r are arrays of size κ_- and κ_+, respectively, that are produced at each iteration. We recall that `++` is the infix version of the `concat` operator.

$$\texttt{|reverse (flatten «}k_-\texttt{» ([}l\texttt{])) ++ }c\texttt{ ++ flatten «}k_+\texttt{» ([}r\texttt{])|}$$

The various parts of the above aggregated array are depicted below. We suppose that the iteration size is η, the current index ι, and that η_c is the size of array c. Already traversed elements are brighter while the current elements are darker.



The built array projection functor In the current prototype, partial aliases are supported with compiler constructed projection functors that describe the location of the partial alias and current elements. As an example, consider the simple partial alias: a `= |[`ai`] as` ap`|`, depicted below, where η is the iteration size and ι the current index.



By denoting δ_a, δ_{ai} and δ_{ap} the location of variables a, ai and ap, respectively, the compiler builds the following location constraints:

$$\begin{cases} \delta_{ap} & := \delta_a.\varphi.(1/2) \\ \delta_{ai} & := \delta_a.\varphi.(2/2).(1/1) \end{cases} \quad \text{where} \quad \varphi := \begin{array}{l} \lambda(1/2).\ \lambda[\kappa{<}\iota].\ @[\kappa{<}\eta].\ \mathbf{1}_\alpha \\ \mid(2/2).\ \lambda(1/1).\ @[\iota{<}\eta].\ \mathbf{1}_\alpha \end{array}$$

The dedicated *built array projection functor* φ allows to build a view on an array of type $[\eta]\alpha$. This view is a pair. Its first component is the left part of the array, of size ι: $\lambda[\kappa{<}\iota].\ @[\kappa{<}\eta].\ \mathbf{1}_\alpha$. The second component is a tuple of arity 1, whose single element is the ι^{th} element of the total array, i.e., the one that is defined by the current iteration.

The functor φ defines a view whose second component is a *tuple* of the current elements. This allows to combine multiple aggregated arrays. The following example builds an array by adding one element on the left and two interleaved elements on the right. Its built array projection functor is given below. It provides three current element locations.

$$a \texttt{ = |concat (reverse ([}ai\texttt{]), zip ([}bi\texttt{], [}ci\texttt{])) as } ap\texttt{|}$$

$$\begin{cases} \delta_{ap} & := \delta_a.\varphi.(1/2) \\ \delta_{ai} & := \delta_a.\varphi.(2/2).(1/3) \\ \delta_{bi} & := \delta_a.\varphi.(2/2).(2/3) \\ \delta_{ci} & := \delta_a.\varphi.(2/2).(3/3) \end{cases} \quad \text{where} \quad \varphi := \begin{array}{l} \lambda(1/2).\ \lambda[\kappa{<}3*\iota].\ @[\kappa+\eta-\iota{<}3*\eta].\ \mathbf{1}_\alpha \\ \mid(2/2).\ \lambda(1/3).\ @[\eta-1-\iota{<}3*\eta].\ \mathbf{1}_\alpha \\ \quad \mid(2/3).\ @[\eta+2*\iota{<}3*\eta].\ \mathbf{1}_\alpha \\ \quad \mid(3/3).\ @[\eta+2*\iota+1{<}3*\eta].\ \mathbf{1}_\alpha \end{array}$$

If the outer dimensions of the aggregated array are fused, the location of current elements, that are arrays themselves, cannot be described as a simple projection, e.g., $@[\iota{<}\eta].\ \mathbf{1}_\alpha$ in the a `= |[`ai`] as` ap`|` example. These locations are thus slices, as shown below:

$$a \texttt{ = |cons (}v\texttt{, flatten «5» ([}ai\texttt{])) as } ap\texttt{|}$$

$$\begin{cases} \delta_{ap} & := \delta_a.\varphi.(1/2) \\ \delta_{ai} & := \delta_a.\varphi.(2/2).(1/1) \end{cases} \quad \text{where} \quad \varphi := \begin{array}{l} \lambda(1/2).\ \lambda[\kappa{<}1+5*\iota].\ @[1+\iota{<}5*\eta].\ \mathbf{1}_\alpha \\ \mid(2/2).\ \lambda(1/1).\ \lambda[\kappa{<}5].\ @[1+5*\iota+\kappa{<}5*\eta].\ \mathbf{1}_\alpha \end{array}$$

The built array projection functor is a *global* description. It is constructed at the level of the built construct — $|\cdot|$ — and is passed to the aggregation constructs — $[\cdot]$ — to retrieve the location of current elements. By contrast, the rules we propose for aggregation without partial aliases in Figure 6.6 are *local*. The locations of current elements are directly deduced from the location of the array: $\delta_{ai} = \delta_a.[\iota{<}\nu]$.

This global description allows to encode an extra property. By construction, the built array projection functor is *injective*. Hence, its various components have distinct locations. For scheduling (see Chapter 7), this allows to treat partial aliases like any other constructs: because locations are disjoint, the partial aliases do not depend on the current elements, without needing dedicated treatments. This property would not hold without the intermediate projection functor.

Conclusion

The description of memory locations is based on a simple model. The memory contains a collection of non-aliasing, typed data without indirection. These memory chunks are accessed by building views using projection functors.

Memory allocation as a type system We propose a memory discipline for MADL that specifies how locations are constrained by programs: (i) structural expressions introduce relations between the location of their inputs and outputs and (ii) location annotations insert additional constraints. The underlying description is the three layer type system summarized in the following table.

	Sizes (η)	Types (τ)	Locations (ε)
Variables	ν	α	$\delta : \tau$
Base objects	$0, 1, 2, \ldots$	`int`, `bool`	–
Derived objects	$\eta + \eta$, $\eta * \eta$	$[\eta]\tau$, $\tau * \ldots * \tau$	$\varepsilon.\varphi$
Existential quantification	Explicit	–	Implicit
Unconstrained variables	Error	Warning	Correct

These three layers are handled with the same principle: polymorphism. Hence, operators are described with an allocation scheme — $\forall \vec{\nu}.\ \forall \vec{\alpha}.\ \forall \vec{\delta}.\ \langle\!\langle \vec{\eta} \rangle\!\rangle\ \vec{\varepsilon} \xrightarrow{\vec{\varepsilon}} \vec{\varepsilon}$ — that quantifies universally on size, type and location variables. These abstract variables are given concrete values upon instantiation.

The language of locations has no base objects. Hence each location refers either to a generalized or to an existentially quantified location variable. Intuitively, generalized variables represent memory locations of the context while existentially quantified variables are local storage of the operator.

Existentially quantified locations are implicit and deduced from the unconstrained variables that remain after location inference. By contrast, existentially quantified size variables are introduced by specific places, e.g., the current index of `forward` iterations, and unconstrained size variables are forbidden because the semantics can depend on them. We have not considered existentially quantified types for now. Because the semantics is independent of types (see Section 2.2.2), unconstrained type variables are harmless, but they are a sign of dead-code.

In iteration, the existential size that represents the current index is crucial, even if it is not captured explicitly (see Section 5.1.3). Indeed, this size is used to define the location of the current element, when traversing or aggregating arrays.

Inference The resolution of location constraints is based on composition and comparison of projection functors. The inference orientates the constraints that arise from invertible functors so as to define each location as a unique composition of functors. For the remaining cases, inference checks that two definitions of the same location are indeed equal.

The orientation of location constraints arises from the limits of the projection functor language. In MADL, it appears through expressions that are either data-driven or demand-driven, depending on the location (variable or view) of inputs and outputs. Although inference is incomplete, it will succeed if all the locations of values that are embedded in same memory chunk can be deduced from a unique memory location that exists in the program, i.e., a variable or a location annotation.

Among the identified limits of our proposal, the readability of locations needs further investigations. Depending on the situation, an explicit display of the functor, or a composition based representation may be appropriate, but it may be better to link locations to variables, expressions or location annotations that exist in the program.

Allocation and schedule The location discipline ensures that the structural and computational expressions are consistent and determines the position of memory in the data-flow graph. However this condition is not sufficient to ensure that a program is correct, which also depends on the existence of a schedule, which is the object of Chapter 7.

In MADL, memory allocation precedes the schedule. By contrast, the source-to-source compilation used for SCADE [CPP05] does the opposite: memory is optimized after scheduling. Because allocation is selected first, inference must preserve schedulability chances by introducing no memory sharing.

7 The Back-end

Introduction

To give an insight of the missing parts of the compilation process, we briefly describe the principles of code generation. Type and allocation inference allows to reconstruct location information from MADL programs. Locations define how memory is used. In particular they specify which local variables are needed and how they are accessed. Before generating sequential code, another analysis is needed: schedulability. It extracts a correct order from an allocated MADL program. Following this order, code generation produces instructions from the computational expressions. These instructions access memory through complex projections defined by the locations.

Both scheduling and code generation are on a preliminary stage in terms of formalization and prototyping. This chapter aims at giving an insight of the complete compilation process. It also summarizes our understanding of these problems, to serve as a starting point for future work.

7.1 Scheduling

Scheduling must build a total order of the computations. This order must be compatible with (i) the data-dependencies, i.e., no values are used before being computed, and (ii) the allocation, i.e., no values are used after being overwritten.

To this aim, programs are traversed to build a located data-flow graph, that we present in Section 7.1.2. This graph contains the data-dependencies, i.e., order relations between a write and a read. It is extended by adding anti-dependencies, i.e., order relations between a read and a write, to ensure that no values are overwritten before being used.

This dependency analysis builds on a simple model of operators: they are described as a set of atomic computations that modify some locations. For this abstraction to be applicable, some non-trivial conditions must be fulfilled. We study them in Section 7.1.1.

7.1.1 Operation atomicity

The atomicity hypothesis can be formulated as local properties of expressions: their results should be computable and accessible at the location given by the location inference presented in Chapter 6. The underlying condition is that the values that are written to memory must not conflict with other results or with the values used to compute them. These checks are performed after location inference.

Injective location of writes So that a written value may be accessed afterwards, its location must be *injective*, i.e., all its components must be mapped to distinct elements of the data it is stored in. This verification is performed after location inference. For an expression that modifies a location $\delta.\varphi$, the projection functor φ must be injective, as defined in Section 3.4.2. The two examples below illustrate situations where non-injective writes are detected.

```
1 let compress (a)
2 returns (b at repeat (_)):
3   b = !a;
```

```
Error: Non-injective write
```

```
1 let compress (a) returns (b at repeat (_)):
2   forward ([ai] = a) returns (b = [bi]):
3     bi = !ai
```

```
Error: Non-injective write
```

In both definitions, the location annotations — `b at repeat (_)` — specify that all the elements of the output array b are mapped to a single location. The location of b is $\delta_b := \delta.\texttt{repeat}\ \nu\ \alpha$. In the left-hand-side operator, the input array a cannot be copied to this location, since `repeat` $\nu\ \alpha$ is not an injective functor. This program would be rejected even if the size of the arrays were 1, because injectivity stems from additional flags that decorate projection functors independently of their size parameters.

For the right-hand-side version, the cause of the error is more subtle. The copy (line 3) duplicates the current element of a, into bi, the current element of b. By denoting $\iota < \nu$ the iteration context (see Section 6.1.2), the location δ_{bi} of bi is defined by:

$$\delta_{bi} = \delta_b.[\iota{<}\nu] = \delta.\texttt{repeat}\ \nu\ \alpha.[\iota{<}\nu] = \delta.\mathbf{1}_\alpha$$

Hence, it is injective. However, this is the result of a composition of functors in which one of them is non injective, it is considered non injective too.

The copy constraint Copies allow to duplicate compound data. They are thus non atomic operations. To ensure that the components can be copied in any order, MADL imposes the following restriction: the location of the source and destination of copy operations must be either equal or disjoint.

The following examples illustrate various tries of definition of an in-place array reversal. In each definition, the location annotations — `· as 'l` — requires that the input and output arrays have the same location.

```
1 let rev (a at 'l) returns (b at 'l):
2   b = reverse (a)
```
```
Error: Invalid locations
```

```
1 let rev (a at 'l) returns (b at 'l):
2   b = !reverse (a)
```
```
Error: Overlapping copy
```

```
1 let rev (a at 'l) returns (b at 'l):
2   b = !reverse (!a)
```
```
val rev: size i. type a. [i]a → [i]a
```

```
1 let rev (a at 'l) returns (b at 'l):
2   b = !!reverse (a)
```
```
val rev: size i. type a. [i]a → [i]a
```

The first program is incorrect because locations are inconsistent. The conjunction of location annotations and the reverse operator lead to the constraint $\delta_l = \delta_l.\texttt{reverse}\ \nu\ \alpha$. It could be solved by mapping all the elements to the same location, i.e., $\delta_l := \delta.\texttt{repeat}\ \nu\ \alpha$ but such allocation would not be relevant. This solution is not considered and location inference fails (see Section 6.2).

The second program, on the top-right corner, is well-allocated. However, the copy is invalid because the location of its source — $\delta_l.\texttt{reverse}\ \nu\ \alpha$ — and the location of its destination δ_l are neither disjoint nor equal. Indeed, such a copy cannot be implemented by moving the elements one-by-one from the source to the destination.

The solution lies in inserting a second copy. Its position does not matter. The two-copy pattern `!!·` introduces a location variable that is independent of the other locations of the program. It resembles the `swap` example introduced in Section 4.2. Hence the array may be freely copied to and from this intermediate location.

Operator instantiation To allow a modular verification of the correctness of location updates, we enriched the location signature defined in Section 6.1.2 with the set of modified locations:

$$\theta := \vec{\delta_i} \xrightarrow[\vec{\delta_w}]{\vec{\delta_s}} \vec{\delta_o}$$

In the above signature, $\vec{\delta_i}$ and $\vec{\delta_o}$ are the input and output locations respectively; $\vec{\delta_s}$ are the locations of the state and $\vec{\delta_w}$ are the locations that are written by the operator.

At instantiation, the compiler checks that the location variables that are used in the written location are instantiated with injective locations. Moreover, these locations must not intersect with any other location. This check is similar to the *uniqueness types* proposed by Henriksen et al. [Hen+17] for Futhark. The objectives of uniqueness types are similar: they flag some inputs that are modified in place. In the location discipline presented in Section 6.1.2, this amounts to requiring an additional property, entitled *UniqueWrites*, in the rule Inst that defines location scheme instantiation. It is given below, where W is the set of location variables.

$$\mathit{UniqueWrites}(\mathcal{I}, W) \iff \forall\, \delta : \varepsilon \in \mathcal{I}^{\varepsilon}.\ \delta \in W \implies \begin{cases} \varepsilon \text{ is injective} \\ \forall\, \delta' : \varepsilon' \in \mathcal{I}^{\varepsilon}.\ v' \neq \delta' \implies \varepsilon \# \varepsilon' \end{cases}$$

The rule INST of the location discipline (see Section 6.1.2) is redefined as follows, where $\mathit{Writes}(\theta)$ denotes the set of location variables used in the written locations of θ, i.e., $\overrightarrow{\delta_w}$.

$$\textsc{Inst}\ \frac{\overrightarrow{\eta'} = \overrightarrow{\eta}\,\{\mathcal{I}/\mathcal{G}\} \qquad \theta' = \theta\,\{\mathcal{I}/\mathcal{G}\} \qquad \mathit{UniqueWrites}(\mathcal{I}, \mathit{Writes}(\theta))}{\theta' := \left(\forall\, \mathcal{G}.\ \text{«}\overrightarrow{\eta}\text{»}\ \theta\right)\ [\mathcal{I}]\ \text{«}\overrightarrow{\eta'}\text{»}}$$

This verification is restrictive. The following nodes `add1` and `add2` define a point-to-point vector addition. They have identical type schemes — $\forall\, \nu.\ [\nu]\texttt{int},\ [\nu]\texttt{int} \longrightarrow [\nu]\texttt{int}$ — but their location schemes differ. Indeed, the location annotations in `add2` impose that the first input and the output are stored at the same location.

```
let add1 (u, v) returns (w):
  forward ([ui] = u) ([vi] = v) returns (w = [wi]):
    wi = ui + vi;
```

$$\texttt{add1} : \forall\, \nu.\ \forall\, \delta_1 : [\nu]\texttt{int},\ \delta_2 : [\nu]\texttt{int},\ \delta_3 : [\nu]\texttt{int}.\ \delta_1,\ \delta_2 \underset{\delta_3}{\longrightarrow} \delta_3$$

```
let add2 (u at 'l, v) returns (w at 'l):
  forward ([ui] = u) ([vi] = v) returns (w = [wi]):
    wi = ui + vi;
```

$$\texttt{add2} : \forall\, \nu.\ \forall\, \delta_1 : [\nu]\texttt{int},\ \delta_2 : [\nu]\texttt{int}.\ \delta_1,\ \delta_2 \underset{\delta_1}{\longrightarrow} \delta_1$$

As a consequence of the modular verification of writings, the first version — `add1` — may not be used to implement an in-place addition. The second version — `add2` — must be used instead.

```
let sum (a) returns (b):
  b init [| 0,0 |] = add1 (last b, a)
```

```
Error: Modified location aliased
```

```
let sum (a) returns (b):
  b init [| 0,0 |] = add2 (last b, a)
```

```
val sum: [2]int → [2]int
```

Some flexibility could be gained by considering a more precise description of writes in the location schemes. We have not explored this.

Iterations Alike complex copies, iterations are non atomic operations. For a `forward` construct to be valid, the iterations must not conflict. A write in a given iteration must not prevent the access to a value needed in a subsequent iteration. The following operator increments an array, where the location of the input and the output are reversed views one of the other.

```
let inc_rev (a at 'l) returns (b at reverse ('l)):
  forward ([ai] = a) returns (b = [bi]):
    bi = ai + 1
```

Incorrect program

This definition must be rejected because the production of b conflicts with the accesses to a. For instance, the first iteration would overwrite the last element of a which is needed in the last iteration.

We *envisage* a restrictive, sufficient condition: iterations may traverse and produce arrays only if they have either identical or disjoint locations. This check resembles the overlapping copy detection. It has not yet been implemented.

7.1.2 Memory-Aware Scheduling

The properties and verifications sketched in Section 7.1.1 allow to abstract operations. Each expression, including local blocks and iterations, is modeled as an atomic operation that modifies a set of locations. These operations are connected by data-dependencies.

A valid schedule of a MADL program is a total order of the operations that is consistent with both the data dependencies and the locations. The first requirement ensures that values

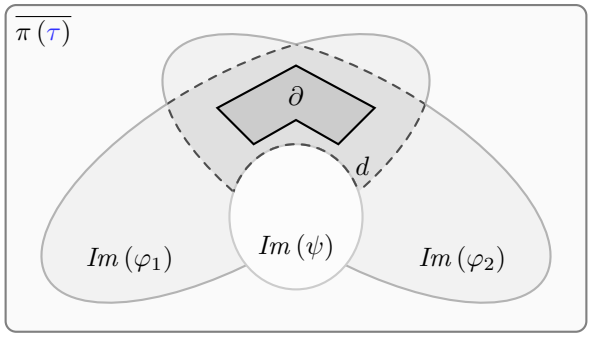


Figure 7.1: *Approximation of dependency ∂ with $d = (\{\varphi_1, \varphi_2\}, \{\psi\})$*

are computed before use. The second requirement guarantees that computed values are not overwritten if they are still needed. In other words, two values that are needed at the same time must have different locations. This property is the basis for register allocation, although it is used for a different purpose. Register allocation determines memory sharing given a fixed order of computation while scheduling retrieves a correct order from sharing constraints.

The scheduling analysis we present here has been implemented for a former internal representation of MADL. However, it has not yet been ported to the current abstract syntax tree.

Representation of dependencies Dependencies are described by locations, hence by projection functors. Properties of projection functors such as inclusion and disjunction, are coarsely approximated. For that reason, we propose a representation of dependencies as a combination of projection functors with better chances of establishing these properties

To simplify the presentation, we will only consider a single location variable δ, of type τ. In practice, dependencies are built for all the location variables independently. We denote $\mathcal{F}(\tau)$ the set of projection functors with type $\tau \to \tau'$ (see Chapter 3). These functors allow to build views over δ.

