HAL
open science

Refactoring functional programs with ornaments

Ambre Williams

To cite this version:

Ambre Williams. Refactoring functional programs with ornaments. Programming Languages [cs.PL]. Université de Paris / Université Paris Diderot (Paris 7), 2020. English. NNT : . tel-03126602v1

HAL Id: tel-03126602

https://inria.hal.science/tel-03126602v1

Submitted on 31 Jan 2021 (v1), last revised 30 Jun 2021 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université de Paris

Université de Paris
Ecole doctorale 386 – Sciences Mathématiques de Paris Centre
Inria

Refactoring functional programs with ornaments

Par Ambre Williams

Thèse de doctorat d'Informatique

Dirigée par Didier Rémy

Présentée et soutenue publiquement le 14 décembre 2020

Devant un jury composé de :

Conor McBride, Reader, University of Strathclyde, rapporteur
Nicolas Tabareau, Directeur de recherche, Inria, rapporteur
Chantal Keller, Maître de conférences, Université Paris-Saclay
Delia Kesner, Professeur, Université de Paris, présidente
Yann Régis-Gianas, Maître de conférence, Nomadic Labs
Didier Rémy, Directeur de recherche, Inria, directeur de thèse

Cette œuvre est mise à disposition sous la licence
https://creativecommons.org/licenses/by/3.0/fr/

Titre : Refactorisation de programmes fonctionnels par les ornements

Résumé :
Les ornements fournissent un moyen de définir des transformations de définitions de types de données inductifs réorganisant, spécialisant et ajoutant des champs à des types de données déjà existants. À partir d'une telle transformation, nous nous intéressons à la refactorisation semi-automatique d'un code déjà existant, écrit en ML et opérant sur le type de base pour le faire opérer sur le type ornementé. Nous décrivons un cadre pour de telles transformations, basé sur deux phases : tout d'abord, le terme de base est généralisé au maximum en ajoutant des abstractions sur les détails des types de données utilisés. Le terme est ensuite spécialisé pour opérer uniquement sur le type ornementé. Nous décrivons un langage intermédiaire fournissant les abstractions nécessaires pour présenter le terme générique, et garantissant qu'il est possible de simplifier le terme spécialisé pour ne présenter à l'utilisateur que des termes du langage de base. Le langage intermédiaire permet notamment d'exprimer des abstractions dépendantes, de représenter les égalités apprises par le filtrage de motifs, et fournit une construction permettant de se référer de façon opaque au résultats de calculs déjà effectués. Nous exploitons la paramétricité du terme généralisé pour prouver une relation de cohérence entre le terme de base et le terme ornementé, garantissant la correction de la refactorisation. Nous présentons une implémentation de cette transformation sur un langage ML noyau, et justifions de son utilité dans de nombreux cas courants de transformation de programme : refactorisation pure, ajout de nouvelles données et spécialisation. Nous présentons aussi une nouvelle technique de dépliage permise par notre transformation qui autorise à changer la structure récursive des fonctions, et illustrons son utilité pour optimiser certaines représentations de données et pour la programmation générique.

Mots clefs : ornements ; langages de programmation fonctionnelle ; types ; relations logiques ; refactorisation ; ML.

Title: Refactoring functional programs with ornaments

Abstract:
Ornaments provide a way to express transformations of inductive datatypes that reorganize, specialize, and add fields to already existing datatypes. From such a transformation, we consider the problem of semi-automatically refactoring an al-

ready existing ML program operating on the base type into a program operating on the ornamented type. We describe a framework for such transformations by decomposing them in two phases: first, the base term is maximally generalized by abstracting over the details of the manipulated datatypes. This generic term is subsequently specialized to operate only on the ornamented type. We describe an intermediate language providing the necessary abstractions to present the generic term. This language notably includes dependent abstractions, allows representing the equalities learned from pattern matching, and provide a way to refer to the opaque result of previous computations. We exploit the parametricity of the generic term to derive a coherence relation between the base term and the ornamented term, guaranteeing the correctness of the ornamented term. We present an implementation of this transformation on a core ML language, and illustrate its usefulness in many common cases of program transformation: pure refactoring, adding new data and specialization. We also present a new unfolding technique afforded by our transformation and present its use for optimizing certain data representation and for generic programming.

Keywords: ornaments; functional programming languages; types; logical relations; refactoring; ML.

Remerciements

J'aurais dû commencer par rédiger ce chapitre pour n'oublier personne, je l'écris finalement la veille de ma soutenance. J'espère que celles et ceux que j'aurai inévitablement oublié de remercier ici pourront m'en excuser.

J'ai commencé à travailler sur les ornements dans le cadre d'un stage de M2, proposé par Didier Rémy et Pierre-Évariste Dagand. Pendant ce stage, j'ai découvert l'équipe Gallium, avec qui il a été très agréable de travailler. Merci pour les cafés et Café-vous-fait, les discussions de cantine, pour les conseils pendant mes répétitions d'exposés, et merci à tous ceux avec qui j'ai partagé un bureau. Vous avez participé à me convaincre de rester pour ma thèse.

Je ne pouvais espérer meilleur directeur que Didier : il a toujours été disponible pour passer plusieurs heures à discuter de mon avancement, apportant un oeil extérieur bienvenu et des idées fraîches quand je rencontrais des difficultés. Beaucoup de cette thèse n'existe que grâce à ses retours, m'encourageant à développer les concepts intéressants, les rendre plus clairs et les généraliser.

En septembre 2018, alors que j'étais déjà en retard, j'estimais qu'il ne me manquait que deux mois avant d'avoir fini de rédiger. Avec la distraction d'un emploi à plein temps, et parce que j'avais sous-estimé le travail restant, cela m'a finalement pris deux ans, pendant lesquels certaines idées ont pu mûrir pour arriver à une présentation, je l'espère, plus clair et intéressante.

Je remercie beaucoup Didier pour m'avoir poussé à continuer pendant cette période, m'avoir rappelé des deadlines administratives que j'aurais sans doute manquées, et avoir continué à prendre régulièrement du temps avec moi pour discuter de mon avancement. J'ai aussi été beaucoup soutenue par Annalí, qui à de nombreuses occasions m'a dissuadé de simplement arrêter. Son aide pendant les derniers mois de la rédaction a été indispensable. Je la remercie aussi beaucoup de sa patience pour les soirées et les week-ends volés par la rédaction, et pour avoir supporté mon stress à une période qui n'était pas non plus facile pour elle. Merci aussi à mes parents pour m'avoir encouragée dans ce processus.

J'ai rédigé une partie de cette thèse au Mexique. Mario y Susana, muchas gracias por la hospitalidad. ¡Espero que nos podamos ver pronto! J'ai passé les deux dernières semaines de rédaction dans le très agréable village de Gratallops, en Catalogne. Moltes gràcies Mireia, una forta abraçada! I also spent a few weeks at Oscar Wilde House in Berkeley while Annalí was staying there. Thank you all for the food and the company.

Je n'ai heureusement pas pensé qu'à la thèse pendant ces six ans. Merci beaucoup à Arthur, Épiphanie, Théo, Cécile, Maiting, Jonathan, Loïc, Irène, Renaud, Guillaume, Ulysse, Anaël, Kenji, Basile, Damien, Julie, Abel, Najib, Aymeric, Grégoire, pour les bons moments passés ensemble, ma famille, qui m'a beaucoup manquée ces derniers mois, et à Annalí pour avoir été tout ce temps une lumière dans ma vie.