Definition 7.1 (Dependency)**.** A *dependency* $d \in \mathcal{D}$ is a pair (d^+, d^-) where $d^+ \in \mathcal{P}(\mathcal{F}(\tau)) \setminus \emptyset$ and $d^- \in \mathcal{P}(\mathcal{F}(\tau))$ are sets of projection functors.

The *full dependency* — $\mathbf{1} := (\{\mathbf{1}_\tau\}, \emptyset)$ — describes a dependency that uses all the parts of the location variable δ.

The *null dependency* — $\boldsymbol{O} := (\{\mathbf{1}_\tau\}, \{\mathbf{1}_\tau\})$ — represents an empty dependency.

Intuitively, a dependency (d^+, d^-) represents a subset D of the atomic components of δ, represented by their maximal projections, see Section 3.2.2. It is defined by:

$$D = \left(\bigcap_{\varphi \in d^+} Im(\varphi) \right) \setminus \left(\bigcup_{\psi \in D^-} Im(\psi) \right)$$

The true dependency — ∂ — approximated by d is contained in the intersection of the image of its *covering functors* d^+ minus the union of the image of the *excluded functors* d^-. This is depicted in Figure 7.1.

As an example, consider an array update operation — y `of` $[n]t$ `=` $(x$ `with` $[i]$ `=` $e)$ — (not prototyped yet) that writes the value of e in the i^{th} element of array x of type $[n]t$. The data-dependency between x and y is described by the dependency $(\{\ \mathbf{1}_{[n]t}\ \},\ \{\ @[i<n].\ \mathbf{1}_t\ \})$ that reads: all the elements of the location of x except the one at index $[i<n]$.

A direct description of the set of maximal projections is impossible because we consider projection functions that are generic in sizes. This representation aims at using the approximations for projection functor inclusion — $\varphi \subset \varphi'$ — and disjunction — $\varphi \# \varphi$ — defined in Section 3.4.2.

Definition 7.2 (Emptiness)**.** A dependency $d = (d^+, d^-) \in \mathcal{D}$ is *empty*, denoted $Empty(d)$, if the following disjunction holds:

$$\exists \varphi \in d^+.\ \exists \psi \in d^-.\ \varphi \subset \psi \quad \vee \quad \exists \varphi, \varphi' \in d^+.\ \varphi \# \varphi'$$

By definition, we have $Empty(\mathbf{0})$ and $\neg Empty(\mathbf{1})$.

Emptiness is approximated by looking for a covering functor that is contained in an excluding functor or two disjoint covering functors.

Definition 7.3 (Composition)**.** Given two dependencies $d_1 = (d_1^+, d_1^-)$ and $d_2 = (d_2^+, d_2^-)$, their *composition* — $d_2 \circ d_1$ — is defined as

$$d_2 \circ d_1 = (d_1^+ \cup d_2^+, d_1^- \cup d_2^-)$$

Definition 7.4 (Conflict)**.** A projection functor $\varphi \in \mathcal{F}(\tau)$ *conflicts* with a dependency $d = (d^+, d^-) \in \mathcal{D}$, denoted $\varphi \not\parallel d$, if:

$$\neg \mathit{Empty}((d^+ \cup \{\varphi\}, d^-))$$

Unfolding the definition of emptiness, a conflict between a projection function $\varphi \in \mathcal{F}(\tau)$ and a dependency $d = (d^+, d^-) \in \mathcal{D}$ is assumed if none of the covering functor $\varphi' \in d^+$ is disjoint of φ and none of the excluded function $\psi \in d^-$ contains φ.

A located data-flow graph The usual data-flow graph is supplemented with location information. Each node is associated to a set of written locations and each dependency is associated to the location of the flows it stems from.

Definition 7.5 (Located data-flow graph)**.** Given a set of identifiers $\mathcal{N}$, a *located data-flow graph* is a pair (N, E) where

- $N : \mathcal{N} \to \mathcal{P}(\mathcal{F}(\tau))$ associates to each *nodes* — n — a set of locations
- $E : \mathcal{N} \times \mathcal{N} \to \mathcal{P}(\mathcal{D})$ associates to each *edges* — $s \to t$ — a set of dependencies. We call s the source and t the target.

Definition 7.6 (Path dependency)**.** Given a *path* $n_0 \to n_1 \to \ldots \to n_k$, its dependency set, also denoted $E(n_0 \to n_1 \to \ldots \to n_k)$, is defined by:

$$E(n_0 \to n_1 \to \ldots \to n_k) = \left\{ d_k \circ \ldots \circ d_1 \circ \left(\mathbf{1}_\tau, \cup_{i=1}^{k-1} N(n_i)\right) \mid d_i \in E(n_{i-1} \to n_i) \right\}$$

The dependencies along a path are made of the composition of the dependencies of each edge minus the written locations of the inner nodes of the path.

Definition 7.7 (Transitive closure)**.** Given a located data-flow graph (N, E), its *transitive closure* is the graph (N, E') such that

$$E(s \to t) = \bigcup_{n_1, \ldots, n_k \in \mathcal{N}} E(s \to n_1 \to \ldots \to n_k \to t)$$

The nodes which have an empty written location set serve only as wiring. For the scheduling, they are useless. Thus, dependency analysis is conducted on *reduced* located data-flow graphs. Given a graph (N, E), its reduced graph is obtained by restricting the transitive closure (N, E') to the subset of identifiers $\mathcal{N}_c \subset \mathcal{N}$ that represent computations, i.e., such that $\forall n \in \mathcal{N}_c.\ N(n) \neq \emptyset$. Figure 7.2 depicts such a reduction on the circular convolution introduced in Section 1.5.2.

Turning interferences into anti-dependencies In the following, we suppose given a reduced located data-flow graph (N, E). Before presenting our scheduling algorithm, we define a notion of interference between a computation (a node) and a dependency (an edge). Intuitively, a computation interferes with a dependency if the computation cannot be scheduled between the source and target of the dependency.

Definition 7.8 (Interference)**.** A node n and an edge $s \to t$ *interfere* – $n \not\parallel s \to t$ – if and only if:

$$\exists \varphi \in N(n).\ \exists d \in E(s \to t).\ \varphi \not\parallel d$$

Interferences are defined from the locations. A node n interferes with an edge $s \to t$ if one of the locations written by n conflicts with one of the dependencies between s and t.

Our scheduling algorithm proceeds as follow:

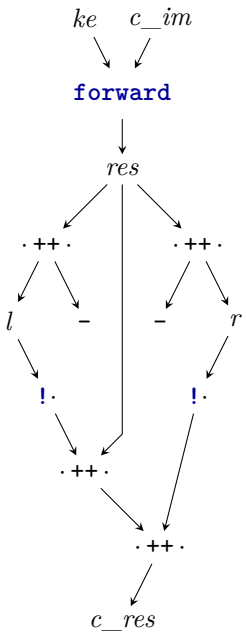
1. *Extract interferences and data-dependencies.* Compute the set of interferences $n \not\parallel s \to t$, as defined above. Build the weakest partial order $<$ that contains the data dependencies: for all edge $s \to t$, $s < t$. Because of causality analysis (see Section 4.1.4) this order should exist.

```
let c_convol (ke, c_im) returns (c_res):
  forward ([im_i] = window «2 * 'k + 1» (c_im))
  returns (res = [res_i]):
    res_i = dot (ke, im_i)
  end
  l ++ _ = res;
  _ ++ r = res;
  dup_l of ['k]_ = !r;
  dup_r of ['k]_ = !l;
  res = dup_l ++ res ++ dup_r;
```

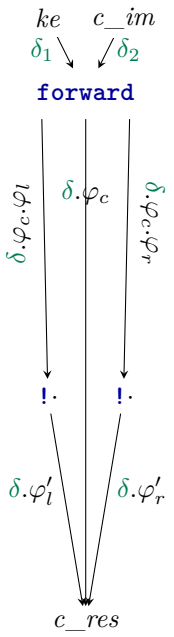
(a) The circular convolution operator

$$\varphi_c = \lambda[i{<}\nu]\,.\,@[i+k{<}n+2*k]\,.\,\mathbf{1}_{\mathtt{int}}$$
$$\varphi_l = \lambda[i{<}\kappa]\,.\,@[i{<}\nu]\,.\,\mathbf{1}_{\mathtt{int}}$$
$$\varphi_r = \lambda[i{<}\kappa]\,.\,@[i+n-k-1{<}\nu]\,.\,\mathbf{1}_{\mathtt{int}}$$

(b) Used projection functors



(c) Full DFG



(d) Reduced DFG

Figure 7.2: *Located data-flow graph (DFG) and reduction of the circular convolution*

2. *Propagate anti-dependencies.*

   (a) Eliminate all the interferences $n \not\parallel s \to t$ such that $n < s$ or $t < s$.

   (b) Given a *propagable* interference, defined as $n \not\parallel s \to t$ where $n < t$ (resp. $s < n$) add the relation $n < s$ (resp. $t < n$) to the partial order.

   (c) Repeat until a cycle is constructed or no *propagable* interferences remain.

   (d) If a non propagable interference $n \not\parallel s \to t$ remains, add the relation $n < s$ or $t < n$ to the partial order and repeat.

3. *Extend the partial order.* Build a total order from the extracted partial order.

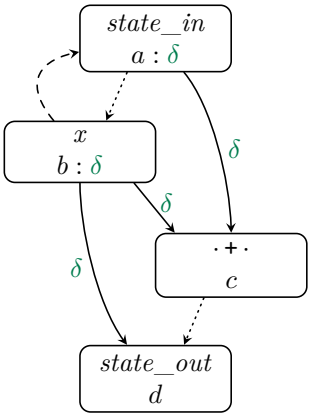
Implementation Located data-flow graphs are built at the level of scopes. Their extraction from the located abstract syntax tree is straightforward.

To describe the interfaces of scopes, the nodes that represent inputs and outputs are considered as computations. In the partial order, the input nodes are smaller that all non-input nodes. Similarly, the output nodes are larger that all non-output nodes. As a consequence, if a node interferes with an edge that is connected to an input or an output node, a cycle arises.

To handle registers, an expression `last` e is represented as an input which produces the value of `last` e and an output which depends on e. The above remark ensures that e and `last` e ar not used simultaneously.

To see this, consider the expression x + `last` x. The located data-flow graph is depicted at right. The names a, b, c and d label nodes. They are supplemented with the locations written by each operation. Here the writes and dependencies are all on the location δ. The straight dotted arrows depict the mandatory order between the input or output and the other nodes.

The node b writes location δ. It conflicts with the edge $a \to c$. Because we have $b < c$, the propagation of interference adds the relation $b < a$, that is depicted with a curve dashed arrow. This introduces a cycle $a \to b \to a$: the program cannot be scheduled.



7.2 Code Generation

The code generator translates a located and scheduled MADL program into C. The static analyses have extracted the three kinds of information. (i) The *typing* information describes the structure of values, in particular the size of arrays; (ii) the allocation information associates to each flow a formal location; and (iii) the schedule gives an order that is consistent with the semantics of the program.

At this stage, only a few tasks remain to generate imperative code. Notably: (i) the compiler must build a representation of the types and sizes; (ii) it must map the abstract locations to memory and accesses; (iii) it must translate the expressions of MADL into C instructions.

7.2.1 Principles

We target a modular code generation. Each operator is translated into a transition function and a description of its state and reset. Both may be empty.

The generated code inherits the structure of the MADL program. Each scope of the source code is turned into a C scope, that declares some local storage and perform computations. In the following, we present the mechanism used for the compilation of a unique scope.

Memory and accesses To illustrate our strategy of compilation, we present below a `swap-ends` operator. It expects an array of integers and returns the same array whose rightmost and leftmost elements have been swapped. The input and output arrays have the same location δ. This is enforced by the location annotations — · **as** $'l$ — (line 1) and because the variable c is the central part of the arrays u (line 3) and v (line 5). The locations of all the variables except *tmp* stem from δ. The variable *tmp* has an existentially quantified location δ'. Line 2 constrains the type the array so that the operator is monomorphic, for both sizes and types.

```
1 let swap_ends (u at 'l) returns (v at 'l):
2   c of [7]int;                    // Monomorphization
3   [|ul|] ++ c ++ [|ur|] = u;      // Decomposition of u
4   tmp,vl,vr = !ul,!ur,!tmp;       // Swap
5   v = [|vl|] ++ c ++ [|vr|];      // Reconstruction of v
```

```
val swap_ends: [9]int → [9]int
```

```
1 void swap_ends (int u[9]) {
2   int tmp;
3   tmp = u[0];
4   u[0] = u[8];
5   u[8] = tmp;
6 }
```

The generated function has a single argument (line 1), that is used for the input and output arrays. The body of the function declares a local variable `tmp` (line 2), that is used to perform the swap of elements `u[0]` and u[8] (lines 3-5). Because the values are integers, the three copies are turned into assignments instead of `memcpy`.

By contrast with MADL, the generated code is made of *instructions*. In particular, the '=' sign denotes an assignment instead of a physical equality as in MADL. The local variables of the generated scopes represent the existentially quantified locations of the MADL scope. In this example, there is only δ'. For readability, the compiler names locations by picking the name of one of the largest variables that are stored at that location. Hence the name `u` for ε.

Neither the decomposition of u (line 3) nor the reconstruction of v (line 5) need any instruction. In particular, the central part c of the array never appears in the generated code. The decomposition and reconstruction operations are indirectly reflected in the generated code: the array accesses `u[0]` and `u[8]` are computed from the locations of variables ul, ur, vl and vr. These locations have been constrained by structural operations (line 3,5 in the MADL program).

State Each scope has an associated set of state locations and a reset condition. For scopes with a `restart` or `reset` c reset condition, the reset call is introduced before the scope. For instance, the scalar product operator leads to the following code:

```
let dot <<n>> (u,v) returns (s):
  forward <<n>> ([ui] = u) ([vi] = v) returns (s = {sl}):
    sl init 0 += ui * vi
```

```
val dot: size i. «i» → [i]int, [i]int → int
```

```
void dot (size_t I, int *s, int *u, int *v) {
  *s = 0;
  for (int idx = 0; idx < I; idx++)
    *s += u[idx] * v[idx];
}
```

For a dynamic reset condition, the reset is guarded. We illustrate this with a modified version of the `integr` operator defined in Section 1.5.1.

```
let integr (rst, a) returns (v):
  forward ([ai] = a) returns (v = {s}):
    s init 0 += ai
  reset rst
```

```
val integr: size i. bool, [i]int → int
```

```
void integr (size_t I, int *a, bool *rst, int *v) {
  if (*rst)
    *v = 0;
  for (int idx = 0; idx < I; idx++)
    *v += a[idx];
}
```

As explained in Section 4.3.1, the scoping restriction on initialization allows a to insert state initialization outside of the scope. For iterations, it avoids branching inside loops.

The copy operation Scalar copies are translated into assignments. For larger values, the copies are implemented with the `memcpy` function. To illustrate it, we redefine the `swap_ends` operator so that it swaps the end slices of size 3 instead of the end elements.

```
let swap_ends (u at 'l) returns (v at 'l):
  c of [7]int;
  ul of [3]int;                  // Monomorphization
  ur of [3]int;
  ul ++ c ++ ur = u;             // Decomposition of u
  tmp,vl,vr = !ul,!ur,!tmp;      // Swap
  v = vl ++ c ++ vr;             // Reconstruction of v
```

```
val swap_ends: [13]int → [13]int
```

```
void swap_ends (int u[13]) {
  int tmp[3];
  memcpy (&tmp, &u[0], sizeof (int[3]));
  memcpy (&u[0], &u[10], sizeof (int[3]));
  memcpy (&u[10], &tmp, sizeof (int[3]));
}
```

Using `memcpy` to copy large values is only possible if they are stored as a contiguous chunk of data whose elements have the same order. If the source or the destination have complex locations, i.e., a view, implementing a copy operation is more involved. Consider the `c_transpose` operator below. The `c` stands for concrete or computation: this operator builds a new value in memory that contains the transposed matrix.

```
let c_transpose (m) = (n):
  n = !transpose (m) of [7][8]int
```

```
val c_transpose: [8][7]int → [7][8]int
```

```
void c_transpose (int m[8][7], int n[7][8]) {
  for (int idx = 0; idx < 7; idx++)
    for (int idx_2 = 0; idx_2 < 8; idx_2++)
      n[idx][idx_2] = m[idx_2][idx];
}
```

This operator has location scheme $\forall\, \delta_1 : [8]\,[7]\mathtt{int},\ \delta_2 : [7]\,[8]\mathtt{int}.\ \delta_1 \longrightarrow \delta_2$. Input and output matrices are stored at different locations. As for the previous example, some annotations make the operator monomorphic.

The connection between the MADL program and the generated code is not as straightforward as it might look. The two nested loops (lines 3,4) do not stem from the instantiation of `transpose`. They are generated because of the copy — `!·` —. It cannot be implemented with a `memcpy` because elements are not stored in the same order. Hence, the array is traversed and elements are copied one by one.

We presented in Section 7.1.1 various ways of implementing an in-place array reversal. Among them, the correct ones need two copies. We propose here an additional version that iterates on both halves of the array, in opposite directions, so as to swap the elements.

```
1 let rev (a at 'l)
2 returns (b at 'l):
3   b = ! reverse (!a) of [42]int
```

```
val rev: [42]int → [42]int
```

```
1 void rev (int a[42]) {
2   int tmp[42];
3   for (int idx = 0; idx < 42; idx++)
4     tmp[-1*idx+41] = a[idx];
5   memcpy (&a, &tmp, sizeof (int[42]));
6 }
```

```
1 let rev (a at 'l)
2 returns (b at 'l):
3   forward ([al] ++ reverse ([ar]) = a)
4   returns (b = [bl] ++ reverse ([br])):
5     tmp, bl, br = !al, !ar, !tmp of int
```

```
val rev: size i. [2*i]int → [2*i]int
```

```
1 void rev (size_t I, int *a) {
2   for (int idx = 0; idx < I; idx++) {
3     int tmp;
4     tmp = a[idx];
5     a[idx] = a[2*I+-1*idx-1];
6     a[2*I+-1*idx-1] = tmp;
7   }
8 }
```

Both versions are less general than the array reversal defined in Section 7.1.1. They apply only to array of integers. Moreover, the size is fixed in the version on the left. Section 7.2.2 explains why. In the version on the right, the size of the arrays must be even, because they are cut in two equally sized parts.

In the version on the left, a temporary array is allocated (line 2). One of the copies is implemented with a `memcpy` because the data are stored in the same order (line 5). As for the `c_transpose` example, the other needs a dedicated loop (line 3).

The version on the right only needs a scalar temporary value. It is local to the iteration scope. At iteration i, the body of the `for` loop swaps the elements of `a` at index i and $2n - i - 1$ where $2n$ is the size of the array.[1]

7.2.2 Polymorphism

Operators are generic in sizes, types and locations. To generate modular code, this polymorphism must be encoded in some way in the generated code.

Sizes The second version of the `rev` operator above has already illustrated size polymorphism. Generalized sizes are turned into additional arguments of type `size_t` and arrays are passed as pointers. We have not explored the use of the variable-length arrays of the C language.

For multi-dimensional arrays, using pointers require care. This relies on a convention for the order of elements. We chose the usual row-major order. Moreover accesses to multi-dimensional arrays of generic size cannot rely on pointer arithmetic — `m[i][j]` —, since the dimension of sub-arrays does not appear in their types. Hence, indexes are computed by the compiler. This is illustrated with the concrete transposition example we have used before, where the sizes are omitted in the type annotation.

```
1 let c_transpose_p (m) = (n):
2   n = !transpose (m) of [_][_]int
```

```
val c_transpose_p: size i j. [i][j]int → [j][i]int
```

```
1 void c_transpose_p (size_t J, size_t I, int *m, int *n) {
2   for (int idx = 0; idx < I; idx++)
3     for (int idx_2 = 0; idx_2 < J; idx_2++)
4       n[idx*J+idx_2] = m[idx_2*I+idx];
5 }
```

In the following code snippets, we mostly use integer sizes for multi-dimensional arrays in order to simplify projections and improve readability.

[1] Because the scheduling is not connected to the compilation chain, we were careful to write the swap (line 5) in the sequential order.

Types The code generator does not support polymorphic types for now. They could be represented at runtime by their size. However, several difficulties arise:

- Values of abstract types must be represented as an untyped pointer (`void *`) and static casts must be inserted.
- Because of alignment requirements in C, the size of a structure is not the sum of the size of its fields, it can be larger. For instance, the type $\texttt{bool} * \alpha$ would be represented by types with different sizes if $\alpha = \texttt{bool}$ or $\alpha = \texttt{int}$.

We propose a restricted support for compound data types with polymorphism. A generalized type α would appear only in a data-structure whose elements are all of type α, e.g., $\alpha * [\nu]\alpha$. This has not been implemented so far.

Locations Operators are generic in locations. In a location scheme $\forall \overrightarrow{\nu}.\ \forall \overrightarrow{\alpha}.\ \forall \overrightarrow{\delta : \tau}.\ \text{«}\overrightarrow{\eta}\text{»}\ \theta$, the generic location variables — $\overrightarrow{\delta : \tau}$ — represent the locations that are managed by the calling context. They are used for the arguments, the results and the state of the operator.

The location discipline allows for complex instantiation of generalized location variables. To support this in the generated code, functions would need a runtime description of indirections, i.e., of projection functors, so that they can access the instantiated locations. Such dynamic indirections would significantly alter the performance of the generated code. For that reason, the compiler limits instantiations of computational operators to locations that can be represented with a single pointer. Informally, these *single-piece* locations are raw data starting at an offset from the location variable they come from. This includes:

- Projections. For instance, in `[| a, b |] =` m, if m is given a single-piece location, both a and b have single-piece locations too.
- Array slices. For instance, in `concat (`l`, `r`) =` u, the locations of the left and right parts of the array u are single-piece if u has a single-piece location.

Currently, location inference checks that the universally quantified location variables of instances of operators that compute, i.e., the **`func`** and **`node`** operator kinds, are instantiated with *single-piece* locations. The following instantiation of the `dot` operator is rejected:

```
1 let rdot <<n>> (u, v) returns (s):
2   s = dot <<n>> (u, reverse (v))
```

```
Error: Non-single piece location
```

This check is performed after location inference. The single-piece constraints could guide location inference so that instantiated locations are indeed single-piece. For instance, the above `rdot` operator would meet the instantiation constraints if it was given the following type scheme:

$$\forall \nu.\ \forall \delta_1 : [\nu]\texttt{int},\ \delta_2 : [\nu]\texttt{int},\ \delta_2 : \texttt{int}.\ \text{«}\nu\text{»}\ \delta_1,\ \delta_2.\texttt{reverse}\ \nu\ \texttt{int} \longrightarrow \delta_3$$

The operators that must be instantiated with non single-piece locations should be inlined. This has not yet been studied.

Context Apart from the representation of values, producing transition functions that are generic in sizes and types raises a second issue. Local data with a polymorphic type cannot be statically allocated in the body of a polymorphic transition function. They are not yet supported.

Dynamic allocation is not an option in the context of safety critical embedded software. Hence, static allocation must be deferred to the monomorphic instantiation of a polymorphic operator. To do so, operators would be represented with functions that expect a state and a *context*. The context is statically allocated workspace where temporary data with polymorphic types would be written.

Contexts resembles states. However, they can be allocated in the stack frame of the monomorphic caller since they do not need to be transmitted from one synchronous cycle to the other. Moreover, multiple contexts may share the same locations, e.g., using a C union, since they are not used simultaneously.

As for the state, existentially quantified size variables cannot be handled by contexts. These sizes, such as iteration indexes, are intrinsically dynamic. More advanced mechanisms are required to allow a static allocation with existentially quantified sizes.

7.2.3 Examples

Linear algebra We implement below the usual definition a matrix multiplication. The rows of the first argument and the column of the second one are traversed to compute scalar products.

```
1 // Matrix product, non-vectorizable
2 let mat_mat (a, b) returns (c):
3   forward <<5>> [i] ([ai] = a),
4           <<3>> [j] ([bj] = transpose (b))
5   returns (c = [[cij]]):
6     forward <<7>> [k] ([aik] = ai) ([bkj] = bj)
7     returns (cij = {s}):
8       s init 0 = last s + aik * bkj
```

```
val mat_mat: [5][7]int, [7][3]int → [5][3]int
```

```
1 void mat_mat (int a[5][7], int b[7][3], int c[5][3]) {
2   for (int i = 0; i < 5; i++)
3     for (int j = 0; j < 3; j++) {
4       c[i][j] = 0;
5       for (int k = 0; k < 7; k++)
6         c[i][j] += a[i][k] * b[k][j];
7     }
8 }
```

The index captures — `[i]`, `[j]` and `[k]` — only serve as naming hints for the compiler, that otherwise generates numbered `idx` variables: `idx`, `idx_2`, *etc.* The accumulator — s — of the inner iteration is a local state. This allows for using different locations at each iteration of the outer `forward`. Hence, the sum is directly computed in the relevant element of the result matrix.

As we emphasize in Section 1.3.3, the above implementation is not vectorizable: the inner loop is necessarily sequential. We propose below a vectorizable version of the matrix product. It is defined with a three level iteration, that accumulates on the second dimension: `[{[`cij`]}]`.

```
1 // Matrix product, vectorizable
2 let mat_mat (a, b) returns (c):
3   forward [i], [j] returns (c0 = [[c0ij]]): c0ij = 0 end
4   forward <<5>> [i],
5           <<7>> [k] ([[aik]] = a),
6           <<3>> [j] ([[bki]] = b)
7   returns (c = [{[cij]}] init c0):
8     cij +=  aik * bki;
```

```
val mat_mat: [5][7]int, [7][3]int → [5][3]int
```

```
1 void mat_mat (int a[5][7], int b[7][3], int c[5][3]) {
2   for (int i = 0; i < 5; i++)
3     for (int j = 0; j < 3; j++)
4       c[i][j] = 0;
5   for (int i = 0; i < 5; i++)
6     for (int k = 0; k < 7; k++)
7       for (int j = 0; j < 3; j++)
8         c[i][j] += a[i][k] * b[k][j];
9 }
```

In this version, the accumulator must be initialized out of the three iteration layers (line 3) because the inner iterations are resumed.

Convolution We gave in Section 5.3.2 a cryptic implementation of a bidimensional convolution. We show the generated code below. For this compiled version, we set the dimensions of the matrices and changed some names to embellish the generated code.

```
let convol_2D <<k, l>> (ke, im) returns (res):
  ke of [4][5]_;
  im of [27][29]_;
  forward [i], [j] returns (r0 = [[r0]]): r0 = 0 end
  forward [i], [k] ([[ke]] = repeat (ke)) ([[im]] = window <<k>> (im)),
          [j], [l] ([[ke]] = repeat (ke)) ([[im]] = window <<l>> (im))
  returns (res = [{[{res}]}] init r0):
    res += ke * im
```

```
val convol_2D: «4, 5» → [4][5]int, [27][29]int → [24][25]int
```

```
void convol_2D (int im[27][29], int ke[4][5], int r0[24][25]) {
  for (int i = 0; i < 24; i++)
    for (int j = 0; j < 25; j++)
      r0[i][j] = 0;
  for (int i = 0; i < 24; i++)
    for (int k = 0; k < 4; k++)
      for (int j = 0; j < 25; j++)
        for (int l = 0; l < 5; l++)
          r0[i][j] += ke[k][l] * im[i+k][j+l];
}
```

The `r0` matrix in the interface is the result of the convolution. Our naming algorithm did not choose the best option here. Neither the `repeat` operator nor the `window` operator result in any value. They serve as a declarative way to introduce index computations in accesses.

Recursive array aggregation All the complexity of partial aliases is contained in location inference. For the code generation, they are handled as any other arrays. To illustrate recursive array accesses, we generate the code for the regular version of Pascal's triangle computation, defined in Section 5.2.2. As in the above example, we name indexes and give fixed values to sizes to make the resulting code more readable

```
let pascal () returns (m):

  // Monomorphization
  m of [9][9]_;

  // Location inference helpers: tell how to deduce r and c locations from m.
  transpose (m) at cons ('c, _);
                m  at cons ('r, _);

  // Computations
  d = 1;
  forward <<1:>> [i] returns (c at 'c = cons (d, [c])): c = 1 end
  forward <<1:>> [j] returns (r at 'r = cons (d, [r])): r = 1 end
  forward <<1:>> [i] (cons (_, [r]) = c) returns (m = cons (r, [m]) as snoc (_, c)),
          <<1:>> [j] (cons (_, [r]) = c) returns (m = cons (r, [m]) as snoc (_, c)):
    m = r + c
```

```
val pascal: [9][9]int
```

```
void pascal (int m[9][9]) {
  m[0][0] = 1;
  for (int i = 1; i < 9; i++)
    m[i][0] = 1;
  for (int j = 1; j < 9; j++)
    m[0][j] = 1;
  for (int i = 1; i < 9; i++)
    for (int j = 1; j < 9; j++)
      m[i][j] = m[i-1][j] + m[i][j-1];
}
```

A novel element is used here: the special `«1:»` size parameters of the `forward` iterators specify an offset for the index. We could also specify the upper bound with `«1:9»`. This offset is used in the generated
kwfor loop. Taking into accound the performed accesses, it may repeated index computations (`i+1` and `j+1` in this case).

As a last example, we propose an in-place, vectorizable implementation of the Cholesky decomposition, that we introduced in Section 5.2.1. The algorithm computes column by column. It accumulates the scalar products (line 18-24) in the decomposed matrix, by initializing the accumulator (lint 19) with the lower part of the column of `m`, hence directly computing the intermediate result $m - l[:j,j] \cdot l[:j,i]$ in place.

```
1  let head ([|h|]++_) returns (h)
2
3  // Cholesky-Crout (column by column) vectorizable, in-place
4  let cholesky (m at 'l) returns (l at 'l):
5
6    // Monomorphization
7    m of [10][10]_;
8
9    // We traverse [m] and build [l] by column
10   forward [j:size j] ([mF_j] = transpose (m))
11   returns (transpose (l) = [lF_j] as cF__):
12
13     // Splitting matrix and vectors in [ Top | Diag | Bot ] parts
14     _ ++ mB_j = mF_j;
15     _ ++ cB__ = transpose (cF__);
16
17     // Compute m[i,j] - dot (l[j,:j], l[i,:j]) for j <= i <n
18     forward [k] ([cB_k] = transpose (cB__))
19     returns (cons (sDjj, sB_j) = {s} init mB_j):
20       forward <<j:>> [i] ([cBik] = cB_k)
21       returns (s = [si]):
22         si -= head (cB_k) * cBik
23       resume      // Mandataory to allow access 'last si' (si -= ...)
24     end
25
26     // Upper part
27     forward <<j>> [i] returns (lT_j = [lTij]): lTij = 0 end
28
29     // Diagonal element
30     lDjj = sqrt (sDjj);
31
32     // Lower part
33     forward <<j+1:>> [i] ([sBij] = sB_j)
34     returns (lB_j = [lBij]):
35       lBij = sBij / lDjj
36     end
37
38     // Build the full column
39     lF_j = lT_j ++[| lDjj |]++ lB_j;
```

```
val head: size i. type a. [1+i]a → a
val cholesky: [10][10]int → [10][10]int
```

```
1  void cholesky (int l[10][10]) {
2    for (int j = 0; j < 10; j++) {
3      for (int k = 0; k < j; k++)
4        for (int i = j; i < 10; i++)
5          l[i][j] -= l[j][k] * l[i][k];
6      for (int i = 0; i < j; i++)
7        l[i][j] = 0;
8      l[j][j] = sqrt (l[j][j]);
9      for (int i = j + 1; i < 10; i++)
10       l[i][j] /= l[j][j];
11   }
12 }
```

Conclusion

This chapter sketched the last steps toward the generation of imperative code. This part is at its (very) early stage.

Scheduling MADL prioritizes memory over scheduling: memory locations are inferred first, independently of schedulability concerns. So as to find a schedule that meets both the data-flow constraints and the location ones, we delineated two phases analysis.

1. Some local requirements apply at the level of memory locations. They ensure that each operation may be correctly implemented with the inferred locations. This allows to model operations as atomic computations that modifies a set of locations.

2. A global analysis of dependencies builds an order that is compatible with locations. To do this, it adds to the data-flow graph some anti-dependencies that arise a notion of interference.

Both steps heavily rely on the properties of projection functors, in particular their injectivity, inclusion and disjunction. Because these properties are over-approximated, we propose a representation of dependencies based on sets of projection functors. This allows to approximate more precisely there properties symbolically.

A location-less analysis of schedulability The scheduling analysis may fail either for the local or global requirements. In our experiments, reports for scheduling conflicts that arise because of locations constraints were barely usable. In Lucid Synchrone [CP01], a causality analysis applies to the input programs. It ensures that a schedule will exists. For MADL, the simple causality analysis sketched in Section 4.1.4 provides more comprehensible errors, but it cannot account for the constraints that arise from location.

Extending such a causality analysis to capture some of the constraints imposed by the underlying memory model is an important direction for future work. It would allow to report earlier incorrect programs such as $x + \texttt{last}\ x$ that cannot be scheduled because of the memory locations of x and $\texttt{last}\ x$.

The role of an extended causality analysis would resemble the one of initialization check: it should approximate the memory allocation so as to give a precise estimate of the expressions that conflict. We anticipate that it could benefit from the structural inclusion defined in Section 4.3.4 for the verification of initialization.

Code generation The generation of imperative code from a located and scheduled MADL program is mostly a matter of representation of values and accesses in the target language. The components of the generated code are directly related to the source program or the inferred locations. The local variables of the generated code are defined by the existentially quantified locations. For the instructions, the mapping of MADL programs to imperative code follows the intuitions below:

- The groups and compose constructs are description-level wiring. They do not exist in the generated code, and they have no semantic content (causality, initialization, *etc*).
- The other structural constructs, such as the concatenation, are memory-level wiring. They are realized by reading or writing memory through statically managed indirections.
- The computational constructs are turned into instructions. They read and write memory through complex accesses induced by locations.

The meaning of the illustration of Chapter 6 (reproduced below) shall now be clearer: the structural part of programs, the *views*, give rise to some existentially quantified locations that are turned into variables and complex accesses that are glued to the *computation* to make target code instructions.

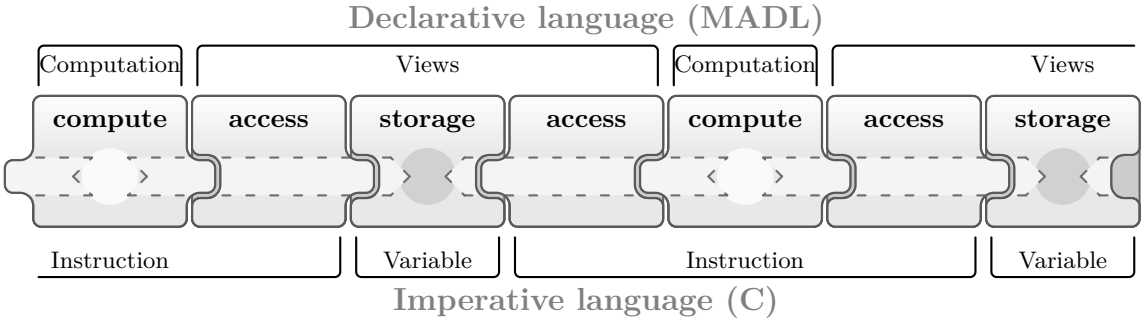


Figure 7.3: *Anatomy of MADL programs: memory, accesses and computations (repeated from Figure 6.2)*

8 Conclusion

The synchronous language SCADE is used to design and implement embedded software for safety critical systems. SCADE programs are mathematical specifications of stream functions that are compiled to sequential code. If compile-time verifications succeed, the generated code is deterministic and executes without deadlock in statically bounded memory and execution time. The language is purely functional and first order.

We study the specification and compilation of synchronous programs that contain data-intensive computation expressed with arrays. For such applications, the quality of the generated code is essential. In particular, implementations should avoid copies and favor memory sharing. The dynamic solutions found in general purpose programming languages are not applicable for SCADE because memory must be statically managed.

8.1 Memory-Aware Declarative Language

The idea of a declarative language with a precise control of memory was born out of the observation that the purely functional nature of SCADE serves almost exclusively safety purposes. On the expressiveness side, the targeted software prevents the use of high-level programming features such as first class functions, so as to keep the generated code statically bounded in memory and execution time. On the compilation side, the purely functional description does not allow to specify allocation nor scheduling, which is crucial in the context of embedded software that run on platform with limited resources.

The lack of control of the generated code is becoming increasingly limiting as the targeted execution platforms become diverse. In particular, the programming of data intensive applications based on arrays with SCADE encourages the generation of code for hardware with parallelism capabilities, either at the level of operations, e.g., vectorized instructions, or at the level of larger tasks, e.g., GPUs.

We propose to make the constraints of the targeted software infuse the source language. Chapter 4 presents a first order synchronous language named MADL, that provides the core constructs of SCADE. It conciliates a declarative description of programs with a specification of their implementation concerning memory mapping. Instead of leaving the choice of memory allocation to the compiler, we propose a type system for locations that specifies how memory usage is deduced from the program. This is similar to the clock type systems used in synchronous languages, in particular that of SCADE, that organises the computation of stream values across cycles. In short, clocks specify when streams are computed, locations specify where.

Describing sizes in types As a first and necessary step toward a static description of locations, the shape of values must be described precisely, in particular the size of arrays. We propose in Chapter 2 an extension of the ML type discipline and Hindley-Milner inference to introduce a notion of sizes into types. It relies on a rudimentary form of refinement types [XP99; Fla06] with restricted sub-typing. The simple types of ML-like type systems are supplemented with two refinements of the integer type that are parameterized by a size η: singleton types $\langle\eta\rangle$ and interval types $[\eta]$. Refined types may be compared to the integer type but not between them. This limitation is key to keep size constraints simple. Instead of inequalities, size relations are only equalities.

The size language is made of multivariate integer polynomials. This choice allows to express most of the size relations and accesses that are necessary for linear algebra operation and signal

processing. Moreover, it is suitable for static verification and formal manipulation. The proposed type system is applicable in a wider context that the one or MADL. In particular, it is compatible with higher order constructs.

This type system suffers two weaknesses that are only apparent. First, the semantics is not type erasable, more precisely, it depends on sizes. Second, no principal types exist. Both weaknesses stem from the pursued usage of sizes. For the former, we want to omit iteration lengths and constant array sizes, by relying on the context to infer them. The latter comes from the extends of our size language, i.e., polynomials, and the need for a complete size information for polymorphic declarations. This is mitigated by adding an extra requirement on size inference. It must reject all ambiguous programs, i.e., the ones that may have multiple semantics. This property comes down into banning arbitrary size substitutions.

A static description of locations Memory is described as a collection of typed, non aliasing data without indirections. Locations are built by viewing these chunks of data through the lenses of *projection functors*, that we introduce in Chapter 3. Projection functors represent statically managed indirections. By mapping the projections over a given type, e.g., a matrix, to projection over another type, e.g., a vector, they allow to refer to the same data with a different structure. These projection functors are formally represented, which allows to compose and compare them.

The correctness of projection functors is based on polynomial inequalities between computed indexes and sizes. It cannot be established in general. For that reason, they are provided through a few correct built-in functors that are composed to build new ones. Depending on the needs, projection functors are handled either in a symbolic way, i.e., as a composition of named primitives, or in a formal way, i.e., as terms of a language. Functor properties and relations, e.g., injectivity and inclusion, are preferably established on a symbolic composition of these atomic functors, but their formal representation is used as fallback for comparing and simplifying them.