Résumé en français

Un programme est une description en terme précis des opérations qu'un ordinateur doit suivre afin d'accomplir une tâche donnée. Les instructions élémentaires interprétées par un ordinateur son extrêmement rudimentaires. Il faut parfois plusieurs centaines d'instructions pour trier une liste de nombres entiers ; les programmes les plus complexes, tels les navigateurs web, sont composés de dizaines de millions de ces instructions. Les langages de programmation sont des outils d'abstraction : ils permettent aux programmeurs de ne donner qu'une explication à un haut niveau d'abstraction de ce que le programme doit faire. Des compilateurs et des interprètes traduisent les représentations de haut niveau manipulées par les programmeurs en instructions de bas niveau. En libérant les programmeurs de la nécessité de penser à l'échelle des instructions élémentaires, les langages de programmation permettent aux programmeurs de construire des programmes plus complexes, plus vite.

Un programme n'est que rarement écrit d'une seule traite, sans ne rien effacer, et distribué pour n'être plus jamais modifié. Même lorsqu'on écrit la version initiale d'un programme, on peut ne pas avoir une idée exacte de la forme qu'il va prendre, et on a souvent besoin de revenir en arrière et changer certaines définitions à mesure que notre compréhension du problème s'affine. Une fois achevée l'écriture de la version initiale, on peut aussi s'apercevoir que les besoins ont changés, nécessitant un code plus général par certains aspects et moins général par d'autres : on doit alors changer certaines parties du programme pour correspondre aux nouvelles demandes. Tous ces changements ne sont pas locaux : même si l'essentiel du changement consiste en ne modifier que quelques éléments du programme, il peut demander une *refactorisation* très étendue pour faire circuler les nouvelles données à travers le code déjà existant. La version refactorisée d'un fragment de code peut venir remplacer le code déjà présent, ou les deux versions du code peuvent être amenées à coexister dans le programme, pour être utilisées dans des contextes différents. Ces transformations sont souvent longues, répétitives, et sans intérêt. Face à une telle tâche, un programmeur a souvent une idée générale de la transformation à réaliser, mais se retrouve obligé d'effectuer manuellement chaque changements requis pour cette refactorisation. Notre but est de fournir des outils rendant cette transformation plus rapide et facile.

Si le programmeur a pris soin d'éviter les effets de bords, les différents fragments du programme communique en échangeant des données à travers les arguments de fonctions et leurs valeurs de retours. Ces données sont contraintes : les parties du programme qui communiquent doivent s'accorder sur un format pour les données qu'elles échangent. Si le langage de programmation le permet, une partie de ces contraintes peut s'exprimer sous forme de *types*. Les types peuvent

être vus comme un système de classification de valeurs et de programmes qui, de plus, est compréhensible par le langage de programmation. Les types forment un outil puissant : ils permettent au langage de programmation de rejeter un programme dès la compilation car il pourrait à l'exécution atteindre un état incorrect. Les types jouent aussi pour le programmeur un rôle de documentation dont l'exactitude est garantie par le langage. En lisant le type d'une fonction, on apprend des contraintes sur la façon dont elle doit être appelée, et on obtient des garanties sur le comportement et les valeurs retournées par cette fonction. Cela permet de s'assurer que des catégories entières de comportement erronés ne peuvent se produire.

Bien qu'ils ne caractérisent le comportement que d'une seule version d'un programme considéré en isolation, les types fournissent un guide utile pour la refactorisation : lorsque l'un des effets de la refactorisation est de changer le type de certaines données, le système de types peut orienter un programmeur vers les emplacements du programme qui sont maintenant mal typés. Le programmeur peut alors effectuer les changements locaux nécessaires et relancer le typeur. Le typeur va alors soit pointer vers d'autres emplacements qui sont maintenant mal typés, soit accepter le programme, concluant ainsi la refactorisation. Ce n'est pourtant pas une solution satisfaisante. Premièrement, certaines transformations ne changent pas les types, seulement la façon dont leurs valeurs sont interprétées. Par exemple, si la transformation inverse le sens d'une valeur booléenne, le système de types sera incapable d'indiquer les endroits où le code doit être modifié, et le programmeur devra garder trace de ces emplacements lui-même. De plus, cela reste un processus extrêmement manuel et ennuyeux : le programmeur ne doit plus chercher lui-même les emplacements à modifier, mais doit tout de même effectuer chaque changement manuellement. Ces changements, qui devraient être simplement mécaniques, sont aussi sources d'erreurs. Tout cela décourage les programmeurs d'effectuer de tels changements : à la place, ils choisissent souvent de laisser un commentaire indiquant qu'un refactorisation devrait être effectuée.

On attend d'un outil de refactorisation qu'il élimine autant que possible ces coûts, ne laisse que le minimum de travail au programmeur, et qu'il fournisse une garantie solide que le nouveau programme est seulement une adaptation du programmeur d'origine à de nouvelles données.

Dans la famille des langages dérivés de ML, les types de données inductifs sont l'outil principal pour décrire l'organisation des données. On les définit comme des sommes étiquetées de produits de types primitifs et d'autres types inductifs. Les produits permettent de former des valeurs composées de plusieurs autres valeurs, permettant par abstraction de les manipuler comme une valeur singulière. Par exemple, plutôt que de manipuler individuellement des paires de coordonnées, un programmeur trouvera plus pratique de manipuler un point à deux dimensions comme une unité de programmation et de pensée, et il ne considérera les deux coordonnées que pour définir des opérations basiques sur ces points. Les sommes permettent l'expression de choix : une forme géométrique est soit un triangle, défini par ses trois sommets, ou un cercle, défini par son centre et son rayon. Finalement, la récursion permet de représenter des structures de taille illimitée : une liste de formes géométriques est soit la liste vide, soit une liste formée en ajoutant un élément en tête d'une autre liste.

Une même structure de données peut souvent être représentée par plusieurs structures isomorphes, en utilisant un arrangement différent de sommes et de

produits. Deux structures de données peuvent aussi n'être différente que par des éléments mineurs. Par exemple, le type des arbres binaires portant des valeurs aux feuilles et le type des arbres binaires portant des valeurs aux branches partagent une même structure où chaque noeud possède deux sous-arbres, mais les fonctions opérant sur ces deux types doivent être définies séparément.

Puisque les types de données inductifs sont la manière principale de représenter les données en ML, et puisque, dans des programmes bien structurés, des modules distants interagissent principalement par l'échange de données, la transformation des types inductifs est une catégorie importante de refactorisation en ML.

La théorie des *ornements* [McBride, 2010, Dagand and McBride, 2013, 2014] fournit un cadre dans lequel exprimer ces transformations. Elle définit des conditions sous lesquelles une nouvelle définition de données peut être décrite comme un *ornement* d'une autre. Un ornement est essentiellement une relation entre deux types de données inductifs qui réorganise, spécialise et ajoute des données à un type de base pour obtenir un type ornementé.

Prenant les ornements comme brique de base, on peut définir des *types d'ornement* qui expriment une relation entre un terme de base et un terme *lifté* : un type d'ornement n'est pas simplement une spécification *unaire* qui classifie des termes pris individuellement, mais plutôt une spécification *binaire* qui indique comment le comportement du terme de base, qui opère sur le type de base, est lié au comportement du terme lifté, qui opère sur le type ornementé. Ces types donnent donc une spécification de l'opération de refactorisation : un lifting est correct si son résultat est lié au terme de base par la relation définie par le type d'ornement demandé.

Nous décrivons dans cette thèse une approche bien-fondée pour obtenir des liftings garantis corrects. Cette approche est basée sur la construction *a posteriori* d'une variante la plus générale possible en abstrayant sur les détails de la représentation de données, qui peut ensuite être spécialisée pour obtenir un lifting concret. Les abstractions nécessaires ne peuvent pas être exprimées en ML. Pour la construire, nous définissions mML, une version de ML étendue avec les mécanismes d'abstraction nécessaires, mais construire de sorte à garantir que les termes spécialisés puissent être simplifiés pour obtenir à nouveau des termes de ML. Nous montrons ensuite que le terme de base peut être exprimé comme une instance particulière du terme généralisé, où l'on sélectionne systématiquement l'ornement identité, qui lie un type avec lui-même. Cela nous permet d'exploiter la paramétricité pour prouver l'existence de la relation désirée entre le terme de base et le terme ornementé.