Location discipline Memory is indirectly specified in MADL. Instead of indicating where data are stored, the language distinguishes between the parts of the program that produce new values in memory from the ones that access already existing values. The former are called the *computational constructs* while the latter are *structural*. Among the computational constructs, the language provides an explicit copy operation. It is the only way to duplicate values. The structural constructs represent wiring, i.e., viewing existing data with a different structure. The emblematic example of *structural* expressions is concatenation. Instead of building a large array by copying the two parts, it requires that they are stored side by side. Hence, the resulting concatenation exists without additional computation. This eliminates the cost of declarative constructions of compound data, which require separate definitions of the parts before aggregating them.

The language also provides *location annotations*. They introduce location constraints between unrelated parts of the program, by requiring that they share the same location. These annotations resemble and extend the proposal of [Gér+12]. The verification of locations is supported by a dedicated type system presented in Chapter 6. It is accompanied by an incomplete inference algorithm.

Scheduling MADL favors memory specification over scheduling: locations are inferred first, then the program is analysed to find an execution order that complies with the data-flow dependencies and location sharing. A valid schedule must ensure that values read from the memory have not been corrupted by other writes.

This analysis is sketched in Section 7.1. It must to decide whether locations, described with projection functors, have intersecting images or not. Such a property can only be derived in simple cases. Because the ability to find a schedule for MADL programs depends on the precision of the intersection estimate, we propose a symbolic representation of dependencies that provides a more precise approximation.

Non-optimizing code generation The generation of sequential code from MADL does not require any optimizations. It has been a design choice. The usage of memory in the generated code is driven by the location type system. This system defines which data must be declared locally and how values are retrieved from memory. Instructions are built out of the *computational* constructs of the source program. They access memory through complex projections that stem from the location of read or written flows. These instructions are ordered according of the memory aware schedule.

This code generation contrasts with the current compilation of SCADE that relies on some internal optimizations that are inevitably fragile. During source-to-source rewriting, SCADE variables progressively shift from a declarative meaning, i.e., identifiers, to an imperative one, i.e., memory locations. Similarly, equations are smoothly transformed into assignments. In MADL, the location system draws a clear separation between the declarative world, made of expressions and the imperative one, defined by location annotations. Declarative equalities are imperative aliases and the copies represent assignments.

Two levels of semantics MADL is primarily designed to allow a memory specification to coexist with a declarative description of programs. We sketched in Section 4.2 the principles of a *two-sided* semantics for MADL. It defines an operational reduction of the terms in an abstract memory used to store and retrieve values. This memory ensures that accesses are consistent with a declarative semantics: a variable can only be read if its location has not been modified since it was defined. Such a data-flow aware memory have been proposed to detect memory corruption attacks [CCH06].

To emphasize the declarative side of MADL. we would like to provide a purely functional semantics, i.e., that is independent of locations. It is sketched in Section 4.3. Compared to the unusual semantics for synchronous languages, the semantics features two distinctive elements. (i) It allows for inverting certain operations, in order to evaluate complex patterns. (ii) Its initialization phase must propagate initial values through the structural constructs of the language.

Correctness of array programs We first considered a type system with sizes for safety purposes. In functional languages, the correctness of array accesses builds on the use of intensional operations. They provide valid access schemes under some assumptions about sizes. Our type system goes one step further, by ensuring *size consistency*. All the sizes that should match are indeed equal. However, the correctness of intensional array operations also relies on some size inequalities, e.g., a sampling step must be positive. These properties cannot be proved with our size equality-based type system.

Yet, this is not an issue in the context of SCADE: top-level programs are monomorphic. Hence, sizes get concrete values at compile-time, which allows to check for correctness by comparing integer constants. We rely on the same mechanism for the remaining constraints. The value of our type system is twofold. (i) It detects size inconsistencies in a modular way and (ii) it provides the necessary size information for the subsequent phases of compilation, notably the analysis of locations.

8.2 Array Iteration as Stream Processing

We proposed in Chapter 5 a language construct `forward` to iterate over arrays. It is largely inspired by LUCID [AW77] and SISAL [FCO90]. These languages provide declarative descriptions of iteration by viewing arrays as finite streams. The body of iteration is thus a stream expression.

Arrays as (in)finite streams We propose to unify the infinite streams of SCADE with the finite streams used for array iteration in LUCID or SISAL. Intuitively, a flow of arrays is handled as a faster flow, that comes in bundles. Iteration amounts to computing multiple steps at each synchronous cycle. The idea of locally faster flows in synchronous language has been proposed for verification purposes [Mik05] and to express systems with different reactive time scales Mandel, Pasteur, and Pouzet [MPP15]. However, its use for array iteration is new.

By viewing the iteration cycles as the consecutive instants of a single flow, accumulation is expressed using the flow constructs of the language, `last` and `init`. For SCADE, this would allow to use the complex activation constructs such as automata when iterating. In the context of MADL, the benefit lies in the memory specification. Accumulations are submitted to the same location constraints as accesses to the state, i.e., they do not introduce copies. This construct has been prototyped in the context of ZRUN, an interpreter for a synchronous language with advanced constructs such as automata [Col+23].

Recursive aggregation We extend the proposed iteration construct to allow *recursive array aggregation*. It aims at expressing in a declarative way efficient implementations of algorithms that compute the value for a given index using the previously computed elements. Pivot-based

algorithms such the Cholesky decomposition are typical use-cases. Our proposal is a restricted form of recursive array comprehension [Sin+17]. It avoids the need for a useless initialization while ensuring that only the already computed elements are accessed. From a stream point of view, these partial aliases give access to a growing part of the history.

8.3 Perspectives and Future Work

The formalization of MADL and the development of its compiler prototype are incipient.

On the experimentation side, the support of several features of the language is to be improved. Porting and extending the previous scheduling experiments is certainly the most pressing direction for future work, alongside a decent support for operator instantiation in the code generator.

On the formalization side, the two-sided semantics should be fully described and the functional semantics remains to be done. The language would also benefit from a formalization of the location-less initialization analysis. To this end, the study of a location-less extended causality will probably help understanding a relevant way to approximate locations from the source programs.

Experimentation The presented material was supported by several prototypes. We first implemented front-ends and type inference algorithms for languages that resemble the core ML introduced in Chapter 2. The development of MADL was started later. Its compiler, written in OCaml, is about 12k lines of code long, excluding blank lines and comments. Among them, the current compilation pipeline, that is used for compiling the examples of this thesis, should account for approximately 4k to 6k LoC. The remainder comprises several attempts of language constructs and internal representations. One of them served notably as an exploratory support for scheduling. The unified handling of expressions presented in Chapter 4 is not entirely implemented. We expect it will help eliminating some code redundancies and ease compilation.

Language Extensions This first proposition of a memory-aware declarative language has left aside various important constructs.

The support for arrays is limited, although sufficient for a large spectrum of applications. The type system has been designed to describe bounded integers, using the interval refinement $[\eta]$. This lays the groundwork for dynamic unguarded array accesses, either for reading or writing. These constructs already exists in Scade— `a.[i]` and `(a with [i] = e)` respectively — but they require dynamic bound checking. The latter is a functional update *a la* OCaml. In a MADL extension, such a construct would be copy-less, allowing for an in-place modification of the array. We anticipate that typing should extend smoothly. However, the description of locations will be more involved. As for iteration indexes, it will rely on some runtime sizes, that should be described with existential quantification.

We did not study conditionals. Although they could be provided as an external ternary operation, conditionals deserves a special treatment in order to avoid copies. For large data-structures such as array, conditionals would benefit from pointer manipulations, that is, allowing for indirection in data, at least at top-level. This opens a vast and surely interesting area for extending the location type system and the dependency analysis.

Synchronous languages such as Scade organize the computation of stream values in time with *clocks*, that allow to activate parts of programs at selected synchronous cycles. The interactions between the clock and location type systems should give fruitful source of developments.

Memory specification Using location annotations in MADL is uneasy, because these annotations are contained in expressions. The language would benefit from a separated space for specifying memory. This could be provided as a dedicated scope that contains location equations, without semantic meaning. Although some annotations are necessary, for instance, in interfaces, such a dedicated context would benefit the separation of algorithms and implementations.

Location sharing is currently limited by types. Two values of different base types, e.g., a boolean and an integer, may not share a common location. Because locations only have an implementation purposes, they could be given untagged union types. This feature would help supporting polymorphic code generation by using contexts, i.e., a local workspace allocated by the calling operator regarding the concrete values of the generalized sizes and types.

To help the programmer in specifying memory, error diagnostics about location would need investigations. In particular, a readable representation of location is still to be found. It should

be directly connected as much as possible to the elements of the source program. The intrinsic complexity of location related errors lends importance to location-less verifications, for instance for initialization and schedulability.

A multidimensional construct The extension we have proposed for multidimensional iteration is insufficient. On the language design side, it is only understandable from its elaboration to unidimensional iterations. The multidimensional `forward` construct should be restricted in some way, possibly by requiring a single return clause instead of one per dimension of iteration.

On the compilation side, a built-in multidimensional iteration construct may provide more advance optimization and transformation chances.

An intermediate language MADL has been designed as a low-level declarative representation of a synchronous data-flow core. It aims at delaying as long as possible the translation of purely functional stream programs into a sequential form by allowing to perform most implementation optimizations on the MADL program itself. This ambition is underpinned by two features: (i) the code generation is straightforward and predictable and (ii) the language allows to specify memory.

Copy elimination and location sharing would be the first candidate for a MADL-level optimization. It would rely on the location and schedule information to increase memory sharing while preserving the existence of a schedule. Incremental location type system and schedule would benefit the exploration of possible sharing opportunities, in order to insert memory constraint without having to recompute the complet location and schedule information.

Chapter 5 illustrates how MADL allows to describe different implementations of the same algorithms, e.g., matrix multiplication. The high-level rewriting techniques for data-intensive applications found in tensor compilers [Clé22] should be applicable to this representation. Adapting the automated or guided transformations techniques for array iterations, such as reordering, fusion and tiling, would be an important step toward the founding principle of Halide [Rag14]: decoupling algorithms from schedule. Algorithms would be expressed in their simplest form, while schedule annotations would guide MADL-level transformations.

Memory and arrays in a higher-level synchronous language Conciliating the specification of memory with a purely functional, stream-based synchronous language such as Scade requires a few concessions over our strictly copy-less discipline.

MADL programs specify the places where copies happen. For synchronous languages such as Scade, that has no notion of location, the control of memory would preferably be achieved by specifying the place where copies cannot happen. For instance, registers and functional array updates are candidates for such specification. However, such annotations would be useless without a minimal set of operations that do not copy, e.g., variable declaration or array projection.

Because of the pure data-flow point-of-view of Scade, our flexible treatment for initialization and access to the state in multidimensional iterations is not applicable. This will not restrict expressiveness, but it may require to separate multidimensional iterations into nested iterations and name intermediate flows.

The successor of Scade, named Swan, is under development at Ansys. It will provide a `forward` iterator similar to the one of MADL. This will give access to the rich sequential sequential constructs of the language (activation constructs, automata, ...) for iterating over arrays. Recursive array aggregation is planned for a future extension.

A Complements on the Type System

1.1 Properties of Type System

In presence of recursion, the correction proof of a type system toward a big step semantics cannot be derived from usual progress and type preservation properties of the reduction, as done for small step semantics, because blocked and diverging terms are undistinguishable. However, a general analysis of big step semantics for soundness conditions [Dag+20] provides a few similar properties to check in order to verify soundness. Because of non-deterministic semantics, two kinds of corrections are distinguished:

- *Soundness-must*: None of possible reduction is blocked
- *Soundness-may*: At least one of possible reduction is not blocked

A mechanized derivation of these global properties (soundness-must in our setting) from local ones was proposed by Dagnino et al. [Dag+20]. They follow from usual properties on the type system that we detail first.

1.1.1 Normalization and Preliminary Lemmas

Definition A.1 (Normalized typing derivation)**.** A typing derivation $\Gamma \vdash e : \tau$ is *normalized* if the instances of rule T-SubType appear in one of the following position:

Bottom most rule $\text{T-SubType}\ \dfrac{T \quad S}{\Gamma \vdash e : \tau}$	First premise of applications $\text{T-SubType}\ \dfrac{T_1 \quad S}{\Gamma \vdash e_1 : \tau_2' \to \tau'}$ $\quad \text{T-App}\ \dfrac{\qquad\qquad \dfrac{T_2}{\Gamma \vdash e_2 : \tau''}}{\Gamma \vdash e_1\ e_2 : \tau'}$
Coercions $\text{T-SubType}\ \dfrac{T \quad S}{\Gamma \vdash e : \tau'}$ $\text{T-Coerce}\ \dfrac{}{\Gamma \vdash e \triangleright \tau' : \tau}$	First premise of declarations $\text{T-SubType}\ \dfrac{T \quad S}{\Gamma,\ \boldsymbol{V} \vdash e : \tau}$ $\quad \text{T-Let}\ \dfrac{\qquad\qquad \dfrac{T_2}{\Gamma,\ x{:}\forall \boldsymbol{V}.\ \tau \vdash e' : \tau'}}{\Gamma \vdash \texttt{let}\ x_{\boldsymbol{V}} : \tau = e\ \texttt{in}\ e' : \tau'}$

Definition A.2. Given τ, τ_1, τ_2 types and S_1, S_2 derivations of the sub-typing relations $\tau_1 <: \tau$ and $\tau <: \tau_2$, $S_1 \ltimes_\tau S_2$ is the following sub-typing derivation of $\tau_1 <: \tau_2$, regarding τ shape:

- $\tau = \texttt{int}$ — Then S_2 is Refl $\dfrac{}{\texttt{int} <: \texttt{int}}$ and $\tau = \tau_2$. The derivation $S_1 \ltimes_\tau S_2$ is S_1.
- $\tau = \texttt{<}\eta\texttt{>}$ — Then S_1 is Refl $\dfrac{}{\texttt{<}\eta\texttt{>} <: \texttt{<}\eta\texttt{>}}$ and $\tau = \tau_1$. The derivation $S_1 \ltimes_\tau S_2$ is S_2.
- $\tau = [\eta]$ — Then S_1 is Refl $\dfrac{}{[\eta] <: [\eta]}$ and $\tau = \tau_1$. The derivation $S_1 \ltimes_\tau S_2$ is S_2.

- $\tau = \tau^d \to \tau^\gamma$ — Then $\tau_1 = \tau_1^d \to \tau_1^\gamma$, S_1 is FUN $\dfrac{\dfrac{S_1^d}{\tau^d <: \tau_1^d} \quad \dfrac{S_1^c}{\tau_1^\gamma <: \tau^\gamma}}{\tau_1^d \to \tau_1^\gamma <: \tau^d \to \tau^\gamma}$, $\tau_2 = \tau_2^d \to \tau_2^\gamma$, S_2 is FUN $\dfrac{\dfrac{S_2^d}{\tau_2^d <: \tau^d} \quad \dfrac{S_2^c}{\tau^\gamma <: \tau_2^\gamma}}{\tau^d \to \tau^\gamma <: \tau_2^d \to \tau_2^\gamma}$. The derivation $S_1 \ltimes_\tau S_2$ is FUN $\dfrac{\dfrac{S_1^d \ltimes_{\tau^d} S_2^d}{\tau_2^d <: \tau^d} \quad \dfrac{S_1^c \ltimes_{\tau^c} S_2^c}{\tau^\gamma <: \tau_2^\gamma}}{\tau_1^d \to \tau_1^\gamma <: \tau_2^d \to \tau_2^\gamma}$.

Corollary A.2.1. *The sub-typing relation $<:$ is transitive.*

Theorem A.3 (Normalization of typing derivation)**.** *If there exists typing derivation, there exists a normalized typing derivation.*

Proof. Typing derivations are normalized using the following rewrite rules:

$$\textsc{SubType}\ \frac{\textsc{SubType}\ \dfrac{P}{\Gamma \vdash e : \tau_2} \quad \dfrac{S_2}{\tau_2 <: \tau_1}}{\Gamma \vdash e : \tau_1} \quad \dfrac{S_1}{\tau_1 <: \tau}}{\Gamma \vdash e : \tau} \to \textsc{SubType}\ \frac{P}{\Gamma \vdash e : \tau_2} \quad \dfrac{S_1 \ltimes_{\tau_1} S_2}{\tau_2 <: \tau}}{\Gamma \vdash e : \tau}$$

$$\textsc{T-SbTy}\ \frac{\textsc{T-Abs}\ \dfrac{T}{\Gamma,\ x{:}\tau \vdash e : \tau_1'} \quad \dfrac{S}{\tau_1' <: \tau_1}}{\Gamma,\ x{:}\tau \vdash e : \tau_1}}{\Gamma \vdash \lambda x{:}\tau.\ e : \tau \to \tau_1} \to \textsc{T-SubType}\ \frac{\textsc{T-Abs}\ \dfrac{T}{\Gamma,\ x{:}\tau \vdash e : \tau_1'}}{\Gamma \vdash \lambda x{:}\tau.\ e : \tau \to \tau_1'} \quad \textsc{S-Fun}\ \dfrac{\textsc{S-Refl}\ \dfrac{}{\tau <: \tau} \quad \dfrac{S}{\tau_1' <: \tau_1}}{\tau \to \tau_1' <: \tau \to \tau_1}}{\Gamma \vdash \lambda x{:}\tau.\ e : \tau \to \tau_1}$$

$$\textsc{T-SubType}\ \frac{\textsc{App}\ \dfrac{T_1}{\Gamma \vdash e_1 : \tau_2' \to \tau'} \quad \textsc{T-SubType}\ \dfrac{\dfrac{T_2}{\Gamma \vdash e_2 : \tau_2} \quad \dfrac{S_2}{\tau_2 <: \tau_2'}}{\Gamma \vdash e_2 : \tau''}}{\Gamma \vdash e_1\ e_2 : \tau'} \quad \dfrac{S_1}{\tau' <: \tau}}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\downarrow$$

$$\textsc{T-SubType}\ \frac{\dfrac{T_1}{\Gamma \vdash e_1 : \tau_2' \to \tau'} \quad \textsc{S-Fun}\ \dfrac{\dfrac{S_2}{\tau_2 <: \tau_2'} \quad \dfrac{S_1}{\tau' <: \tau}}{\tau_2' \to \tau' <: \tau_2 \to \tau}}{\Gamma \vdash e_1 : \tau_2 \to \tau} \quad \dfrac{T_2}{\Gamma \vdash e_2 : \tau_2}}{\Gamma \vdash e_1\ e_2 : \tau}\ \textsc{T-App}$$

$$\textsc{T-Let}\ \frac{\dfrac{T_1}{\Gamma,\ \boldsymbol{V} \vdash e : \tau} \quad \textsc{T-SubType}\ \dfrac{\dfrac{T_2}{\Gamma,\ x{:}\forall \boldsymbol{V}.\ \tau \vdash e' : \tau_1'} \quad \dfrac{S}{\tau_1' <: \tau_1}}{\Gamma,\ x{:}\forall \boldsymbol{V}.\ \tau \vdash e' : \tau_1}}{\Gamma \vdash \texttt{let}\ x_{\boldsymbol{V}} : \tau = e\ \texttt{in}\ e' : \tau_1}$$

$$\downarrow$$

$$\textsc{T-SubType}\ \frac{\textsc{T-Let}\ \dfrac{\dfrac{T_1}{\Gamma,\ \boldsymbol{V} \vdash e : \tau} \quad \dfrac{T_2}{\Gamma,\ x{:}\forall \boldsymbol{V}.\ \tau \vdash e' : \tau_1'}}{\Gamma \vdash \texttt{let}\ x_{\boldsymbol{V}} : \tau = e\ \texttt{in}\ e' : \tau_1'} \quad \dfrac{S}{\tau_1' <: \tau_1}}{\Gamma \vdash \texttt{let}\ x_{\boldsymbol{V}} : \tau = e\ \texttt{in}\ e' : \tau_1}$$

This rewrite system terminates since instances of the rule T-SUBTYPE are either pushed downward or introduced in a normalized position. Moreover, any non-normalized positions for T-SUBTYPE rule reduces with one of the above rewrite rules. □