Ce travail théorique est accompagné d'une implémentation opérant sur une version noyau de ML avec types de données et polymorphisme. Ce prototype, disponible en ligne sur `https://morphis.me/ornaments/`, met en œuvre un certain nombre de techniques, que nous décrivons dans ce manuscrit, pour générer du code proche de ce qu'un programmeur aurait écrit manuellement. Il est accompagné d'une bibliothèque d'exemples illustrant ses capacités.

La première partie de ce manuscrit introduit le concept d'ornement par le biais d'exemples illustrant différentes possibilités d'utilisation : refactorisation pure, ajout de données, spécialisation, optimisation et programmation générique. Ces exemples servent aussi d'illustration des capacités de notre implémentation. Nous donnons ensuite une vue d'ensemble du processus d'ornementation : nous faisons abstraction des détails les plus techniques et présentons les

principes généraux guidant la transformation.

Dans une deuxième partie, nous laissons pour un moment de côté les ornements, et nous nous concentrons sur la conception et la formalisation des briques de base permettant de construire notre transformation. Bien qu'ils aient étés conçus pour permettre l'ornementation, ces briques sont indépendantes des ornements et pourraient être appliquées pour réaliser d'autres transformations de programme. Nous présentons tout d'abord une définition et la méta-théorie de notre ML noyau. Nous ajoutons ensuite les fonctionnalités nécessaires à la transformation désirée. Tout d'abord, les programmes font en pratique intervenir des effets de bord : ils peuvent ne pas terminer, terminer avec une erreur, tirer des nombres aléatoires, ou lire et écrire des fichiers. Ces opérations, non purement fonctionnelle, sont un obstacle à la transformation de programmes : elles ne peuvent pas être librement dupliquées, réordonnées, ou supprimées. Nous définissions un système d'étiquetage permettant de se référer au résultat d'une de ces opérations précédemment effectuée sans avoir à dupliquer l'opération elle-même. Nous utilisons ces étiquettes pour définir eML, une extension de ML ajoutant une construction de filtrage par motifs au niveau des types. Pour permettre de réduire le filtrage au niveau des types, le filtrage au niveau des termes introduit une égalité entre le terme sur lequel on filtre et le motif de la branche sélectionnée. Si le terme sur lequel on filtre contient une application de fonction, on utilise son étiquette pour s'y référer, tout en gardant l'application opaque pour les règles de raisonnement de eML. Cela permet notamment de fournir un jugement d'égalité décidable. Nous définissons ensuite un langage appelé mML en ajoutant des abstractions dépendantes à eML. Nous prêtons particulièrement attention à isoler ces abstractions pour garantir qu'il soit possible de les réduire entièrement sans évaluer les parties eML du terme : nous garantissons ainsi qu'elles peuvent être éliminées du terme lifté avant de le présenter à l'utilisateur. Afin de raisonner sur mML, nous définissons une relation logique indicée. Cette relation fournit un cadre sur lequel il sera possible de définir des ornements d'ordre supérieur à partir d'ornements de base, et d'exprimer la relation entre un type de base et un type ornementé. Finalement, nous décrivons un procédé permettant de simplifier un terme eML pour obtenir un terme ML bien typé, garantissant ainsi que les constructions puissantes utilisées pour la transformation ne soient plus apparentes dans le code généré.

Une fois les briques de base définies, nous revenons à la définition de notre transformation, et nous construisons une preuve de sa correction. Nous décrivons tout d'abord comment transformer une définition d'ornement fournie par l'utilisateur à la fois en une définition de relation compatible avec la relation logique, et en fragments de code utilisés pour spécialiser le terme générique. Nous définissons, relativement à la relation induite, un critère de correction pour ces fragments de code, et montrons qu'il est satisfait. Nous décrivons ensuite comment transformer un terme de base en terme générique. Nous définissons sa spécialisation par l'ornement identité, et prouvons qu'elle est équivalente au terme de base. Finalement, nous expliquons comment une demande de lifting fournie par l'utilisateur est transcrite en une spécialisation d'un terme générique, et prouvons le lien entre le terme de base et le terme ornementé.

Nous nous tournons ensuite à l'aspect pratique du travail : nous décrivons certains aspects clés de l'implémentation du prototype, et les stratégies utilisées pour présenter à l'utilisateur du code lisible, ainsi que les améliorations permettant de réduire le travail nécessaire pour spécifier un lifting. Nous présentons une

technique de *dépliage* permettant de définir des liftings qui ne respectent pas la structure récursive de la fonction de base, et nous présentons des exemples de manipulations de structure de données rendues possibles par cette technique. Nous décrivons finalement quelques pistes de travail dans la continuation de cette thèse, et les liens avec des travaux existants.

Contents

1 Introduction **21**

I Ornaments in practice **25**

2 Ornaments by example **27**
2.1 Code refactoring 27
2.2 Code refinement 29
2.3 Composing transformations: a practical use case 31
2.4 Hiding administrative data 35
2.5 Higher-order types, recursive types 37

3 Overview of the lifting process **39**
3.1 Encoding ornaments 40
3.1.1 Eliminating the encoding 41
3.1.2 Inferring a generic lifting 42

II A calculus for program transformation **45**

4 Core ML **49**
4.1 Notation 49
4.2 Types and datatypes 49
4.3 The syntax of explicit ML 53
4.4 Evaluation 55
4.5 Type soundness 56

5 Labelling ML terms **61**
5.1 Overview 61
5.2 Labelled reduction 63
5.3 Full reduction 66
5.4 An attempt at well-labelling 68

6 A language for equalities **75**
6.1 Design constraints 75
6.2 Description of *e*ML 76
6.2.1 Extended syntax 76
6.2.2 Extended labeled reduction 77
6.2.3 Combining typing and labeling 77

6.2.4 The non-expansive equality judgment 81
6.2.5 Full term equality 82
6.3 Metatheory of eML 89
6.3.1 Basic properties 89
6.3.2 Extraction 92
6.3.3 Subject reduction 93
6.3.4 Soudness via a logical relation for equality 96
6.3.5 Full term equality and reduction 103

7 Staging with mML 105
7.1 Overview of the design 105
7.2 Definition of mML 106
7.2.1 Syntax and typing 106
7.2.2 The meta reduction 109
7.2.3 Equality 113
7.3 Metatheory of mML 119
7.3.1 Confluence 119
7.3.2 Basic properties of the typing derivation 132
7.3.3 Strong normalization 132
7.3.4 Subject reduction and soundness 138
7.4 mML elimination 146

8 A logical relation for reasoning on mML 155
8.1 A deterministic reduction 155
8.2 Interpretation of kinds 157
8.3 The logical relation 159

9 From eML to ML 167
9.1 Expanded terms 168
9.2 Simplification 168
9.3 Removing equalities 173

III Encoding ornaments 179

10 Encoding ornaments in mML 181
10.1 Ornamentation as a logical relation 181
10.2 Deep pattern matching in eML 183
10.3 Defining datatype ornaments 186
10.4 Shallow ornaments 188
10.5 From high-level definition to low-level definition 191
10.6 Encoding ornaments in mML 193
10.7 Correctness of the encoding 195

11 Elaborating to the generic term 197
11.1 Preparing an ML term for lifting 197
11.2 Elaboration environments 200
11.3 Elaboration 204
11.4 Correctness of elaboration 206
11.5 Identity instantiation 206