Lemma A.4 (Inversion)**.** *Let Γ, e, τ such that $\Gamma \vdash e : \tau$*

1. *If $e = {}^{\boldsymbol{S}}x$, then $\Gamma(x) = \forall \boldsymbol{V}.\ \tau'$ and $\tau'\ PColor\{\boldsymbol{S}/\boldsymbol{V}\} <: \tau$*
2. *If $e = e_1\ e_2$, then there exists τ' such that $\Gamma,\ x{:}\forall \varepsilon.\ \tau \vdash e_2 : \tau' \to \tau$ and $\Gamma,\ x{:}\forall \varepsilon.\ \tau \vdash e_1 : \tau'$*
3. *If $e = \lambda x{:}\tau.\ e'$, then $\tau = \tau_1 \to \tau_2$ et $\Gamma,\ x{:}\forall \varepsilon.\ \tau_1 \vdash e' : \tau_2$*
4. *If $e = n$, then $\tau = \texttt{int}$, $\tau = \texttt{<}\nu\texttt{>}$ or, if $0 \leq n < p$, $\tau = [\mu]$*
5. *If $e = \texttt{<}\eta\texttt{>}$, then $\texttt{<}\eta\texttt{>} <: \tau$*
6. *If $e = e \triangleright \tau'$, then $\tau' <: \tau$ and $\Gamma \vdash e : \tau'$*
7. *If $e = \texttt{let}\ x_{\boldsymbol{V}} : \tau' = e_1\ \texttt{in}\ e_2$, then $\Gamma,\ \boldsymbol{V} \vdash e_1 : \tau'$ and $\Gamma,\ x{:}\forall \boldsymbol{V}.\ \tau' \vdash e_2 : \tau$*

Proof. Case-based reasoning on expression shape and analysis of applicable rules. □

Lemma A.5 (Substitution). *The type of an expression is preserved by well-typed substitution:*

$$\left.\begin{array}{r} \Gamma,\ x{:}\forall \boldsymbol{V}.\ \tau' \vdash e : \tau \\ \Gamma,\ \boldsymbol{V} \vdash e' : \tau'' \\ \tau' <: \tau'' \end{array}\right\} \implies \Gamma \vdash e\{e'\{\boldsymbol{S}/\boldsymbol{V}\}/^{\boldsymbol{S}}x\} : \tau$$

Proof. A finite derivation tree of $\Gamma \vdash e\{e'/x\} : \tau$ is obtained by substituting in a finite derivation of $\Gamma,\ x{:}\forall \boldsymbol{V}.\ \tau' \vdash e : \tau$ every occurrence of rule VAR (in finite number) by a derivation of $\Gamma \vdash e' : \tau''$ (also finite) and an instance of SUBTYPE rule. Cares is required with environment that are distinct for each occurrence (they might be extended). This lemma is valid because we considered only name resolved terms such that no clashes occur. Thus, typing is preserved by extension of the environment since added variables are fresh and may not mask existing ones. □

Lemma A.6 (Canonical form). *Given a type τ, $\{\!|\tau|\!\}$ denotes $\{v \in \mathcal{V} | \vdash v : \tau\}$. Then,*

$$\{\!|\boldsymbol{int}|\!\} = \{n \mid n \in \mathbb{Z}\}$$
$$\{\!|\cdot \to \cdot|\!\} = \{\lambda \cdot{:}\cdot.\ \cdot\}$$

Proof. Case-based analysis on value's shape. □

1.1.2 Soundness Sufficient Conditions

To prove soundness and preservation of a type system toward a big-step semantics, Dagnino et al. [Dag+20] proposed a general reduction to three local properties. These sufficient conditions only require rule examination, while the induction is conducted by the generic construction. To help presenting them, we borrow the proposed syntax of inline format for instances of rules:

$$(e_1 \rightsquigarrow v_1, \ldots, e_n \rightsquigarrow v_n, e_{n+1} \rightsquigarrow v_{n+1}, e) \stackrel{def}{=} \frac{e_1 \rightsquigarrow v_1 \qquad \ldots \qquad e_n \rightsquigarrow v_n \qquad e_{n+1} \rightsquigarrow v_{n+1}}{e \rightsquigarrow v_{n+1}}$$

The $e_1 \rightsquigarrow v_1, \ldots, e_n \rightsquigarrow v_n$ are rule's *premises* and $e_{n+1} \rightsquigarrow v_{n+1}$ is the *continuation*, that produces the result of rule instance. For rules with no natural continuation, a trivial one is inserted: $v_{n+1} \rightsquigarrow v_{n+1}$.

Lemma A.7 ((S1) Local Preservation). *For any instance $(e_1 \rightsquigarrow v_1, \ldots, e_n \rightsquigarrow v_n, e_{n+1} \rightsquigarrow v_{n+1}, e)$, such that $\vdash e : \tau$, there exists $\tau_1, \ldots, \tau_{n+1}$ with $\tau_{n+1} = \tau$ such that :*

$$\forall k \in [\![1, n+1]\!],\ (\forall h \in [\![1, k-1]\!],\ \vdash v_h : \tau_h) \implies \vdash e_k : \tau_k$$

Proof. Case-base reasoning on instances de semantics rules, using extensively the inversion lemma:

E-SIZE $e = \nu$ — There is only the continuation $n \rightsquigarrow n$ that verifies $\vdash n : \texttt{<}\nu\texttt{>}$

E-APP $e = e_1\ e_2$ — Lets assumes there exists τ such that $\vdash e : \tau$.
By inversion lemma (2), there exists τ' such that $\vdash e_1 : \tau \to \tau'$ and $\vdash e_2 : \tau$ Lets find the right type for each premises:

- $e_1 \rightsquigarrow \lambda x{:}\tau.\ e$ — Immediately, $\vdash e_1 : \tau \to \tau'$
- $e_2 \triangleright \tau \rightsquigarrow v$ — Immediately, $\vdash e_2 \triangleright \tau : \tau$
- $e\{v/x\} \rightsquigarrow v'$ — By hypothesis, $\vdash \lambda x{:}\tau.\ e : \tau \to \tau'$. The inversion lemma (3) gives $x{:}\forall \varepsilon.\ \tau \vdash e : \tau'$ allowing to conclude with substitution lemma: $\vdash e\{v/x\} : \tau'$

E-COERCE $e = e' \triangleright \tau'$ — By inversion lemma (7), there exists τ' such that $\vdash e : \tau''$ and $\tau' <: \tau$.

- $e \rightsquigarrow v$ — Immediately, $\vdash e : \tau''$
- $v \triangleright \tau' \rightsquigarrow v'$ — Immediately, $\vdash v \triangleright \tau' : \tau'$

E-LET $e = \texttt{let}\ x_{\boldsymbol{V}} : \tau' = e_1\ \texttt{in}\ e_2$ — Combined used of inversion lemma and substitution one.

□

Lemma A.8 ((S2) ∃-progress). *For any $e \notin \mathcal{V}$, if there exists τ such that $\vdash e : \tau$, then there exists a rule instance of the form $(j_1, \ldots, j_n, j_{n+1}, e)$*

Expression Inference $\Gamma \vdash e : \tau \dashv \mathcal{U}$

$$\text{I-Var}\ \frac{\Gamma(x) = \forall \boldsymbol{V}.\ \tau \qquad \boldsymbol{S} = Fresh(\boldsymbol{V}) \qquad \Gamma \vdash \boldsymbol{S} \dashv \mathcal{U}}{\Gamma \vdash {}^{l}x : \tau\{\boldsymbol{S}/\boldsymbol{V}\} \dashv \mathcal{U}\cdot\{l \mapsto \boldsymbol{S}\}} \qquad \text{I-Size}\ \frac{\Gamma \vdash \eta \dashv \mathcal{U}}{\Gamma \vdash \texttt{<}\eta\texttt{>} : \texttt{<}\eta\texttt{>} \dashv \mathcal{U}} \qquad \text{I-Int}\ \frac{}{\Gamma \vdash n : \texttt{int} \dashv \{\}}$$

$$\text{I-Abs}\ \frac{\Gamma \vdash \tau \dashv \mathcal{U}_\tau \qquad \Gamma,\ x{:}\forall \varepsilon.\ \tau \vdash e : \tau' \dashv \mathcal{U}}{\Gamma \vdash \lambda x{:}\tau.\ e : \tau \to \tau' \dashv \mathcal{U}_\tau\cdot\mathcal{U}} \qquad \text{I-App}\ \frac{\Gamma \vdash e' : \tau' \dashv \mathcal{U}' \qquad \Gamma \vdash (e : \tau' \to \alpha) \dashv \mathcal{U}}{\Gamma \vdash e\ e' : \alpha \dashv \mathcal{U}\cdot\mathcal{U}'}$$

$$\text{I-Coerce}\ \frac{\Gamma \vdash (e : \tau) \dashv \mathcal{U}}{\Gamma \vdash e \triangleright \tau : \tau \dashv \mathcal{U}\cdot\mathcal{U}_\tau} \qquad \text{I-Let}\ \frac{\Gamma \vdash d \Rightarrow x : \sigma \dashv \mathcal{U} \qquad \Gamma,\ x{:}\sigma \vdash e' : \tau' \dashv \mathcal{U}'}{\Gamma \vdash \texttt{let}\ d\ \texttt{in}\ e : \tau' \dashv \mathcal{U}\cdot\mathcal{U}'}$$

Declaration Inference $\Gamma \vdash d \Rightarrow x : \sigma \dashv \mathcal{U}$

$$\text{I-Decl}\ \frac{\Gamma \vdash (e : \tau) \dashv (V, C, \pi, \rho) \qquad \begin{matrix}\rho' = Solve(V, C)\\ V' = Gen(V, C, \rho')\end{matrix}}{\Gamma \vdash x_l{:}\tau = e \Rightarrow x : \forall \boldsymbol{V'}.\ \tau\{\rho'\} \dashv (\emptyset, C\{\rho'\}, \pi\cdot\{l \to V'\}, \rho' \circ \rho)}$$

Constraint Insertion $\Gamma \vdash (e : \tau) \dashv \mathcal{U}$

$$\text{I-Cstr}\ \frac{\Gamma \vdash e : \tau' \dashv \mathcal{U} \qquad \Gamma \vdash \tau \dashv \mathcal{U}_\tau}{\Gamma \vdash (e : \tau) \dashv \mathcal{U}\cdot\mathcal{U}_\tau\cdot\{\tau' <: \tau\}}$$

Figure A.1: *The inference algorithm. The function —*$\Gamma \vdash \boldsymbol{S} \dashv \underline{\mathcal{U}}$*— registers free size and type variables (the ones that are unbound in* Γ*) of instantiation list* $\boldsymbol{S}$*. The function —*$\Gamma \vdash (e : \tau) \dashv \underline{\mathcal{U}}$*— where expression and type are bracketed combines expression inference, type's free variables registering and sub-typing constraint insertion.*

Proof. Trivial case-based reasoning on expression shape. □

Lemma A.9 ((S3) ∀-progress)**.** *For any rule instance* $(e_1 \rightsquigarrow v_1, \ldots, e_n \rightsquigarrow v_n, e_{n+1} \rightsquigarrow v_{n+1}, e)$*, assuming there exists* τ *such that* $\vdash e : \tau$*, then, for any* $k \in [\![1, n+1]\!]$*,*

$$\begin{gathered}\textit{assuming for any } h < k,\ e_h \rightsquigarrow v_h \textit{ and } e_k \rightsquigarrow v,\\ \textit{then there exists a rule instance } (j'_1, \ldots, j'_{n'}, j_{n'+1}, e') \textit{ such that}\\ \forall h < k,\ j'_h = j_h,\ e' = e,\ \textit{et } j_k = e'' \rightsquigarrow v\end{gathered}$$

Intuitively, it amounts to checking that the evaluation of sub-expressions gives results that fulfill their use (the expected form of value).

Proof.

- E-App — By typing, (rule T-App), $\vdash e_1 : \tau_1 \to \tau_2$. Thus the canonical form lemma gives $e_1 \rightsquigarrow \lambda x{:}\tau.\ e$ The rule E-App can be instantiated. The continuation may be freely instantiated.
- Other rules do not constrain the result of evaluation of their sub-expressions, thus fulfilling the property.

□

Theorem A.10. *As proved in [Dag+20] the three properties allow to deduce Theorem 2.5:*

$$\begin{aligned}(S1) &\implies (\textit{Type preservation})\\ (S1) + (S2) + (S3) &\implies (\textit{Type soundness})\end{aligned}$$

1.2 Inference Properties

1.2.1 Algorithm

We formalize here the inference algorithm sketched in Section 2.3.

Constraint collecting Our algorithm builds a *unifier* $\mathcal{U} = (V, C, \pi, \rho)$ by traversing expressions bottom-up (from leaves that are variable occurrences, size values and constants to the top-level declaration). It collects (i) $V \subset \mathcal{V}_\eta \cup \mathcal{V}_\tau$ a set of free size and type variables ; (ii) C a set of sub-typing constraints ; (iii) π the definition of encountered polymorphism labels and (iv) ρ the substitution of some size and type variables, supposed acyclic. Substitutions' domains, free variables and variables that appear in a generalization list are supposed disjoint.

The empty unifier is $\{\}$. The union of unifiers is denoted $\mathcal{U}_1 \cdot \mathcal{U}_2$. It requires the substitutions' domains and free variable sets to be disjoint. This property holds during inference at the condition that size and type variables appear only in one place in traversed term. Singleton unifiers are unambiguously denoted $\{\alpha\}$ (a single free type variable), $\{\tau <: \tau'\}$ (a single sub-typing constraint),

$\{l \mapsto \boldsymbol{V}\}$ or $\{l \mapsto \boldsymbol{S}\}$ (a single polymorphism label definition). Last, $\mathcal{U} \setminus \boldsymbol{V}$ denotes the unifier $\mathcal{U}$ where the size and type variables $\boldsymbol{V}$ have been removed from its free variables.

Inference is made of a few mutually recursive functions defined in Figure A.1, that use environments as introduced in Section 2.2.3. To present them, their outputs are underlined. The main *expression inference* function $-\!\!-\Gamma \vdash e : \underline{\tau} \dashv \underline{\mathcal{U}}-\!\!-$ collects the constraints and builds expression's type. It is accompanied with a *registering* function $-\!\!-\Gamma \vdash \boldsymbol{S} \dashv \underline{\mathcal{U}}-\!\!-$ that returns a unifier containing the size and type variables of $\boldsymbol{S}$ that are unbounded in the environment[1]. For convenience, the *constraint insertion* function $-\!\!-\Gamma \vdash (e : \tau) \dashv \underline{\mathcal{U}}-\!\!-$, with bracketed expression and type combines expression inference and sub-typing constraint insertion, thus only producing a unifier. Last, the handling of declarations is set apart: the *declaration inference* function $-\!\!-\Gamma \vdash d \Rightarrow \underline{x} : \underline{\sigma} \dashv \underline{\mathcal{U}}-\!\!-$ builds the type scheme of introduced variables as well as declaration unifier.

Variables introduction Variable are immediately instantiated with fresh sizes and types (rule Var and auxiliary function *Fresh*). The definition of this instantiation label $-\!\!-l-\!\!-$ is registered in the unifier as well as the generated sizes and types. When handling abstractions (rule Abs), the free variables of the type τ are registered and the environment is extended with the monomorphic introduced variable. For applications (rule App), a fresh type variable α is picked. It allows constructing expression type without solving any type constraints, namely that the type of e in rule App should by a function, which is enforced by adding a constraint.

Polymorphism Let bindings introduce generalization (rule I-Decl): once expression has been traversed, function *Solve* turns as many unifier's sub-typing constraints as possible into a substitution of its free variables (see subsection 2.3.4). Function *Gen* extracts the free size and type variables, i.e., the ones that are not substituted, and checks that they do not appear in any remaining constraints so that they might be generalized. The resulting unifier is built by composing substitutions and registering generalization label's definition.

Reconstruction The reconstruction of top level terms $-\!\!-\Gamma \vdash e \rightarrowtail e'-\!\!-$ is defined with the unique rule

$$\text{I-Top}\ \frac{\Gamma \vdash e : \tau \dashv (\emptyset, \emptyset, \pi, \rho)}{\Gamma \vdash e \rightarrowtail e\{\pi\}\{\rho\}}$$

It requires all constraints to be solved, and no free variables to remain.[2] The resulting definition of polymorphism markers is applied $-\!\!-\{\pi\}-\!\!-$, then the substitution $-\!\!-\{\rho\}-\!\!-$. Note that the variables used in π for instantiation might get substituted, hence the order.

Definition A.11 (Reconstruction)**.** Given an environment Γ, expressions e and e', e' is a *reconstruction* of e, denoted $\Gamma \vdash e \leftarrowtail e'$. if and only if:

$$\exists \pi \rho \tau,\ \begin{cases} e' = e\{\pi\}\{\rho\} \\ \Gamma \vdash e' : \tau \end{cases}$$

Theorem A.12 (Inference soundness)**.** *Inference produces well-typed terms, i.e., given expressions e and e',*

$$\Gamma \vdash e \rightarrowtail e' \implies \Gamma \vdash e \leftarrowtail e'$$

Definition A.13. Relation & rule instance substitution Given a n-ary relation $\mathcal{R}(x_1, \ldots, x_n)$ and a substitution ρ, the substituted relation is defined as:

$$\mathcal{R}(x_1, \ldots, x_n)\{\rho\} \stackrel{def}{=} \mathcal{R}(x_1\{\rho\}, \ldots, x_n\{\rho\})$$

Similarly, given we define substituted rule instances as:

$$\text{Rule}\ \frac{p_1 \quad \cdots \quad p_k}{c}\ \{\rho\} \stackrel{def}{=} \text{Rule}\ \frac{p_1\{\rho\} \quad \cdots \quad p_k\{\rho\}}{c\{\rho\}}$$

[1] At this point, binding size or type variable is impossible in $\mathcal{L}^\eta$, but extensions (polymorphic recursion) will allow it.