12 Lifting by instantiation **209**
12.1 Specifying liftings 209
12.2 Correctness of the lifting 212

IV Implementing lifting **215**

13 Implementation **217**
13.1 Generalization by inference 217
13.2 Strategies for instantiation 220
13.3 Refolding terms after the transformation 222

14 Unfolding of definitions **225**
14.1 Unfolding the recursive structure 225
14.2 Unfolding for specialization 226
14.3 Generic programming with unfolding 227

15 Extensions and future work **231**
15.1 Handling effects 231
15.2 Generalizing the ornamentation relations 232
15.3 Improving the patching language 233
15.4 Scaling up the prototype 233

V Conclusion **235**

16 Related works **237**
16.1 Ornaments 237
16.2 Refactoring 238
16.3 Porting operations to similar datatypes 240

17 Conclusion **243**

List of Figures

3.1 Overview of the lifting process 39

4.1 Types, kinds and environments for ML 50
4.2 Definition of some datatypes 51
4.3 Well-formedness rules 52
4.4 Syntax of ML 53
4.5 Typing rules for ML 54
4.6 Reduction for ML 55

5.1 Syntax of labelled ML 64
5.2 Reduction for labeled ML 65
5.3 Making ML terms reusable 66
5.4 Full head reduction for labeled ML 67
5.5 Contexts for full reduction 67
5.6 Translation 68
5.7 Full reduction 68
5.8 Well-labeling 70

6.1 Types and kinds for eML 77
6.2 Additional head reduction rule for eML 77
6.3 Additional contexts for eML reduction 77
6.4 Typing environments for eML 78
6.5 Typing rules for eML 79
6.6 Well-formedness rules of eML 81
6.7 Type equality 83
6.8 Non-expansive term equality 84
6.9 Non-expansive term equality (congruence rules) 85
6.10 Decomposition of terms 87
6.11 Non-expansive reduction 97
6.12 Environment typing 98
6.13 Interpretation of types 100

7.1 Kinds and types of mML 107
7.2 Syntax of mML 108
7.3 Well-formedness for environments and kind 109
7.4 Kinding rules for mML 110
7.5 Typing rules for mML (eML rules) 111
7.6 Typing rules for mML (new rules) 112

7.7 Kind equality for $m\mathsf{ML}$ . . . 113
7.8 Type equality for $m\mathsf{ML}$: new congruence rules . . . 114
7.9 Term equality for $m\mathsf{ML}$: new congruence rules . . . 115
7.10 Type equality for $m\mathsf{ML}$: subkinding, reduction and split . . . 116
7.11 Term equality for $m\mathsf{ML}$: new reduction rules . . . 116
7.12 Definition of $\longrightarrow_{\#}$ . . . 117
7.13 Label translation for $m\mathsf{ML}$ . . . 117
7.14 Well-labeling for types . . . 120
7.15 Well-labeling for types (cont.) . . . 121
7.16 Well-labeling for types . . . 121
7.17 Well-labeling for kinds . . . 122
7.18 Parallel reduction: congruence for terms . . . 124
7.19 Parallel reductions: congruence for types and kinds . . . 125
7.20 Parallel reductions: reduction rules . . . 126
7.21 Interpretation of kinds as sets of interpretations . . . 133
7.22 Interpretation of kinds and types as sets of types and terms . . . 134
7.23 Normal equalities (except congruence rules) . . . 139
7.24 Normal equalities: congruence rules . . . 140
7.25 Kind depth . . . 141
7.26 Logically normalizable equalities . . . 141
7.27 Logical relation for $m\mathsf{ML}$ elimination: type interpretation . . . 147
7.28 Logical relation for $m\mathsf{ML}$ elimination: kind interpretation . . . 148
7.29 Logical relation for $m\mathsf{ML}$ elimination: constraints on type interpretation . . . 148
7.30 Logical relation for $m\mathsf{ML}$ elimination: constraints on kind interpretation . . . 148
7.31 Logical relation for $m\mathsf{ML}$ elimination: environments . . . 148

8.1 Deterministic reduction $\longrightarrow_d$ for $m\mathsf{ML}$ . . . 156
8.2 Interpretation of kinds . . . 158
8.3 Interpretation of environments . . . 160
8.4 Interpretation of types . . . 161

9.1 Expanded terms, simple terms and types . . . 168
9.2 Binding contexts, expansion contexts . . . 169
9.3 Simplification rules . . . 169

10.1 Ornament types . . . 182
10.2 Projection of ornament types . . . 182
10.3 Well-formedness of ornament types . . . 182
10.4 Deep patterns . . . 183
10.5 Typing for patterns . . . 184
10.6 The matching function . . . 185
10.7 Values matched by a pattern . . . 187
10.8 Missing part of a pattern . . . 194
10.9 Patching a pattern . . . 194

11.1 Environments for ML restricted to top-level polymorphism . . . 198
11.2 Well-formedness for global environments . . . 198
11.3 Typing rules for ML restricted to top-level polymorphism . . . 199

11.4 Processing of top-level definitions 199
11.5 Environments 201
11.6 Projections for ornament types 201
11.7 Environment projections 201
11.8 Well-formedness for elaboration 202
11.9 Elaboration to a generalized term 205
11.10 Elaborating a declaration 205

12.1 Ornament instantiation: checking and projection 210
12.2 Lifting environment 210
12.3 Patch instantiation: grammar, projection and well-formedness . . 211
12.4 Lifting 212

Chapter 1

Introduction

A program is a description in precise terms of the operations a computer should follow to accomplish a particular task. The elementary instructions executed by computers are extremely rudimentary. Sorting a list of integers may require hundreds of instructions; complex programs, such as web browsers, are composed of tens of millions of these elementary instructions. A programming language is a tool of abstraction: it allows programmers to give a higher-level explanation of what the program should do. Compilers and interpreters translate the high-level representations manipulated by programmers into low-level instructions. By freeing the programmer from having to think at the level of the computer, programming languages enable programmers to build more complex programs faster.

Programs are rarely written in one go, without erasing anything, then shipped and never modified again. Even when writing the initial version of a program, the programmer may not have an exact idea of what a program should look like, and may often need to go back and change some definition as they understand the requirements better. After the initial version is written, they may realize that the requirements have changed, requiring more generality in some aspects, and less in some others, and change some part of the program to fit the new requirements. Some of these changes are non-local: while the core of the change may consist in modifying just a few program locations, they require extensive *refactoring* to weave new data and behavior throughout the existing code base. The refactored version of a piece of code may replace the existing code, or both versions of the code may co-exist in the program, to be used in different contexts. These transformations are often repetitive, time-consuming, and uninteresting. Programmers may have a high-level view of the transformation they wish to perform, but have to manually perform every single change required by the refactoring. Our goal is to provide tools making these transformations easier.

If the programmer has been careful in avoiding side-effects, the different parts of their program communicate by exchanging data through function arguments and return values. This data is constrained: the parts of a program communicating together must agree on a format for the data they are exchanging. If the language permits it, some of these constraints may be expressed as *types*: types can be understood as a kind of classification of programs that is accessible to the programming language. Types are a powerful tool: they al-

low the programming language to reject a program at compilation time because its execution may reach an incorrect state. Types also operate as a form of guaranteed-correct documentation to programmers: reading the type of a function gives information to the programmer about how the function ought to be called, and what kind of result may be expected. They allow the programmer to trust that whole classes of behaviors will never occur.

Even though they only characterize the behavior of one version of a program in isolation, types are useful to guide refactoring: when one effect of the refactoring is to change the type of some data, the type system is able to point the programmer to now ill-typed program locations. The programmer makes the necessary changes and runs the type checker again: the type checker will either point to new ill-typed locations, or accept the program, signalling that the refactoring is complete. This is still not a satisfying solution: First, some refactorings do not change the types, but only the way values are interpreted: if the meaning of a boolean is inverted, the type system would be unable to point us to the program location that need fixing and the programmer would have to track this location themselves. More importantly, this is still a boring and manual process: the programmer does not have to find the location to change anymore, but still has to perform all changes manually. These changes, that should often be simply mechanical, are still error-prone. This discourages programmers from attempting such changes: they may instead simply leave a note saying that a refactoring ought be performed.

A principled refactoring tool should eliminate as much as possible of these costs, leaving as little work as possible to the programmer, and providing a strong guarantee that the final program is merely an adaptation of the old one to the new data.

In the ML family of languages, inductive datatypes are the key tool used to describe the organization of data. They are defined as labeled sums and products of primitive types and other inductive datatypes. Products allow for composite values abstracting over sets of multiple values: for example, instead of manipulating pairs of coordinates, a programmer will find more convenient to manipulate two-dimensional points as a unit of programming and thought, and only look at the individual coordinates for implementing basic operations on the points. Sums allow the expression of choice: a shape is either a triangle, defined by three points, or a circle, defined by its center and radius. Finally, recursion allows representing data structures of unbounded size: a list of shapes is either the empty list, or a list formed by adding a first shape in front of another list of shapes.

However, the same data can often be represented with several isomorphic data-structures, using a different arrangement of sums and products. Two data-structures may also differ in minor ways, for instance sharing the same recursive structure, but one carrying an extra information at some specific nodes. For example, the type of leaf binary trees and the type of node binary trees both share a common binary-branching structure and are isomorphic but functions operating on them must be defined independently.

Since inductive datatypes are the main form of representing data in ML, and since distant modules in well-structured programs interact mostly through the exchange of data, the transformation of inductive datatypes is an important category of refactoring in ML.

The theory of ornaments [McBride, 2010, Dagand and McBride, 2013, 2014]

provides a framework to express these changes. It defines conditions under which a new datatype definition can be described as an *ornament* of another. In essence, an ornament is a relation between two datatypes, reorganizing, specializing, and adding data to a bare type to obtain an ornamented type.

From ornaments as building blocks, we define *ornament types* relating a base term and a *lifted* term: an ornament type is not simply a unary specification classifying single terms, but a binary specification explaining how the behavior of the base term, operating on the base type, is related to the lifted term, operating on the ornamented type. These types thus give a specification for a lifting: a correct lifting is a lifting that is related to its base term at the desired ornament type.

We describe a principled approach to obtaining correct liftings. We abstract *a posteriori* a base term into a most general elaborated term, that can then be instantiated into a concrete lifting. This abstraction is not expressible in ML: we define a superset mML of ML that supports the necessary abstraction mechanisms, and guarantees that instantiated terms can be simplified back to terms of the original language. The base term can be seen as an instantiation of the generic term with identity ornaments, transforming a type into itself. We exploit parametricity to prove the expected relation between the base term and the ornamented term.

We implement this transformation on a core ML language with datatypes and polymorphism. Our prototype tool, available online at `https://morphis.me/ornaments/`, implements a number of techniques to generate code that is close to code that would have been manually written, and comes with a collection of examples illustrating its features.

The rest of this dissertation is structured as follows. In Part I, we first give an introduction to ornaments by way of examples (Chapter 2). These examples also serve as a demonstration of our prototype tool, and show a variety of program transformations afforded by ornamentation. We then provide an overview of the ornamentation process, skipping over the most technical details to present the guiding principles of the transformation (Chapter 3).

In Part II, we forget about ornaments for a moment and design and formalize the building blocks of our transformation. Although they were designed for ornamentation, they are independent from ornaments and could be applied to other program refactorings. In Chapter 4, we present the definition and meta-theory of core ML. The next three chapters add the features necessary to present the transformation. Practical ML programs feature side-effects: they might not terminate, terminate with an error, draw random numbers, or read and write files. These impure computations are an obstacle to program transformation since they cannot be freely duplicated, reordered, or removed. We define in Chapter 5 a notion of *labels* allowing to refer to the results of previously performed computations in the terms evaluated subsequently, without duplicating the computations themselves. This is used in Chapter 6 to define eML, a language extending the types of ML with type-level pattern matching. To reduce type-level pattern matching, term-level pattern matching introduces in a branch an equality between the scrutinee and the pattern of the branch. If the scrutinee contains a function application, labels allow us to refer to the result of these applications while leaving them otherwise opaque to eML's reasoning rules. To allow us to define the generic term, we define in Chapter 7 a language mML formed by adding dependent abstractions to eML. These ab-

stractions are carefully isolated by enforcing staging so that they can be fully reduced without evaluating the eML parts of the term: this guarantees that they can be eliminated from the lifted term before presenting the term to the user. This is important, as we want the lifted term to be a term of the original language, that can be used as input to the same compiler. We define in Chapter 8 a binary logical relation on mML. That will give us a framework to build higher-order ornaments from datatype ornaments, and allow us to express the relation between the base type and the ornamented type. Finally, in Chapter 9, we describe how to simplify an eML term back to an ML term, ensuring that our intermediate language do not leak in the generated code.

Having defined the necessary machinery, we come back to ornaments in Part III. This part formalizes the concepts introduced in Chapter 3, and builds up to a correctness proof for lifting. In Chapter 10, we describe how to turn an ornament definition into a datatype ornament compatible with the logical relation, and into the construction and destruction functions required to instantiate the generic term. We define what it means for these functions to be correct with respect to an ornament, and prove that our elaborated definitions are indeed correct. In Chapter 11, we explain how to turn a base term into a generic term that can be instantiated. We prove that the identity instantiation of the generic term is indeed equivalent to the base term. Finally, in Chapter 12, we describe how to instantiate a generic term and compute the resulting lifting specification. We then prove that this lifting specification indeed describes the relation between the base term and the lifted term.

This theoretical work is accompanied by a proof-of-concept implementation that demonstrates the practical usefulness of ornamentation. In Part IV, we start by describing some keys aspects of the implementation (Chapter 13) and some strategies to generate readable code as output, as well as features that help reduce the work involved in specifying liftings. In Chapter 14, we describe unfolding, a way to define liftings that do not follow the recursive structure of the original function. We give examples of interesting manipulations of data structures that are allowed by unfolding. In Chapter 15, we explore some future work around possible extensions to the theory and our implementation, notably how one would adapt lifting to work on a real-world language such as OCaml.

Finally, in Part V, we describe some related work (Chapter 16) before concluding (Chapter 17).

Part I

Ornaments in practice

Chapter 2

Ornaments by example

Let us discover ornaments by means of examples. All examples preceded by a blue vertical bar have been processed by a prototype implementation[1], which follows an OCaml-like[2] syntax. Output of the prototype appears with a wider green vertical bar. The code that appears without a vertical mark is internal intermediate code for sake of explanation and has not been processed.

2.1 Code refactoring

The most striking application of ornaments is the special case of code refactoring, which is an often annoying but necessary task when programming. We start with an example reorganizing a sum data structure into a sum of sums. Consider the following datatype representing arithmetic expressions, together with an evaluation function.

```
type expr =
  | Const of int
  | Add of expr * expr
  | Mul of expr * expr
```

```
let rec eval a = match a with
  | Const i     → i
  | Add (u, v) → add (eval u) (eval v)
  | Mul (u, v) → mul (eval u) (eval v)
```

The programmer may realize that the binary operators Add and Mul can be factored, and thus prefer the following version expr' using an auxiliary type of binary operators, given below.

```
type binop = Add' | Mul'
type expr' =
  | Const' of int
  | Binop' of binop * expr' * expr'
```

There is a relation between these two types, which we may describe as an *ornament* oexpr from the base type expr to the ornamented type expr'.

[1] The prototype, available at url `https://morphis.me/ornaments/`, contains a library of detailed examples, including those presented here.