[2] An extra declaration might be added to introduce a constraint solving point, i.e., `let` x_ε `:_ =` e `in` x

Proof. By construction, inference produces a type τ, polymorphism definitions π and a substitution ρ such that $e' = e\{\pi\}\{\rho\}$. It remains to show that $\Gamma \vdash e : \tau.$, by proving the following invariant: given an environment Γ an expression e, a type τ, size and type variables V, constraints C, polymorphism definitions π and a substitution ρ such that $\Gamma \vdash e : \tau \dashv (V, C, \pi, \rho)$, then

$$\forall \rho',\ \Gamma \vdash C\{\rho'\} \implies (\Gamma \vdash e\{\pi\}\{\rho\} : \tau)\,\{\rho'\}$$

Intuitively, this amounts to showing that any substitution that solves the remaining constraints leads to a well-type term, i.e., constraint collection captures all the necessary type relations.

$$\left.\begin{array}{l}\Gamma \vdash e : \tau \dashv (\boldsymbol{V}, C, \rho, \pi)\\ \vdash C\{\rho'\}.\end{array}\right\} \implies \mathrm{T}\ \frac{\cdots}{\Gamma \vdash e\{\pi\}\{\rho\} : \tau}\ \{\rho'\}$$

Because inference is syntax directed, we proceed by induction on expressions: given the resulting unifier and a substitution that solves the remaining constraints, we build a correct typing derivation for e:

- n — Inference has the following form:

$$\textsc{I-Int}\ \frac{}{\Gamma \vdash n : \mathtt{int} \dashv (\emptyset, \emptyset, \varepsilon, \varepsilon))}$$

  Given ρ' a substitution that solves $\emptyset$, the following typing derivation is correct, since $n\{\pi\}\{\rho\} = n$:

$$\begin{array}{r}\textsc{T-SubType}\end{array}\ \frac{\begin{array}{r}\textsc{T-ISize}\end{array}\ \frac{}{\Gamma \vdash n : \nu}\ \{\rho'\} \qquad \textsc{S-Size}\ \frac{}{\vdash \nu \leq \mathtt{int}}\ \{\rho'\}}{\Gamma \vdash n : \mathtt{int}}\ \{\rho'\}$$

- ${}^{l}x$ — Inference has the following form:

$$\textsc{I-Var}\ \frac{\Gamma(x) = \forall \boldsymbol{V}.\ \tau \qquad \boldsymbol{S} = \mathit{Fresh}(\boldsymbol{V})}{\Gamma \vdash {}^{l}x : \tau\,\{\boldsymbol{S}/\boldsymbol{V}\} \dashv (\boldsymbol{S}, \emptyset, \{l \mapsto \boldsymbol{S}\}, \varepsilon)}$$

  Given ρ' a substitution that solves $\emptyset$, the following typing derivation is correct, since ${}^{l}x\{\pi\}\{\rho\} = {}^{\boldsymbol{S}}x$:

$$\textsc{T-Var}\ \frac{\Gamma(x) = \forall \boldsymbol{V}.\ \tau}{\Gamma \vdash {}^{\boldsymbol{S}}x : \tau\,\{\boldsymbol{S}/\boldsymbol{V}\}}\ \{\rho'\}$$

- <η> — *idem*

- $e \triangleright \tau'$ — Inference has the following form:

$$\begin{array}{r}\textsc{I-Coerce}\end{array}\ \frac{\begin{array}{r}\textsc{I-Cstr}\end{array}\ \frac{\mathrm{I}\ \frac{\cdots}{\Gamma \vdash e_1 : \tau' \dashv (V, C, \pi, \rho)} \qquad \frac{}{\Gamma \vdash \tau' \dashv (V_\tau, \emptyset, \varepsilon, \varepsilon)}}{\Gamma \vdash (e : \tau') \dashv (V \cdot \mathcal{V}_\tau, C \cdot \{\tau' <: \tau\}, \pi, \rho)}}{\Gamma \vdash e \triangleright \tau' : \tau \dashv (V \cdot \mathcal{V}_\tau, C \cdot \{\tau' <: \tau\}, \pi, \rho)}$$

  Given ρ' a substitution that solves $C \cdot \{\tau' <: \tau\}$ then $\vdash C\{\rho'\}$, the induction hypothesis allows to construct T a typing derivation of $e\{\pi\}\{\rho\}$. Moreover, because $\tau'\{\rho'\} <: \tau\{\rho'\}$, it exists a sub-typing derivation S such that the following derivation is valid:

$$\begin{array}{r}\textsc{T-SubType}\end{array}\ \frac{\begin{array}{r}\textsc{T-Coerce}\end{array}\ \frac{\mathrm{T}\ \frac{\cdots}{\Gamma \vdash e\{\pi\}\{\rho\} : \tau'}\ \{\rho'\}}{\Gamma \vdash e\{\pi\}\{\rho\} \triangleright \tau' : \tau'}\ \{\rho'\} \qquad \mathrm{S}\ \frac{\cdots}{\tau' <: \tau}\ \{\rho'\}}{\Gamma \vdash e\{\pi\}\{\rho\} \triangleright \tau' : \tau}\ \{\rho'\}$$

- $e_1\ e_2$ — Inference has the following form:

$$\begin{array}{r}\textsc{I-App}\end{array}\ \frac{\begin{array}{r}\textsc{I-2}\end{array}\ \frac{\cdots}{\Gamma \vdash e_2 : \tau_2 \dashv (V_2, C_2, \pi_2, \rho_2)} \qquad \begin{array}{r}\textsc{I-Cstr}\end{array}\ \frac{\begin{array}{r}\textsc{I-1}\end{array}\ \frac{\cdots}{\Gamma \vdash e_1 : \tau_1 \dashv (V_1, C_1, \pi_1, \rho_1)} \qquad \frac{}{\Gamma \vdash (\tau_2 \to \alpha) \dashv (\{\alpha\}, \emptyset, \varepsilon, \varepsilon)}}{\Gamma \vdash (e_1 : \tau_2 \to \alpha) \dashv (V_1 \cdot \{\alpha\}, C_1 \cdot \{\tau_1 <: \tau_2 \to \alpha\}, \pi_1, \rho_1)}}{\Gamma \vdash e_1\ e_2 : \alpha \dashv (V_1 \cdot V_2 \cdot \{\alpha\}, C_1 \cdot C_2 \cdot \{\tau_1 <: \tau_2 \to \alpha\}, \pi_1 \cdot \pi_2, \rho_1 \cdot \rho_2)}$$

  Given ρ' a substitution that solves C_1, C_2 and $\{\tau_1 <: \tau_2 \to \alpha\}$, using induction hypothesis on I-1 and I-2 defining T-1 and `T-2` as above and a sub-typing derivation for $\{\tau_1 <: \tau_2 \to \alpha\}$, the following typing derivation of $e\{\pi\}\{\rho'\} = e_1\{\pi_1\}\{\rho_1\}\ e_2\{\pi_2\}\{\rho_2\}$ is correct:

$$\begin{array}{r}\textsc{T-App}\end{array}\ \frac{\begin{array}{r}\textsc{T-SubType}\end{array}\ \frac{\begin{array}{r}\textsc{T-1}\end{array}\ \frac{\cdots}{\Gamma \vdash e_1\{\pi_1\}\{\rho_1\} : \tau_1}\ \{\rho'\} \qquad \mathrm{S}\ \frac{\cdots}{\tau_1 <: \tau_2 \to \alpha}\ \{\rho'\}}{\Gamma \vdash e_1\{\pi_1\}\{\rho_1\} : \tau_2 \to \alpha}\ \{\rho'\} \qquad \begin{array}{r}\textsc{T-2}\end{array}\ \frac{\cdots}{\Gamma \vdash e_2\{\pi_2\}\{\rho_2\} : \tau_2}\ \{\rho'\}}{\Gamma \vdash e_1\{\pi_1\}\{\rho_1\}\ e_2\{\pi_2\}\{\rho_2\} : \alpha}\ \{\rho'\}$$

- $\lambda x{:}\tau.\ e'$ — Inference has the following form:

$$\text{I-Abs}\ \frac{\overline{\Gamma \vdash \tau \dashv (V_\tau, \emptyset, \varepsilon, \varepsilon)} \qquad \text{I}\ \frac{\cdots}{\Gamma,\ x{:}\forall\varepsilon.\ \tau \vdash e' : \tau' \dashv (V, C, \pi, \rho)}}{\Gamma \vdash \lambda x{:}\tau.\ e' : \tau \to \tau' \dashv (V_\tau \cdot V, C, \pi, \rho)}$$

Given ρ' a substitution that solves C, using induction hypothesis on I (defining T), the following typing derivation of $e\{\pi\}\{\rho\}$ is correct since τ variables are registered in collected free variables, hence get defined by ρ':

$$\text{T-Abs}\ \frac{\text{T}\ \frac{\cdots}{\Gamma,\ x{:}\forall\varepsilon.\ \tau \vdash e'\{\pi_1\}\{\rho_1\} : \tau'}\ \{\rho'\}}{\Gamma \vdash \lambda x{:}\tau.\ e\{\pi_1\}\{\rho_1\}' : \tau \to \tau'}\ \{\rho'\}$$

- $\texttt{let}\ x_{\boldsymbol{V}} {:} \tau' = e_1\ \texttt{in}\ e_2$ — Inference has the following form:

$$\text{I-Decl}\ \frac{\text{I-Cstr}\ \frac{\text{I-1}\ \frac{\cdots}{\Gamma \vdash e_1 : \tau \dashv (V_1, C_1, \pi_1, \rho_1)} \qquad \overline{\Gamma \vdash \tau_1 \dashv (V_\tau, \emptyset, \varepsilon, \varepsilon)}}{\Gamma \vdash (e_1 : \tau_1) \dashv (V_1 \cdot V_\tau, C_1 \cdot \{\tau <: \tau_1\}, \pi_1, \rho_1)} \qquad V, \rho = Solve(V_1 \cdot V_\tau, C_1 \cdot \{\tau <: \tau_1\})}{\Gamma \vdash x_l : \tau_1 = e_1 \Rightarrow x : \forall \boldsymbol{V}.\ \tau_1\{\rho\} \dashv (\emptyset, C_1\{\rho\} \cdot \{\tau\{\rho\} <: \tau_1\{\rho\}\}, \pi_1 \cdot \{l \to V\}, \rho \circ \rho_1)}$$

$$\text{I-Let}\ \frac{\text{I-2}\ \frac{\cdots}{\Gamma,\ x{:}\forall \boldsymbol{V}.\ \tau_1\{\rho\} \vdash e_2 : \tau_2 \dashv (V_2, C_2, \pi_2, \rho_2)}}{\Gamma \vdash \texttt{let}\ x_l : \tau_1 = e_1\ \texttt{in}\ e_2 : \tau_2 \dashv (V_2, C_1\{\rho\} \cdot \{\tau\{\rho\} <: \tau_1\{\rho\}\} \cdot C_2, \pi_1 \cdot \{l \to V\} \cdot \pi_2, \rho \circ \rho_1 \cdot \rho_2)}$$

Here, Solve combines the Solve and Gen function of the algorithms. It defines a substitution ρ of variables $V_1 \cdot V_\tau \setminus V$. The constraint set C_1 is substituted and transmitted in the resulting unifier Thus, given ρ' a substitution that solves C, $\rho' \circ \rho$ solves the constraints $C_1 \cdot \{\tau <: \tau_1\}$. The reconstructed term has the following shape: $\texttt{let}\ x_V : \tau_1\{\rho \circ \rho_1\} = e_1\{\pi_1\}\{\rho \circ \rho_1\}\ \texttt{in}\ e_2\{\pi_2\}\{\rho_2\}$. Its type can be derived with:

$$\text{T-SubType}\ \frac{\text{T-1}\ \frac{\cdots}{\Gamma,\ V \vdash e_1\{\pi_1\}\{\rho_1\} : \tau\{\rho_1\}}\ \{\rho' \circ \rho\} \qquad \text{S}\ \frac{\cdots}{\tau\{\rho_1\} <: \tau_1\{\rho_1\}}\ \{\rho' \circ \rho\}}{\Gamma,\ V \vdash e_1\{\pi_1\}\{\rho_1\} : \tau_1\{\rho_1\}}\ \{\rho' \circ \rho\}$$

$$\text{T-Let}\ \frac{\text{T-2}\ \frac{\cdots}{\Gamma,\ x{:}\forall V.\ \tau \vdash e_2\{\pi_2\}\{\rho_2\} : \tau_2\{\rho_2\}}\ \{\rho'\}}{\Gamma \vdash \texttt{let}\ x_V : \tau_1\{\rho \circ \rho_1\} = e_1\{\pi_1\}\{\rho \circ \rho_1\}\ \texttt{in}\ e_2\{\pi_2\}\{\rho_2\} : \tau_2}\ \{\rho'\}$$

Once established, this invariant allows an easy deriving of inference soundness: for top-level terms, the constraint set must be empty, hence choosing the empty substitution yields a valid type derivation.

□

Conjecture A.14 (Inference non-specialization)**.** *Given an expression e and two reconstructed terms e_1, e_2 where e_1 is the result of the inference, then they have equivalent observable semantics. Formally:*

$$\begin{cases} \Gamma \vdash e \rightarrowtail e_1 \\ \Gamma \vdash e \rightarrowtail e_2 \end{cases} \implies e_1 \equiv e_2$$

This property states that inference rejects any terms that might be given multiple semantics, hence that the implicitly typed language is deterministic.

The proof has not been fully conducted yet. The main difficulty lies in the handling of diverging environments: let bindings introduce variables that can be given multiple type schemes. At instantiation places, these variables induce different types leading to different constraint sets. We must then prove that the size constraints extracted by the inference are indeed fulfilled by the arbitrary reconstruction. This would allow to deduce that the substitution built by the inference is builds a *most general size unifier*.

Let define a characterization of type schemes built by the inference.

Definition A.15 (Subsumption)**.** Given two type schemes $\sigma_1 := \forall \boldsymbol{V}_1.\ \tau_1$ and $\sigma_2 := \forall \boldsymbol{V}_2.\ \tau_2$, the *subsumption* relation — $\sigma_1 \preccurlyeq \sigma_2$ — holds if and only if any instance of the second is a sub-type of an instance of the first. Formally, one of the two equivalent formulation must hold:

$$\forall \boldsymbol{S}_2.\ \exists \boldsymbol{S}_1.\ \tau_1\{\boldsymbol{S}_1/\boldsymbol{V}_1\} <: \tau_2\{\boldsymbol{S}_2/\boldsymbol{V}_2\}$$

$$\exists \boldsymbol{S}.\ \tau_1\{\boldsymbol{S}/\boldsymbol{V}_1\} <: \tau_2 \quad (\text{where } \mathcal{FV}(\ \boldsymbol{S}\ ) \in \boldsymbol{V}_2)$$

Because our type systems does not have principal types, the types produced by the inference cannot be the most polymorphic one, i.e., such that is subsume to any other valid type. This property must be refined for sizes:

Definition A.16 (Size subsumption)**.** Given two type schemes σ_1 and σ_2, the *size subsumption* relation — $\sigma_1 \preccurlyeq_\eta \sigma_2$ — holds if and only if:

$$\exists \sigma\ \vec{\alpha}\ \vec{\tau}.\ \begin{cases} \sigma_1 \preccurlyeq \sigma\{\overrightarrow{\texttt{int}}/\vec{\alpha}\} \\ \sigma\{\vec{\tau}/\vec{\alpha}\} \preccurlyeq \sigma_2 \\ \vec{\tau} <: \overrightarrow{\texttt{int}} \end{cases}$$

The intuition is: if $\sigma_1 \preccurlyeq_\eta \sigma_2$, the second type scheme is obtained by adding some refinements (in both positive and negative positions) in place of some `int` in σ_1. In a term, this would guarantee that the semantics is independent of those extra refinements (since a semantics exists without). Thus showing that inference builds such minimal term for the size subsumption relation would help establishing our conjecture.

Bibliography

[Mea55] George H Mealy. “A method for synthesizing sequential circuits”. In: *The Bell System Technical Journal* 34.5 (1955), pp. 1045–1079 (cit. on p. 7).

[Ive62] Kenneth E. Iverson. “A programming language”. In: *Proceedings of the 1962 spring joint computer conference, AFIPS 1962 (Spring), San Francisco, California, USA, May 1-3, 1962*. Ed. by G. A. Barnard III. ACM, 1962, pp. 345–351. DOI: 10.1145/1460833.1460872 (cit. on pp. 11, 12).

[TE68] Lawrence G. Tesler and Horace J. Enea. “A language design for concurrent processes”. In: *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference, Atlantic City, NJ, USA, 30 April - 2 May 1968*. Vol. 32. AFIPS Conference Proceedings. Thomson Book Company, Washington D.C., 1968, pp. 403–408. DOI: 10.1145/1468075.1468134 (cit. on pp. xviii, 11).

[Hin69] Roger Hindley. “The principal type-scheme of an object in combinatory logic”. In: *Transactions of the american mathematical society* 146 (1969), pp. 29–60 (cit. on pp. 13, 43, 44).

[Bru70] N.G. de Bruijn. “The mathematical language AUTOMATH, its usage and some of its extensions”. In: *Proceedings Symposium on Automatic Demonstration, Versailles, France, December 1968*. Ed. by M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger. Vol. 125. Lecture Notes in Mathematics. Springer, 1970, pp. 29–61. DOI: 10.1007/BFb0060623 (cit. on p. 14).

[Kah74] Gilles Kahn. “The Semantics of a Simple Language for Parallel Programming”. In: *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*. Ed. by Jack L. Rosenfeld. North-Holland, 1974, pp. 471–475 (cit. on pp. 4, 5).

[Lam74] Leslie Lamport. “The Hyperplane Method for an Array Computer”. In: *Parallel Processing, Proceedings of the Sagamore Computer Conference, Sagamore, Adirondack Mountains, NY, USA, August 20-23, 1974*. Ed. by Tse-Yun Feng. Vol. 24. Lecture Notes in Computer Science. Springer, 1974, pp. 113–131. DOI: 10.1007/3-540-07135-0_114 (cit. on p. 20).

[Sco76] Dana S. Scott. “Data Types as Lattices”. In: *SIAM Journal on Computing* 5.3 (1976), pp. 522–587. DOI: 10.1137/0205037 (cit. on p. 5).

[AW77] Edward A. Ashcroft and William W. Wadge. “Lucid, a Nonprocedural Language with Iteration”. In: *Commun. ACM* 20.7 (1977), pp. 519–526. DOI: 10.1145/359636.359715 (cit. on pp. xviii, 4, 175).

[Bac78] John W. Backus. “Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs”. In: *Communications of the ACM* 21.8 (1978), pp. 613–641. DOI: 10.1145/359576.359579 (cit. on p. 16).

[Mil78] Robin Milner. “A Theory of Type Polymorphism in Programming”. In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4 (cit. on pp. 13, 37, 45, 47).

[Con83] Robert L. Constable. “Mathematics as Programming”. In: *Logics of Programs, Workshop, Carnegie Mellon University, Pittsburgh, PA, USA, June 6-8, 1983, Proceedings*. Ed. by Edmund M. Clarke and Dexter Kozen. Vol. 164. Lecture Notes in Computer

Science. Springer, 1983, pp. 116–128. DOI: 10.1007/3-540-12896-4_359 (cit. on p. 14).

[Mee83] Lambert G. L. T. Meertens. "Incremental Polymorphic Type Checking in B". In: *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983*. Ed. by John R. Wright, Larry Landweber, Alan J. Demers, and Tim Teitelbaum. ACM Press, 1983, pp. 265–275. DOI: 10.1145/567067.567092 (cit. on p. 48).

[BC84] Gérard Berry and Laurent Cosserat. "The ESTEREL Synchronous Programming Language and its Mathematical Semantics". In: *Seminar on Concurrency, Carnegie-Mellon University, Pittsburg, PA, USA, July 9-11, 1984*. Ed. by Stephen D. Brookes, A. W. Roscoe, and Glynn Winskel. Vol. 197. Lecture Notes in Computer Science. Springer, 1984, pp. 389–448. DOI: 10.1007/3-540-15670-4_19 (cit. on p. 4).

[HP84] David Harel and Amir Pnueli. "On the Development of Reactive Systems". In: *Logics and Models of Concurrent Systems - Conference proceedings, Colle-sur-Loup (near Nice), France, 8-19 October 1984*. Ed. by Krzysztof R. Apt. Vol. 13. NATO ASI Series. Springer, 1984, pp. 477–498. DOI: 10.1007/978-3-642-82453-1_17 (cit. on p. 1).

[Mar84] Per Martin-Löf. *Intuitionistic type theory*. Vol. 1. Studies in proof theory. Bibliopolis, 1984. ISBN: 978-88-7088-228-5 (cit. on p. 14).

[Myc84] Alan Mycroft. "Polymorphic Type Schemes and Recursive Definitions". In: *International Symposium on Programming, 6th Colloquium, Toulouse, France, April 17-19, 1984, Proceedings*. Ed. by Manfred Paul and Bernard J. Robinet. Vol. 167. Lecture Notes in Computer Science. Springer, 1984, pp. 217–228. DOI: 10.1007/3-540-12925-1_41 (cit. on p. 48).

[She85] Mary Sheeran. "Designing Regular Array Architectures using Higher Order Functions". In: *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*. Ed. by Jean-Pierre Jouannaud. Vol. 201. Lecture Notes in Computer Science. Springer, 1985, pp. 220–237. DOI: 10.1007/3-540-15975-4_39 (cit. on pp. 13, 17).

[WA+85] William W Wadge, Edward A Ashcroft, et al. *Lucid, the dataflow programming language*. Vol. 303. Academic Press London, 1985 (cit. on pp. 11, 100, 131).

[Mol86] Cleve Moler. "Matrix computation on distributed memory multiprocessors". In: *Hypercube Multiprocessors* 86.181-195 (1986), p. 31 (cit. on p. 16).

[Cas+87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. "Lustre: A Declarative Language for Programming Synchronous Systems". In: *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*. ACM Press, 1987, pp. 178–188. DOI: 10.1145/41625.41641 (cit. on p. 7).

[Gir87] Jean-Yves Girard. "Linear Logic". In: *Theoretical computer science* 50 (1987), pp. 1–102. DOI: 10.1016/0304-3975(87)90045-4 (cit. on p. 19).

[HHP87] Robert Harper, Furio Honsell, and Gordon D. Plotkin. "A Framework for Defining Logics". In: *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*. IEEE Computer Society, 1987, pp. 194–204 (cit. on p. 14).

[BW88] Richard S. Bird and Philip Wadler. *Introduction to functional programming*. Prentice Hall International series in computer science. Prentice Hall, 1988. ISBN: 978-0-13-484197-7 (cit. on p. 11).

[CH88] Thierry Coquand and Gérard P. Huet. "The Calculus of Constructions". In: *Information and Computation* 76.2/3 (1988), pp. 95–120. DOI: 10.1016/0890-5401(88)90005-3 (cit. on p. 14).

[MP88] John C. Mitchell and Gordon D. Plotkin. "Abstract Types Have Existential Type". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 10.3 (1988), pp. 470–502. DOI: 10.1145/44501.45065 (cit. on p. 15).

[NN88] Hanne Riis Nielson and Flemming Nielson. "Automatic Binding Time Analysis for a Typed lambda-Calculus". In: *Science of computer programming* 10.1 (1988), pp. 139–176. DOI: 10.1016/0167-6423(88)90025-1 (cit. on pp. 13, 49).

[Ber89] Gérard Berry. "Real Time Programming: Special Purpose or General Purpose Languages". In: *Information Processing 89, Proceedings of the IFIP 11th World Computer Congress, San Francisco, USA, August 28 - September 1, 1989*. Ed. by Gerhard X. Ritter. North-Holland/IFIP, 1989, pp. 11–17 (cit. on p. 3).

[FCO90] John Feo, David C. Cann, and R. R. Oldehoeft. "A Report on the Sisal Language Project". In: *J. Parallel Distributed Comput.* 10.4 (1990), pp. 349–366. DOI: 10.1016/0743-7315(90)90035-N (cit. on pp. xviii, 11, 12, 27, 30, 37, 109, 126, 131, 175).

[Wad90] Philip Wadler. "Linear Types can Change the World!" In: *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*. Ed. by Manfred Broy and Cliff B. Jones. North-Holland, 1990, p. 561 (cit. on p. 19).

[Bak91] Henry G. Baker. "Shallow binding makes functional arrays fast". In: *ACM SIGPLAN Notices* 26.8 (1991), pp. 145–147. DOI: 10.1145/122598.122614 (cit. on p. 18).

[BB91] Albert Benveniste and Gérard Berry. "The synchronous approach to reactive and real-time systems". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1270–1282. DOI: 10.1109/5.97297 (cit. on p. 3).

[BLJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. "Synchronous Programming with Events and Relations: the SIGNAL Language and Its Semantics". In: *Science of computer programming* 16.2 (1991), pp. 103–149. DOI: 10.1016/0167-6423(91)90001-E (cit. on p. 4).

[Dem91] James Demmel. "LAPACK: A portable linear algebra library for high-performance computers". In: *Concurrency - Practice and Experience* 3.6 (1991), pp. 655–666. DOI: 10.1002/cpe.4330030610 (cit. on p. 20).

[Fea91] Paul Feautrier. "Dataflow analysis of array and scalar references". In: *International Journal of Parallel Programming* 20.1 (1991), pp. 23–53. DOI: 10.1007/BF01407931 (cit. on p. 21).

[FP91] Timothy S. Freeman and Frank Pfenning. "Refinement Types for ML". In: *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*. Ed. by David S. Wise. ACM, 1991, pp. 268–277. DOI: 10.1145/113445.113468 (cit. on p. 14).

[Hal+91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. "The synchronous data flow programming language LUSTRE". In: *Proc. IEEE* 79.9 (1991), pp. 1305–1320. DOI: 10.1109/5.97300 (cit. on pp. vii, 4, 99).

[HRR91] Nicolas Halbwachs, Pascal Raymond, and Christophe Ratel. "Generating Efficient Code From Data-Flow Programs". In: *Programming Language Implementation and Logic Programming, 3rd International Symposium, PLILP'91, Passau, Germany, August 26-28, 1991, Proceedings*. Ed. by Jan Maluszynski and Martin Wirsing. Vol. 528. Lecture Notes in Computer Science. Springer, 1991, pp. 207–218. DOI: 10.1007/3-540-54444-5_100 (cit. on p. 7).

[Pug91] William W. Pugh. "Uniform techniques for loop optimization". In: *Proceedings of the 5th international conference on Supercomputing, ICS 1991, Cologne, Germany, June 17-21, 1991*. Ed. by Edward S. Davidson and Friedel Hossfeld. ACM, 1991, pp. 341–352. DOI: 10.1145/109025.109108 (cit. on p. 21).

[BG92] Gérard Berry and Georges Gonthier. "The Esterel Synchronous Programming Language: Design, Semantics, Implementation". In: *Science of computer programming* 19.2 (1992), pp. 87–152. DOI: 10.1016/0167-6423(92)90005-V (cit. on p. 7).

[OW92] Mehmet A. Orgun and William W. Wadge. "Towards a Unified Theory of Intensional Logic Programming". In: *The Journal of Logic Programming* 13.4 (1992), pp. 413–440. DOI: 10.1016/0743-1066(92)90055-8 (cit. on p. 11).

[AW93] Alexander Aiken and Edward L. Wimmers. “Type Inclusion Constraints and Type Inference”. In: *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*. Ed. by John Williams. ACM, 1993, pp. 31–41. DOI: 10.1145/165180.165188 (cit. on pp. xi, 45, 46).

[Fit93] Steven Fitzgerald. “Copy elimination for true multidimensional arrays in SISAL 2.0”. In: *Proceedings SISAL* 93 (1993), pp. 59–73 (cit. on p. 19).

[Hen93] Fritz Henglein. “Type Inference with Polymorphic Recursion”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15.2 (1993), pp. 253–289. DOI: 10.1145/169701.169692 (cit. on p. 49).

[Wil93] Doran Wilde. *A library for doing polyhedral operations*. Tech. rep. RR-2157. INRIA Research Report, 1993 (cit. on p. 21).

[JC94] C. Barry Jay and J. Robin B. Cockett. “Shapely Types and Shape Polymorphism”. In: *Programming Languages and Systems - ESOP’94, 5th European Symposium on Programming, Edinburgh, UK, April 11-13, 1994, Proceedings*. Ed. by Donald Sannella. Vol. 788. Lecture Notes in Computer Science. Springer, 1994, pp. 302–316. DOI: 10.1007/3-540-57880-3_20 (cit. on p. 13).

[Ken94] Andrew Kennedy. “Dimension Types”. In: *Programming Languages and Systems - ESOP’94, 5th European Symposium on Programming, Edinburgh, UK, April 11-13, 1994, Proceedings*. Ed. by Donald Sannella. Vol. 788. Lecture Notes in Computer Science. Springer, 1994, pp. 348–362. DOI: 10.1007/3-540-57880-3_23 (cit. on pp. viii, 14, 16).

[QRW94] Patrice Quinton, Sanjay Rajopadhye, and Doran Wilde. “Using static analysis to derive imperative code from ALPHA”. PhD thesis. INRIA, 1994 (cit. on p. 21).

[Kar95] Anatolii Alexeevich Karatsuba. “The complexity of computations”. In: *Proceedings of the Steklov Institute of Mathematics-Interperiodica Translation* 211 (1995), pp. 169–183 (cit. on p. 20).

[Wad95] Philip Wadler. “Monads for Functional Programming”. In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*. Ed. by Johan Jeuring and Erik Meijer. Vol. 925. Lecture Notes in Computer Science. Springer, 1995, pp. 24–52. DOI: 10.1007/3-540-59451-5_2 (cit. on p. 19).

[And96a] Charles André. “Representation and analysis of reactive behaviors: A synchronous approach”. In: *Computational Engineering in Systems Applications, CESA*. Vol. 96. 1996, pp. 19–29 (cit. on pp. 4, 9).

[And96b] Charles André. “SyncCharts: A visual representation of reactive behaviors”. In: *I3S, Sophia-Antipolis, France, Tech. Rep. RR* (1996), pp. 95–52 (cit. on p. 4).

[CP96] Paul Caspi and Marc Pouzet. “Synchronous Kahn Networks”. In: *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*. Ed. by Robert Harper and Richard L. Wexelblat. ACM, 1996, pp. 226–238. DOI: 10.1145/232627.232651 (cit. on pp. 4, 8, 9, 28, 100, 101).

[Hal+96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. “Type Classes in Haskell”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18.2 (1996), pp. 109–138. DOI: 10.1145/227699.227700 (cit. on p. 39).

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. “Proving the Correctness of Reactive Systems Using Sized Types”. In: *Conference Record of POPL’96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*. Ed. by Hans-Juergen Boehm and Guy L. Steele Jr. ACM Press, 1996, pp. 410–423. DOI: 10.1145/237721.240882 (cit. on pp. 14, 16).

[Gau+97] J-L Gaudiot, Wim Bohm, Walid Najjar, Tom DeBoni, John Feo, and Patrick Miller. "The Sisal model of functional programming and its implementation". In: *Parallel Algorithms/Architecture Synthesis*. Proceedings. Second Aizu International Symposium. IEEE, 1997, pp. 112–123 (cit. on pp. 19, 75).

[GdM97] Jean-Louis Giavitto, Dominique de Vito, and Olivier Michel. "Semantics and Compilation of Recursive Sequential Streams in 8½". In: *Programming Languages: Implementations, Logics, and Programs, 9th International Symposium, PLILP'97, Including a Special Trach on Declarative Programming Languages in Education, Southampton, UK, September 3-5, 1997, Proceedings*. Ed. by Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen. Vol. 1292. Lecture Notes in Computer Science. Springer, 1997, pp. 207–223. DOI: 10.1007/BFb0033846 (cit. on p. 7).

[JS97] C Barry Jay and Milan Sekanina. "Shape checking of array programs". In: *Computing: the Australasian Theory Seminar, Proceedings*. Vol. 19. Australian Computer Science Communications. University of Technology, Sydney, Australia, 1997, pp. 113–121 (cit. on pp. 13, 37, 49).

[Jon97] Mark P. Jones. "First-class Polymorphism with Type Inference". In: *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*. Ed. by Peter Lee, Fritz Henglein, and Neil D. Jones. ACM Press, 1997, pp. 483–496. DOI: 10.1145/263699.263765 (cit. on p. 15).

[Zen97] Christoph Zenger. "Indexed Types". In: *Theoretical computer science* 187.1-2 (1997), pp. 147–165. DOI: 10.1016/S0304-3975(97)00062-5 (cit. on pp. viii, 14, 16, 27).

[Bje+98] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. "Lava: Hardware Design in Haskell". In: *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*. Ed. by Matthias Felleisen, Paul Hudak, and Christian Queinnec. ACM, 1998, pp. 174–184. DOI: 10.1145/289423.289440 (cit. on pp. 13, 22).

[CP98] Paul Caspi and Marc Pouzet. "A Co-iterative Characterization of Synchronous Stream Functions". In: *First Workshop on Coalgebraic Methods in Computer Science, CMCS 1998, Lisbon, Portugal, March 28-29, 1998*. Ed. by Bart Jacobs, Larry Moss, Horst Reichel, and Jan J. M. M. Rutten. Vol. 11. Electronic Notes in Theoretical Computer Science. Elsevier, 1998, pp. 1–21. DOI: 10.1016/S1571-0661(04)00050-7 (cit. on p. 5).

[Jay98] C Barry Jay. *The FISh language definition*. Tech. rep. 1998 (cit. on p. 13).

[RW98] Panos Rondogiannis and William W Wadge. "Intensional programming languages". In: *Proceedings of the First Panhellenic Conference on New Information Technologies (NIT'98), Athens, Greece*. Citeseer. 1998, pp. 85–94 (cit. on p. 11).

[ROS98] John M. Rushby, Sam Owre, and Natarajan Shankar. "Subtypes for Specifications: Predicate Subtyping in PVS". In: *IEEE Trans. Software Eng.* 24.9 (1998), pp. 709–720. DOI: 10.1109/32.713327 (cit. on p. 14).

[XP98] Hongwei Xi and Frank Pfenning. "Eliminating Array Bound Checking Through Dependent Types". In: *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*. Ed. by Jack W. Davidson, Keith D. Cooper, and A. Michael Berman. ACM, 1998, pp. 249–257. DOI: 10.1145/277650.277732 (cit. on pp. 15, 16, 47).

[Bie+99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. "Symbolic Model Checking without BDDs". In: *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*. Ed. by Rance Cleaveland. Vol. 1579. Lecture Notes in Computer Science. Springer, 1999, pp. 193–207. DOI: 10.1007/3-540-49059-0_14 (cit. on p. 5).

[LM99] Daan Leijen and Erik Meijer. "Domain specific embedded compilers". In: *Proceedings of the Second Conference on Domain-Specific Languages (DSL '99), Austin, Texas, USA, October 3-5, 1999*. Ed. by Thomas Ball. ACM, 1999, pp. 109–122. DOI: 10.1145/331960.331977 (cit. on p. 14).

[XP99] Hongwei Xi and Frank Pfenning. "Dependent Types in Practical Programming". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. Ed. by Andrew W. Appel and Alex Aiken. ACM, 1999, pp. 214–227. DOI: 10.1145/292540.292560 (cit. on pp. viii, 15, 37, 40, 44, 173).

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. "Checking Safety Properties Using Induction and a SAT-Solver". In: *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*. Ed. by Warren A. Hunt Jr. and Steven D. Johnson. Vol. 1954. Lecture Notes in Computer Science. Springer, 2000, pp. 108–125. DOI: 10.1007/3-540-40922-X_8 (cit. on p. 5).

[CP01] Pascal Cuoq and Marc Pouzet. "Modular Causality in a Synchronous Stream Language". In: *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*. Ed. by David Sands. Vol. 2028. Lecture Notes in Computer Science. Springer, 2001, pp. 237–251. DOI: 10.1007/3-540-45309-1_16 (cit. on pp. 87, 172).

[Pot01] François Pottier. "Simplifying Subtyping Constraints: A Theory". In: *Information and computation* 170.2 (2001), pp. 153–183. DOI: 10.1006/inco.2001.2963 (cit. on p. 46).

[CP02] Jean-Louis Colaço and Marc Pouzet. "Type-based Initialization Analysis of a Synchronous Data-flow Language". In: *Synchronous Languages, Applications, and Programming*. Vol. 65. Electronic Notes in Theoretical Computer Science, 2002 (cit. on p. 99).

[Ham02] Grégoire Hamon. "Calcul d'horloges et structures de contrôle dans Lucid Synchrone, un langage de flots synchrones à la ML". PhD thesis. Paris 6, 2002 (cit. on p. 82).

[AAG03] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. "Categories of Containers". In: *Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*. Ed. by Andrew D. Gordon. Vol. 2620. Lecture Notes in Computer Science. Springer, 2003, pp. 23–38. DOI: 10.1007/3-540-36576-1_2 (cit. on p. 61).

[Bas+03] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. "Putting Polyhedral Loop Transformations to Work". In: *Languages and Compilers for Parallel Computing, 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003, Revised Papers*. Ed. by Lawrence Rauchwerger. Vol. 2958. Lecture Notes in Computer Science. Springer, 2003, pp. 209–225. DOI: 10.1007/978-3-540-24644-2_14 (cit. on p. 21).

[EL03] Stephen A. Edwards and Edward A. Lee. "The semantics and execution of a synchronous block-diagram language". In: *Science of Computer Programming* 48.1 (2003), pp. 21–42. DOI: 10.1016/S0167-6423(02)00096-5 (cit. on p. 5).

[Sch03] Sven-Bodo Scholz. "Single Assignment C: efficient support for high-level array operations in a functional setting". In: *Journal of functional programming* 13.6 (2003), pp. 1005–1059. DOI: 10.1017/S0956796802004458 (cit. on p. 15).

[Bas04] Cédric Bastoul. "Code Generation in the Polyhedral Model Is Easier Than You Think". In: *13th International Conference on Parallel Architectures and Compilation Techniques (PACT 2004), 29 September - 3 October 2004, Antibes Juan-les-Pins, France*. IEEE Computer Society, 2004, pp. 7–16. DOI: 10.1109/PACT.2004.10018 (cit. on p. 21).

[Col+04] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. "Towards a higher-order synchronous data-flow language". In: *EMSOFT 2004, September 27-29, 2004, Pisa, Italy, Fourth ACM International Conference On Embedded Software, Proceedings*. Ed. by Giorgio C. Buttazzo. ACM, 2004, pp. 230–239. DOI: 10.1145/1017753.1017792 (cit. on p. 9).

[Hin04] Ralf Hinze. "An Algebra of Scans". In: *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings*. Ed. by Dexter Kozen and Carron Shankland. Vol. 3125. Lecture Notes in Computer Science. Springer, 2004, pp. 186–210. DOI: 10.1007/978-3-540-27764-4_11 (cit. on p. 18).

[MM04] Florence Maraninchi and Lionel Morel. "Arrays and Contracts for the Specification and Analysis of Regular Systems". In: *4th International Conference on Application of Concurrency to System Design (ACSD 2004), 16-18 June 2004, Hamilton, Canada*. IEEE Computer Society, 2004, pp. 57–66. DOI: 10.1109/CSD.2004.1309116 (cit. on p. 110).

[ACS05] Emil Axelsson, Koen Claessen, and Mary Sheeran. "Wired: Wire-Aware Circuit Design". In: *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings*. Ed. by Dominique Borrione and Wolfgang J. Paul. Vol. 3725. Lecture Notes in Computer Science. Springer, 2005, pp. 5–19. DOI: 10.1007/11560548_4 (cit. on p. 13).

[CPP05] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. "A conservative extension of synchronous data-flow with state machines". In: *EMSOFT 2005, September 18-22, 2005, Jersey City, NJ, USA, 5th ACM International Conference On Embedded Software, Proceedings*. Ed. by Wayne H. Wolf. ACM, 2005, pp. 173–182. DOI: 10.1145/1086228.1086261 (cit. on pp. 8, 9, 158).

[Hal05] Nicolas Halbwachs. "A synchronous language at work: the story of Lustre". In: *3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2005), 11-14 July 2005, Verona, Italy, Proceedings*. IEEE Computer Society, 2005, pp. 3–11. DOI: 10.1109/MEMCOD.2005.1487884 (cit. on p. 7).

[MC05] Jan Mikac and Paul Caspi. "Temporal refinement for Lustre". In: *International Workshop on Synchronous Languages, Applications and Programs*. Elsevier Science Publishers. 2005 (cit. on pp. xviii, 132).

[Mik05] Jan Mikác. "Raffinement et preuves de systèmes Lustre. (Refinements and Proofs of Lustre Systems)". PhD thesis. Grenoble Institute of Technology, France, 2005 (cit. on pp. 110, 175).

[Mor05] Lionel Morel. "Exploitation des structures régulières et des spécifications locales pour le developpement correct de systèmes réactifs de grande taille". PhD thesis. Institut National Polytechnique de Grenoble-INPG, 2005 (cit. on p. 110).

[CCH06] Miguel Castro, Manuel Costa, and Tim Harris. "Securing Software by Enforcing Data-flow Integrity". In: *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*. Ed. by Brian N. Bershad and Jeffrey C. Mogul. USENIX Association, 2006, pp. 147–160 (cit. on pp. 90, 175).

[Fla06] Cormac Flanagan. "Hybrid type checking". In: *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. Ed. by J. Gregory Morrisett and Simon L. Peyton Jones. ACM, 2006, pp. 245–256. DOI: 10.1145/1111037.1111059 (cit. on pp. 14, 41, 44, 173).