[2] `http://caml.inria.fr/`

```
type ornament oexpr : expr ⇒ expr' with
  | Const i     ⇒ Const' i
  | Add (u, v) ⇒ Binop' (Add', u, v)  when u v : oexpr
  | Mul (u, v) ⇒ Binop' (Mul', u, v)  when u v : oexpr
```

This definition is to be understood as

```
type ornament oexpr : expr ⇒ expr' with
  | Const i₋      ⇒ Const' i₊  when i₋ ⇒ i₊ in int
  | Add (u₋, v₋) ⇒ Binop' (Add', u₊, v₊)
      when u₋ ⇒ u₊ and v₋ ⇒ v₊ in oexpr
  | Mul (u₋, v₋) ⇒ Binop' (Mul', u₊, v₊)
      when u₋ ⇒ u₊ and v₋ ⇒ v₊ in oexpr
```

This recursively defines the oexpr relation. A clause "$x_- \Rightarrow x_+$ **in** orn" means that x_- and x_+ should be related by the ornament relation orn. The first clause is the base case. By default, the absence of ornament specification for variable i (of type int) has been expanded to "**when** $i_- \Rightarrow i_+$ **in** int" and means that i_- and i_+ should be related by the identity ornament at type int, which is also named int for convenience. The next clause is an inductive case: it means that Add(u_-, v_-) and Binop'(Add',u_+, v_+) are in the oexpr relation whenever u_- and u_+ on the one hand and v_- and v_+ on the other hand are already in the oexpr relation.

In this example, the relation happens to be an isomorphism and we say that the ornament is a *pure* refactoring. Hence, the compiler has enough information to automatically lift the old version of the code to the new version. We simply request this lifting as follows:

```
let eval' = lifting eval : oexpr → _
```

The expression oexpr → _ is an *ornament signature*, which follows the syntax of types but replacing type constructors by ornaments. (The wildcard represents a part of the ornament specification that is inferred; it could have been replaced by int, the ornament specification corresponding to the identity ornament on the type int.) Here, the compiler will automatically elaborate eval' to the expected code, without any further user interaction:

```
let rec eval' a = match a with
  | Const' i → i
  | Binop' (Add', u, v) → add (eval' u) (eval' v)
  | Binop' (Mul', u, v) → mul (eval' u) (eval' v)
```

Not only is this well-typed, but the semantics is also preserved—by construction. Notice that a pure refactoring also works in the other direction: we could have instead started with the definition of eval', defined the reverse ornament from expr' to expr, and obtained eval as a lifting of eval'.

Pure refactorings such as oexpr are a particular but quite interesting subcase of ornaments because the lifting process is fully automated. As a tool built upon ornamentation, we provide a shortcut for refactoring: one only has to write the definitions of expr' and oexpr, and lifting declarations are generated to transform a whole source file. Thus, pure refactoring is already a very useful application of ornaments: these transformations become almost free, even on a large code base.

Besides, proper ornaments as described next that decorate an existing node with new pieces of information can often be decomposed into a possibly complex but pure refactoring and another proper, but hopefully simpler ornament. Notice that pure code refactoring need not even define a new type. One such example is to invert values of a boolean type:

```
type bool = True | False
type ornament not : bool ⇒ bool with
    | True ⇒ False
    | False ⇒ True
```

Then, we may define bool_or as a lifting of bool_and, and the compiler inverts the constructors.

```
let bool_and u v = match u with True → v | False → False
let bool_or = lifting band : not → not → not
```

```
let bool_or u v = match u with
    | True → True
    | False → v
```

It may also do this selectively, only at some given occurrences of the bool type. For example, we may only invert the first argument:

```
let bool_notand = lifting bool_and : not → bool → bool
```

```
let bool_notand u v = match u with
    | True → False
    | False → v
```

Still, the compiler will carefully reject inconsistencies, such as:

```
let bool_andnot = lifting bool_and : bool → not → bool
```

Indeed, given the structure of the program, the second argument can be returned untransformed. Lifting preserves this structure: then coherent liftings must use the same ornaments for the second argument and the result.

2.2 Code refinement

Code refinement is an example of a proper ornament where the intention is to *derive* new code from existing code, rather than *modify* existing code and forget the original version afterwards. To illustrate code refinement, observe that lists can be considered as an ornament of Peano numbers:

```
type nat = Z | S of nat

type 'a list = Nil | Cons of 'a * 'a list
type ornament 'a natlist : nat ⇒ 'a list with
  | Z    ⇒  Nil
  | S m  ⇒  Cons (_, m)  when m : 'a natlist
```

The parametrized ornamentation relation 'a natlist is not an isomorphism: a natural number $\mathsf{S}\ \mathsf{m}_-$ will be in relation with all values of the form $\mathsf{Cons}\ (\mathsf{x}, \mathsf{m}_+)$ for any x, as long as m_- is in relation with m_+. We use an underscore "_" instead of x on Cons (_, m) to emphasize that it does not appear on the left-hand side and thus freely ranges over values of its type. Hence, the mapping from nat to 'a list is incompletely determined: we need additional information to translate a successor node. (Here, the ornament definition may also be read in the reverse direction, which defines a projection from 'a list to nat, the length function! but we do not use this information hereafter.)

The addition on numbers may have been defined as follows (on the left-hand side):

```
let rec add m n = match m with
  | Z → n
  | S m' → S (add m' n)
val add : nat → nat → nat
```

```
let rec append m n = match m with
  | Nil → n
  | Cons (x, m') → Cons(x, append m' n)
val append : 'a list → 'a list → 'a list
```

Observe the similarity with append, given above (on the right-hand side). Having already recognized an ornament between nat and list, we expect append to be definable as a lifting of add.

```
let append₀ = lifting add : _ natlist → _ natlist → _ natlist
```

However, this returns an incomplete lifting:

```
let rec append₀ m n = match m with
    | Nil → n
    | Cons (x, m') → Cons (#2, append₀ m' n)
```

Indeed, this requires building a cons node from a successor node, which is underdetermined. This is reported to the user by leaving a labeled hole #2 in the generated code. The programmer may use this label to provide a *patch* that will fill this hole. The patch may use all bindings that were already in context at the same location in the bare version. In particular, the first argument of Cons cannot be obtained directly, but only by matching on m again:

```
let append = lifting add : _ natlist → _ natlist → _ natlist
  with #2 ← match m with Cons(x, _) → x
```

The lifting is now complete, and produces exactly the code of append given above. The superfluous pattern matching in the patch has been automatically removed: the patch "**match** m **with** Cons(x_0,_) → x_0" has not just been inserted in the hole, but also simplified by observing that x_0 is actually equal to x and need not be extracted again from m. This also removes an incomplete pattern matching. This simplification process relies on the ability of the meta-language to maintain equalities between terms via dependent types (§6), and is needed to make the lifted code as close as possible to manually written code. This is essential, since the lifted code may become the next version of the source code to be read and modified by the programmer. This is a strong argument in favor of the principled approach that we present next and formalize in the rest of the manuscript.

Although the hole cannot be uniquely determined by ornamentation alone, it is here the obvious choice: since the append function is polymorphic we need an element of the same type as the unnamed argument of Cons, so this is the obvious value to pick—but not the only one, as one could also look further in the tail of the list. Instead of giving an explicit patch, we could give a tactic that would fill in the hole with the "obvious choice" in such cases. However, while important in practice, this is an orthogonal issue related to code inference which is not the focus of this work. Below, we stick to the case where patches are always explicitly determined and we leave holes in the skeleton when patches are missing.

This example is chosen here for pedagogical purposes, as it illustrates the key ideas of ornamentation. While it may seem anecdotal, there is a strong relation between recursive data structures and numerical representations, whose relation to ornamentation has been considered by Ko [2014].

2.3 Composing transformations: a practical use case

Ornamentation could be used in different scenarios: the intent of *refactoring* is to replace the base code with the generated code, even though the base code could also be kept for archival purposes; when *enriching* a data structure, both codes may coexist in the same program. To support both of these usages, we try to generate code that is close to manually written code. For other uses, the base code and the lifting instructions may be kept to regenerate the lifted code when the base code changes. This already works well in the absence of patches; otherwise, we would need a patch description language that is more robust to changes in the base code. We could also postprocess ornamentation with some simple form of code inference that would automatically try to fill the holes with "obvious" patches, as illustrated below. Our tool currently works in batch mode and is just providing the building blocks for ornamentation. The ability to output the result of a partially specified lifting makes it possible to build an interactive tool on top of our interface.

The following example shows how different use-cases of ornaments can be composed to reorganize, enrich, and cleanup an incorrect program, so that the final bug fix can be reduced to a manual but simple step. The underlying idea is to reduce manual transformations by using automatic program transformations whenever possible. Notice that since lifting preserves the behavior of the original program, fixing a bug cannot just be done by ornamentation.

Let us consider a small calculus with abstractions and applications and tuples (which we will take unary for conciseness) and projections. We assume given a type id representing variables.

```
type expr =
    | Abs of id * expr
    | App of expr * expr
    | Var of id
    | Tup of expr
    | Proj of expr
```

We write an expression evaluator using environments of type (id * expr) list . We assume given an assoc function of type 'a → ('a * 'b) list → 'b option that searches a binding in the environment.

```
let rec eval env e =
  match e with
  | Var x → assoc x env
  | Abs(x, f) → Some (Abs(x, f))
  | App(e1,e2) →
    begin match eval env e1 with
     | Some (Abs(x,f)) →
        begin match eval env e2 with
        | Some v → eval (Cons((x,v), env)) f
        | None → None
        end
     | Some (Tup _) → None      (* Type error        *)
     | None → None              (* Error propagation *)
     | Some _ → fail ()         (* Not a value ?!    *)
    end
```

```
  | Tup(e) →
    begin match eval env e with
    | Some v → Some (Tup v)
    | None → None
    end
  | Proj(e) →
    begin match eval env e with
    | Some (Tup v) → Some v
    | Some (Abs _) → None      (* Type error        *)
    | None → None              (* Error propagation *)
    | Some _ → fail ()         (* Not a value ?!    *)
    end
(* eval : (id * expr) list -> expr -> expr option *)
```

The evaluator distinguishes type (or scope) errors in the program, where it returns None, and internal errors when the expression returned by the evaluator is not a value. In this case, the evaluator raises an exception by calling fail ().

We soon realize that we mistakenly implemented dynamic scoping: the result of evaluating an abstraction should not be an abstraction but a closure that holds the lexical environment of the abstraction. One path to fixing this evaluator is to start by separating the subset of *values* returned by the evaluator from general expressions. We define a type of values as an ornament of expressions.

```
type value =
  | VAbs of id * expr
  | VTup of value

type ornament expr_value : expr ⇒ value with
  | Abs(x, e) ⇒ VAbs(x, e) when e : expr
  | Tup(e)    ⇒ VTup(e) when e : expr_value
  | _ → ~
```

This ornament is intendedly *partial*: some cases are not lifted. Indeed, ornaments define a relation between the bare type and the ornamented type. They are defined syntactically, with both sides being linear patterns. Moreover, the pattern for the ornamented type should be total, i.e. match all expressions of the ornamented type. Conversely, the pattern of the bare type need not be total, but it must not throw away any information: in particular, it cannot use wildcards or alternative patterns. When lifting a pattern matching with a partial ornament, the inaccessible cases will be dropped. On the other hand, when constructing a value that is impossible in the lifted type, the user will be asked to construct a patch of the empty type, which could be filled for example by an assertion failure. In some cases, the holes that should be filled with values of the empty type will disappear during simplification, guaranteeing that the program will not attempt to construct such values. The notation ~ corresponds to the empty pattern.

This ornament does not preserve the recursive structure of the original datatype: the recursive occurrences are transformed into values or expressions depending on their position. By contrast with prior works [Williams et al., 2014, Dagand and McBride, 2013, 2014], we do not treat recursion specifically. Hence, mutual recursion is not a problem; for instance, we can ornament a mutually recursive definition of trees and forests or modify the recursive structure during ornamentation. We also allow changing the recursive structure of func-

tions, transforming a single recursive function into a group of mutually recursive functions (Chapter 14).

Using the ornament expr_value we transform the evaluator by making explicit the fact that it only returns values and that the environment only contains values (as long as this is initially true):

```
let eval' = lifting eval : (id * expr_value) list → expr → expr_value option
(* val eval' : (id * value) list -> expr -> value option *)
```

The lifting succeeds—and eliminates all occurrences of fail () in eval'.

```
let rec eval' env e = match e with
    | Var x → assoc x env
    | Abs(x, e) → Some (VAbs(x, e))
    | App(e1, e2) →
      begin match eval' env e1 with
        | (None | Some (VTup _)) → None
        | Some (VAbs(x, e)) →
          begin match eval' env e2 with
            | None → None
            | Some v → eval' (Cons((x, v), env)) e
          end
      end
    | Tup e →
      begin match eval' env e with
        | None → None
        | Some e → Some (VTup e)
      end
    | Proj e →
      begin match eval' env e with
        | (None | Some (VAbs(_, _))) → None
        | Some (VTup e) → Some e
      end
```

We may now refine the code to add a field for storing the environment in closures:

```
type value' =
  | VClos' of id * (id * value') list * expr
  | VTup' of value'

type ornament value_value' : value ⇒ value' with
  | VAbs(x, e) ⇒ VClos'(x, _, e)
  | VTup(v)    ⇒ VTup'(v) when v : value_value'
```

Since this ornament is not one-to-one, the lifting of eval' is partial. The advanced user may realize that there should be a single hole in the lifted code that should be filled with the current environment env, and may directly write the clause "| * ← env":

```
let eval'' = lifting eval' with ornament * ← value_value', @id | * ← env
```

The annotation **ornament** * ← value_value', @id is another way to indicate which ornaments to use. This is sometimes more convenient than giving a signature. For each type that needs to be ornamented, we first try value_value', and use the identity ornament if this fails (*e.g.* on types other than value). A more pedestrian path to writing the patch is to first look at the output of the partial lifting:

```
let eval'' = lifting eval' with ornament * ← value_value', @id
```

```
let rec eval'' env e = match e with
  | Abs(x, e) → Some (VClos'(x, #32, e))
  | App (e1, e2) → ... |  ...
```

The hole has been labeled #32 which can then be used to refer to this specific program point. The patch associated to this label can access all variables in scope where the label #32 appears in the partially lifted term.

```
let eval'' = lifting eval' with ornament ∗ ← value_value', @id | #32 ← env
```

An interactive tool could point the user to this hole in the partially lifted code shown above, so that they directly enters the code env, and the tool would automatically generate the lifting command just above. Notice that env is the most obvious way to fill the hole here, because it is the only variable of the expected type available in context. Hence, a very simple form of type-based code inference could pre-fill the hole with env and just ask the user to confirm.

When the programmer is quite confident, they could even ask for this to be done in batch mode:

```
let eval'' = lifting eval' with
    | ornament ∗ ← value_value', @id
    | ∗ ← try by type
```

Example-based code inference would be another interesting extension of our prototype, which would increase the robustness of patches to program changes. Here, the user could instead write:

```
let eval'' = lifting eval' with
  | ornament ∗ ← value_value', @id
  | ∗ ← try eval env (VAbs (_, _)) = Some (Closure (_, env, _))
```

providing a partial definition of eval that is sufficient to completely determine the patch.