[Lee06] Edward A. Lee. "The Problem with Threads". In: *Computer* 39.5 (2006), pp. 33–42. DOI: 10.1109/MC.2006.180 (cit. on p. 3).

[Bou07] Pierre Boulet. "Array-OL revisited, multidimensional intensive signal processing specification". PhD thesis. INRIA, 2007 (cit. on p. 12).

[KF07] Kenneth L. Knowles and Cormac Flanagan. "Type Reconstruction for General Refinement Types". In: *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*. Ed. by Rocco De Nicola. Vol. 4421. Lecture Notes in Computer Science. Springer, 2007, pp. 505–519. DOI: 10.1007/978-3-540-71316-6_34 (cit. on pp. 15, 16, 47).

[Pey+07] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. "Practical type inference for arbitrary-rank types". In: *Journal of functional programming* 17.1 (2007), pp. 1–82. DOI: 10.1017/S0956796806006034 (cit. on pp. 15, 45).

[Bie+08] Dariusz Biernacki, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. "Clock-directed modular code generation for synchronous data-flow languages". In: *Proceedings of the 2008 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08), Tucson, AZ, USA, June 12-13, 2008*. Ed. by Krisztián Flautner and John Regehr. ACM, 2008, pp. 121–130. DOI: 10.1145/1375657.1375674 (cit. on pp. vii, 7, 8, 100).

[Cas+08] Paul Caspi, Albert Benveniste, Roberto Lublinerman, and Stavros Tripakis. *Actors without Directors: a Kahnian View of Heterogeneous Systems*. Tech. rep. 2008-6. Verimag Research Report, 2008 (cit. on p. 5).

[CF08] Sylvain Conchon and Jean-Christophe Filliâtre. "Semi-persistent Data Structures". In: *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by Sophia Drossopoulou. Vol. 4960. Lecture Notes in Computer Science. Springer, 2008, pp. 322–336. DOI: 10.1007/978-3-540-78739-6_25 (cit. on p. 19).

[RKJ08] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. "Liquid types". In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. Ed. by Rajiv Gupta and Saman P. Amarasinghe. ACM, 2008, pp. 159–169. DOI: 10.1145/1375581.1375602 (cit. on pp. 15, 47).

[SSC08] Joel Svensson, Mary Sheeran, and Koen Claessen. "Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors". In: *Implementation and Application of Functional Languages - 20th International Symposium, IFL 2008, Hatfield, UK, September 10-12, 2008. Revised Selected Papers*. Ed. by Sven-Bodo Scholz and Olaf Chitil. Vol. 5836. Lecture Notes in Computer Science. Springer, 2008, pp. 156–173. DOI: 10.1007/978-3-642-24452-0_9 (cit. on p. 22).

[PR09] Marc Pouzet and Pascal Raymond. "Modular static scheduling of synchronous data-flow networks: an efficient symbolic representation". In: *Proceedings of the 9th ACM & IEEE International conference on Embedded software, EMSOFT 2009, Grenoble, France, October 12-16, 2009*. Ed. by Samarjit Chakraborty and Nicolas Halbwachs. ACM, 2009, pp. 215–224. DOI: 10.1145/1629335.1629365 (cit. on p. 8).

[TG09] Kai Trojahner and Clemens Grelck. "Dependently typed array programs don't go wrong". In: *The Journal of Logic and Algebraic Programming* 78.7 (2009), pp. 643–664. DOI: 10.1016/j.jlap.2009.03.002 (cit. on pp. 15, 16).

[Ben+11] Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet. "Divide and recycle: types and compilation for a hybrid synchronous language". In: *Proceedings of the ACM SIGPLAN/SIGBED 2011 conference on Languages, compilers, and tools for embedded systems, LCTES 2011, Chicago, IL, USA, April 11-14, 2011*. Ed. by Jan Vitek and Bjorn De Sutter. ACM, 2011, pp. 61–70. DOI: 10.1145/1967677.1967687 (cit. on p. 5).

[KT11] Temesghen Kahsai and Cesare Tinelli. "PKind: A parallel k-induction based model checker". In: *Proceedings 10th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2011, Snowbird, Utah, USA, July 14, 2011*. Ed. by Jiri

Barnat and Keijo Heljanko. Vol. 72. EPTCS. 2011, pp. 55–62. DOI: 10.4204/EPTCS.72.6 (cit. on p. 5).

[CSS12] Koen Claessen, Mary Sheeran, and Joel Svensson. "Expressive array constructs in an embedded GPU kernel programming language". In: *Proceedings of the POPL 2012 Workshop on Declarative Aspects of Multicore Programming, DAMP 2012, Philadelphia, PA, USA, Saturday, January 28, 2012*. Ed. by Umut A. Acar and Vítor Santos Costa. ACM, 2012, pp. 21–30. DOI: 10.1145/2103736.2103740 (cit. on pp. 22, 27, 149).

[Gér+12] Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet. "A modular memory optimization for synchronous data-flow languages: application to arrays in a lustre compiler". In: *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2012, LCTES '12, Beijing, China - June 12 - 13, 2012*. Ed. by Reinhard Wilhelm, Heiko Falk, and Wang Yi. ACM, 2012, pp. 51–60. DOI: 10.1145/2248418.2248426 (cit. on pp. 12, 19, 79, 174).

[BP13] Timothy Bourke and Marc Pouzet. "Zélus: a synchronous language with ODEs". In: *Proceedings of the 16th international conference on Hybrid systems: computation and control, HSCC 2013, April 8-11, 2013, Philadelphia, PA, USA*. Ed. by Calin Belta and Franjo Ivancic. ACM, 2013, pp. 113–118. DOI: 10.1145/2461328.2461348 (cit. on p. 5).

[GR13] Jacques Garrigue and Didier Rémy. "Ambivalent Types for Principal Type Inference with GADTs". In: *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*. Ed. by Chung-chieh Shan. Vol. 8301. Lecture Notes in Computer Science. Springer, 2013, pp. 257–272. DOI: 10.1007/978-3-319-03542-0_19 (cit. on p. 50).

[Pas13] Cédric Pasteur. "Raffinement temporel et exécution parallèle dans un langage synchrone fonctionnel". PhD thesis. Pierre and Marie Curie University, Paris, France, 2013 (cit. on p. 110).

[Rag+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 519–530. DOI: 10.1145/2491956.2462176 (cit. on pp. viii, 13, 28).

[Ben+14] Albert Benveniste, Timothy Bourke, Benoît Caillaud, Bruno Pagano, and Marc Pouzet. "A type-based analysis of causality loops in hybrid systems modelers". In: *17th International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC'14, Berlin, Germany, April 15-17, 2014*. Ed. by Martin Fränzle and John Lygeros. ACM, 2014, pp. 71–82. DOI: 10.1145/2562059.2562125 (cit. on p. 87).

[Pot14] François Pottier. "Hindley-milner elaboration in applicative style: functional pearl". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. Ed. by Johan Jeuring and Manuel M. T. Chakravarty. ACM, 2014, pp. 203–212. DOI: 10.1145/2628136.2628145 (cit. on pp. 145, 146).

[Rag14] Jonathan Ragan-Kelley. "Decoupling algorithms from the organization of computation for high performance image processing". PhD thesis. Massachusetts Institute of Technology, Cambridge, MA, USA, 2014 (cit. on pp. 21, 177).

[SSM14] Justin Slepak, Olin Shivers, and Panagiotis Manolios. "An Array-Oriented Language with Static Rank Polymorphism". In: *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Ed. by Zhong Shao. Vol. 8410. Lecture Notes in Computer Science. Springer, 2014, pp. 27–46. DOI: 10.1007/978-3-642-54833-8_3 (cit. on p. 15).

[Bou+15] Timothy Bourke, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. "A Synchronous-Based Code Generator for Explicit Hybrid Systems Languages". In: *Compiler Construction - 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Björn Franke. Vol. 9031. Lecture Notes in Computer Science. Springer, 2015, pp. 69–88. DOI: 10.1007/978-3-662-46663-6_4 (cit. on p. 9).

[MPP15] Louis Mandel, Cédric Pasteur, and Marc Pouzet. "Time refinement in a functional synchronous language". In: *Science of Computer Programming* 111 (2015), pp. 190–211. DOI: 10.1016/j.scico.2015.07.002 (cit. on pp. xviii, 132, 175).

[VV15] Field G. Van Zee and Robert A. Van de Geijn. "BLIS: A Framework for Rapidly Instantiating BLAS Functionality". In: *ACM Transactions on Mathematical Software (TOMS)* 41.3 (2015), 14:1–14:33. DOI: 10.1145/2764454 (cit. on p. 17).

[Cha+16] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. "The Kind 2 Model Checker". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 510–517. DOI: 10.1007/978-3-319-41540-6_29 (cit. on p. 5).

[Gua16] Adrien Guatto. "A synchronous functional language with integer clocks. (Un langage synchrone fonctionnel avec horloges entières)". PhD thesis. PSL Research University, Paris, France, 2016 (cit. on p. 110).

[Mul+16] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. "Automatically scheduling halide image processing pipelines". In: *ACM Transactions on Graphics (TOG)* 35.4 (2016), 83:1–83:11. DOI: 10.1145/2897824.2925952 (cit. on p. 22).

[Bou+17] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. "A formally verified compiler for Lustre". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 586–601. DOI: 10.1145/3062341.3062358 (cit. on pp. vii, 8, 101).

[CPP17] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. "SCADE 6: A formal language for embedded critical software development (invited paper)". In: *11th International Symposium on Theoretical Aspects of Software Engineering, TASE 2017, Sophia Antipolis, France, September 13-15, 2017*. Ed. by Frédéric Mallet, Min Zhang, and Eric Madelaine. IEEE Computer Society, 2017, pp. 1–11. DOI: 10.1109/TASE.2017.8285623 (cit. on pp. vii, 3, 9, 82, 99, 109).

[Hen+17] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. "Futhark: purely functional GPU-programming with nested parallelism and in-place array updates". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 556–571. DOI: 10.1145/3062341.3062354 (cit. on pp. viii, 17, 19, 22, 37, 38, 40, 49, 72, 75, 149, 160).

[Sin+17] Artjoms Sinkarovs, Sven-Bodo Scholz, Robert J. Stewart, and Hans-Nikolai Vießmann. "Recursive Array Comprehensions in a Call-by-Value Language". In: *Proceedings of the 29th Symposium on Implementation and Application of Functional Programming Languages, IFL 2017, Bristol, UK, August 30 - September 01, 2017*. Ed. by Nicolas Wu. ACM, 2017, 5:1–5:12. DOI: 10.1145/3205368.3205373 (cit. on pp. 36, 121, 176).

[Che+18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. Ed. by Andrea C.

Arpaci-Dusseau and Geoff Voelker. USENIX Association, 2018, pp. 578–594 (cit. on p. 22).

[Col+18] Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. "Scade 6: From a Kahn Semantics to a Kahn Implementation for Multicore". In: *2018 Forum on Specification & Design Languages, FDL 2018, Garching, Germany, September 10-12, 2018*. Ed. by Hiren D. Patel, Tom J. Kazmierski, and Sebastian Steinhorst. IEEE, 2018, pp. 5–16. DOI: 10.1109/FDL.2018.8524052 (cit. on p. 9).

[KN18] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. USA: No Starch Press, 2018. ISBN: 1593278284 (cit. on p. 18).

[Ray18] Pascal Raymond. *Synchronous program verification with Lustre/Lesar*. Tech. rep. TR-2018-11. Verimag Research Report, 2018 (cit. on pp. 5–7).

[BBP20] Timothy Bourke, Lélio Brun, and Marc Pouzet. "Mechanized semantics and verified compilation for a dataflow synchronous language with reset". In: *Proceedings of the ACM on Programming Languages* 4.POPL (2020), 44:1–44:29. DOI: 10.1145/3371112 (cit. on p. 8).

[Bru20] Lélio Brun. "Mechanized semantics and verified compilation for a dataflow synchronous language with reset. (Sémantique mécanisée et compilation vérifiée pour un langage synchrone à flots de données avec réinitialisation)". PhD thesis. Paris Sciences et Lettres University, France, 2020 (cit. on p. 101).

[Dag+20] Francesco Dagnino, Viviana Bono, Elena Zucca, and Mariangiola Dezani-Ciancaglini. "Soundness Conditions for Big-Step Semantics". In: *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. Ed. by Peter Müller. Vol. 12075. Lecture Notes in Computer Science. Springer, 2020, pp. 169–196. DOI: 10.1007/978-3-030-44914-8_7 (cit. on pp. 44, 48, 179, 181, 182).

[Bou+21] Timothy Bourke, Paul Jeanmaire, Basile Pesin, and Marc Pouzet. "Verified Lustre Normalization with Node Subsampling". In: *ACM Trans. Embed. Comput. Syst.* 20.5s (2021), 98:1–98:25. DOI: 10.1145/3477041 (cit. on p. 100).

[HE21] Troels Henriksen and Martin Elsman. "Towards size-dependent types for array programming". In: *ARRAY 2021: Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming, Virtual Event, Canada, 21 June, 2021*. Ed. by Tze Meng Low and Jeremy Gibbons. ACM, 2021, pp. 1–14. DOI: 10.1145/3460944.3464310 (cit. on pp. 15, 16).

[IGR21] Paul Iannetta, Laure Gonnord, and Gabriel Radanne. "Compiling pattern matching to in-place modifications". In: *GPCE '21: Concepts and Experiences, Chicago, IL, USA, October 17 - 18, 2021*. Ed. by Eli Tilevich and Coen De Roover. ACM, 2021, pp. 123–129. DOI: 10.1145/3486609.3487204 (cit. on p. 27).

[Pas+21] Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. "Getting to the point: index sets and parallelism-preserving autodiff for pointful array programming". In: *Proceedings of the ACM on Programming Languages* 5.ICFP (2021), pp. 1–29. DOI: 10.1145/3473593 (cit. on pp. 15, 16).

[TMŚ21] Stephen Thomas, Lenore Mullin, and Katarzyna Świrydowicz. "Improving the Performance of DGEMM with MoA and Cache-Blocking". In: *ARRAY '21: ACM Symposium on Array Programming, 20 – 26 June, 2021, Virtual*. ACM, 2021. DOI: 10.1145/1122445.1122456 (cit. on p. 17).

[Clé22] Basile Clément. "Translation Validation of Tensor Compilers. (Validation de Traduction pour Compilateurs de Tenseurs)". PhD thesis. École Normale Supérieure, Paris, France, 2022 (cit. on pp. 22, 177).

[CPP22] Jean-Louis Colaço, Baptiste Pauget, and Marc Pouzet. "Inférer et vérifier les tailles de tableaux avec des types polymorphes". In: *33èmes Journées Francophones des Langages Applicatifs*. 2022 (cit. on p. 37).

[Col+23] Jean-Louis Colaco, Michael Mendler, Baptiste Pauget, and Marc Pouzet. "A Constructive State-based Semantics and Interpreter for a Synchronous Data-flow Language with State machines". In: *International Conference on Embedded Software (EMSOFT), September 17-22, Hamburg, Germany*. ACM. 2023 (cit. on pp. 5, 132, 175).

[CPP23] Jean-Louis Colaço, Baptiste Pauget, and Marc Pouzet. "Polymorphic Types with Polynomial Sizes". In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming, ARRAY 2023, Orlando, FL, USA, 18 June 2023*. Ed. by Troels Henriksen and Artjoms Sinkarovs. ACM, 2023, pp. 36–49. DOI: 10.1145/3589246.3595372 (cit. on p. 37).

[LLS+23] Anton Lorenzen, Daan Leijen, Wouter Swierstra, et al. *FP^ 2: Fully in-Place Functional Programming*. Tech. rep. MSR-TR-2023-19. Microsoft Research, 2023 (cit. on p. 19).

[Sen23] Andrew Sengul. "Faster APL with Lazy Extensions". In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming, ARRAY 2023, Orlando, FL, USA, 18 June 2023*. Ed. by Troels Henriksen and Artjoms Sinkarovs. ACM, 2023, pp. 62–74. DOI: 10.1145/3589246.3595374 (cit. on p. 13).

Index

RÉSUMÉ

SCADE est un langage de programmation synchrone utilisé depuis la fin des années 1990 pour concevoir et implémenter des systèmes critiques embarqués tels ceux que l'on trouve dans l'aviation. Cette criticité requiert la formalisation (i) des programmes, afin de garantir qu'ils s'exécutent sans erreurs ou retards pendant une durée arbitraire et (ii) des outils de développement, en particulier le compilateur, pour s'assurer de la préservation des propriétés des modèles lors de la génération de code. Ces activités reposent sur une description flot de données des modèles, inspirée des schémas-blocs, dont la sémantique s'exprime par des fonctions de suites, ainsi que d'une compilation précisément documentée.
SCADE descend de LUSTRE. La version 6 a introduit des évolutions majeures, dans les pas de LUCID SYNCHRONE, dont les automates hiérarchiques et les tableaux. Pour ces derniers, leur taille est connue et vérifiée statiquement afin d'assurer un temps d'exécution et une mémoire bornés. L'utilisation croissante des tableaux de SCADE pour des opérations intensives a révélé plusieurs axes d'amélioration: verbosité des modèles, manque de contrôle de la compilation ou description peu commode de certains algorithmes.
Dans cette thèse, ces trois aspects ont été étudiés à l'aide d'un prototype de compilateur. (i) Nous avons développé un système de types $a\ la$ Hindley-Milner spécifiant les tailles sous la forme de polynômes multivariés. Cette proposition permet de vérifier et d'inférer la plupart des tailles de manière modulaire. (ii) Nous avons exploré une méthode de compilation alternative, fondée sur un *langage déclaratif conscient de la mémoire*. Il vise à concilier le style flot de données avec une spécification précise des emplacements mémoire. La description modulaire des tailles en est un élément clé. (iii) Enfin, nous proposons une construction d'itération inspirée de SISAL qui complète les itérateurs actuels. En traitant les tableaux comme des suites finies, elle donne accès aux constructions séquentielles de SCADE (automates) lors d'itérations. De plus, elle permet de décrire de manière déclarative des implémentations efficaces d'algorithmes comme la décomposition de Cholesky. Cette compilation contrôlable est un premier pas nécessaire pour la compilation vers des GPUs.

MOTS CLÉS

Systèmes embarqués ⋆ Programmation synchrone ⋆ Tableaux ⋆ Systèmes de types ⋆ Compilation

ABSTRACT

The synchronous programming language SCADE has been used since the end of the 1990s to design safety critical embedded systems such as those found in avionics. This context requires to formally reason about (i) the programs, to ensure that they run for an arbitrary long time without errors or missed deadlines and (ii) the tools, in particular the compiler, to give high confidence in the preservation of model properties through the code generation process. These activities build on a data-flow description of models, inspired by block diagrams, that enjoys a formal semantics in terms of streams and a well-documented compilation process.
SCADE stems from LUSTRE. The SCADE6 version introduced major evolutions following LUCID SYNCHRONE, notably hierarchical automata and arrays. For the latter, sizes are statically known and checked, so as to fulfill the bounded resources and execution time constraints. The increasing use of SCADE arrays for data-intensive computations has revealed areas for improvement: concision of the models, control of the generated code and idiomatic description of array computations.
Using a prototype implementation of a compiler, the present work investigates these three aspects. (i) We designed a Hindley-Milner-like type system for representing sizes with multivariate polynomials. This proposal allows to check and infer most sizes in a modular way. (ii) We explored an alternative compilation process based on a *memory-aware declarative language*. It aims at reconciling the data-flow style with a precise specification of memory locations. The underlying memory model builds on our modular description of sizes. (iii) Last, we propose an iteration construct inspired by SISAL that supplements the available iterators. By viewing arrays as finite streams, iteration can benefit from the sequential constructs of SCADE, e.g., automata. Moreover, it allows declarative descriptions of efficient algorithm implementations such as the Cholesky decomposition. This controllable compilation process is a mandatory step toward code generation for GPUs.

KEYWORDS

Embedded systems ⋆ Synchronous programming ⋆ Arrays ⋆ Type systems ⋆ Compilation