For each of these possible specifications, the system will return the same answer:

```
let rec eval'' env e = match e with
  | Abs(x, e) → Some (VClos'(x, env, e))
  | App(e1, e2) → bind' (eval'' env e1) (function
     | VClos'(x, _, e) → bind' (eval'' env e2) (fun v → eval'' (Cons((x, v), env)) e)
     | VTup' _ → None)
  | Var x → assoc x env
  | ...
```

So far, we have not changed the behavior of the evaluator: the ornaments guarantee that the result of eval'' on some expression is essentially the same as the result of eval—up to the addition of an environment in closures. The final modification must be performed manually: when applying functions, we need to use the environment of the closure instead of the current environment.

```
let rec eval'' env e = match e with
  | Var x → assoc x env
  | Abs(x, e) → Some (VClos'(x, env, e))
  | App(e1, e2) →
    begin match eval'' env e1 with
      | (None | Some (VTup' _)) → None
      | Some (VClos'(x, closure_env, e)) →
        begin match eval'' env e2 with
```

```
        | None → None
        | Some v → eval'' (Cons((x, v), closure_env (* was env *) )) e
      end
    end
  | Tup e →
    begin match eval'' env e with
      | None → None
      | Some v → Some (VTup' v)
    end
  | Proj e →
    begin match eval'' env e with
      | (None | Some (VClos'(_, _, _))) → None
      | Some (VTup' v) → Some v
    end
```

2.4 Hiding administrative data

Sometimes data structures need to carry annotations, which are useful information for certain purposes but not at the core of the algorithms. A typical example is location information attached to abstract syntax trees for error reporting purposes. The problem with data structure annotations is that they often obfuscate the code. We show how ornaments can be used to keep programming on the bare view of the data structures and lift the code to the ornamented view with annotations. In particular, scanning algorithms can be manually written on the bare structure and automatically lifted to the ornamented structure with only a few patches to describe how locations must be used for error reporting.

Consider for example, the type of λ-expressions and its call-by-name evaluator:

```
type 'a option =
  | None
  | Some of 'a
type expr =
  | Abs of (expr → expr)
  | App of expr * expr
  | Const of int
```

```
let rec eval e = match e with
  | App (u, v) →
      (match eval u with Some (Abs f) → Some (f v)
      | _ → None)
  | v → Some (v)
```

The datatype expr' that holds location information can be presented as an ornament of expr:

```
type loc = Location of string * int * int
type expr' =
  | App' of (expr' * loc) * (expr' * loc)
  | Abs' of (expr' * loc → expr' * loc)
  | Const' of int
```

```
type ornament add_loc : expr ⇒ expr' ∗ loc with
  | Abs f ⇒ (Abs' f, _) when f : add_loc → add_loc
  | App (u, v) ⇒ (App' (u, v), _) when u v : add_loc
  | Const i ⇒ (Const' i, _)
```

The datatype for returning results is an ornament of the option type:

```
type ('a, 'err) result =
  | Ok of 'a
  | Error of 'err
```

```
type ornament ('a, 'err) optres : 'a option ⇒ ('a, 'err) result with
  | Some a ⇒ Ok a
  | None ⇒ Error _
```

If we try to lift the function without further information,

```
let eval_incomplete = lifting eval : add_loc → (add_loc, loc) optres
```

the system will only be able to do a partial lifting, unsurprisingly:

```
let rec eval_incomplete e = match e with
    | (App'(u, v), x) →
      begin match (∗ _2 ∗) eval_incomplete u with
        | Ok (Abs' f, x) → Ok (f v)
        | ((Ok (App'(_, _, _)) | Ok (Const'(_, _)))
          | Error _) → Error #4
      end
    | (Abs' f, x) → Ok e
    | (Const' i, x) → Ok e
```

Indeed, in the erroneous case, eval' must now return a value of the form Error (...) instead of None, but it has no way of knowing which arguments to pass to the constructor, hence the hole labeled #4.

To complete the lifting, we provide the following patch, using the auxiliary identifier _2 to refer to the inner match expression. This auxiliary identifier is defined for convenience, and is indicated as a comment next to the corresponding term when printing partial liftings, as in the code above.

```
let eval_loc = lifting eval : add_loc → (add_loc, loc) optres with
  | #4 ← begin match _2 with Error err → err
           | Ok _ → (match e with (_, loc) → loc) end
```

We then obtain the expected complete code. Notice that the pattern matching on all error cases has been expanded to match the different branches of the patch:

```
let rec eval_loc e = match e with
    | (App'(u, v), loc) →
      begin match eval_loc u with
        | Ok (Abs' f, x) → Ok (f v)
        | (Ok (App'(_, _, _)) | Ok (Const'(_, _))) →
          Error loc
        | Error err → Error err
      end
    | (Abs' f, loc) → Ok e
    | (Const' i, loc) → Ok e
```

Common branches could actually be refactored using wildcard abbreviations whenever possible, leading to the following code, but this has not been implemented yet:

```
let rec eval_loc e → match e with
  | App' (u, v), loc →
      begin match eval_loc u with
        | Ok (Abs' f, loc) → Ok (f v)
        | Ok (_, _) → Error loc
        | Error err → Error err
      end
  | _ → Ok e
```

While this example is limited to the simple case where we only read the abstract syntax tree, some compilation passes often need to transform the abstract syntax tree carrying location information around. More experiment is still needed to see how the ornament approach scales up here to more complex transformations. This might be a case where appropriate tactics for filling the holes could be helpful.

This example suggests a new use of ornaments in a programming environment where the bare code and the lifted code will be kept in sync, and the user will be able to switch between the two views, using the bare code for the core of the algorithm that need not see the decorations and the lifted code only when necessary.

2.5 Higher-order types, recursive types

Lifting also works with higher-order types and recursive datatype definitions with negative occurrences. For example, we could extend arithmetic expressions with nodes for abstraction and application, with functions represented by functions of the host language:

```
type expr =
  | Const of int
  | Add of expr * expr
  | Mul of expr * expr
  | Abs of (expr → expr option)
  | App of expr * expr
```

Then, the evaluation function is partial:

```
let rec eval e = match e with
  | Const i → Some(Const i)
  | Add ( u , v ) →
      begin match (eval u, eval v) with
        | (Some (Const i1), Some (Const i2)) → Some(Const (add i1 i2))
        | _ → None
      end
  | Mul ( u , v ) →
      begin match (eval u, eval v) with
        | (Some (Const i1), Some (Const i2)) → Some(Const (mul i1 i2))
        | _ → None
      end
  | Abs f → Some(Abs f)
  | App(u, v) →
```

```
    begin match eval u with
      | Some(Abs f) →
        begin match eval v with None → None | Some x → f x end
      | _ → None
    end
val eval : expr → expr option
```

We could still prefer the following representation factoring the arithmetic operations:

```
type binop' =
  | Add'
  | Mul'

type expr' =
  | Const' of int
  | Binop' of binop' ∗ expr' ∗ expr'
  | Abs' of (expr' → expr' option)
  | App' of expr' ∗ expr'
```

Then, we can define an ornament between these types, despite expr' recursively occurring to the left of a function arrow in its definition:

```
type ornament oexpr : expr ⇒ expr' with
  | Const ( i ) ⇒ Const' ( i )
  | Add ( u , v ) ⇒ Binop' ( Add' , u , v ) when u v : oexpr
  | Mul ( u , v ) ⇒ Binop' ( Mul' , u , v ) when u v : oexpr
  | Abs f ⇒ Abs' f when f : oexpr → oexpr option
  | App ( u , v ) ⇒ App' ( u , v ) when u v : oexpr
```

In the clause of Abs, the lifting of the argument is specified by an higher-order ornament type oexpr → oexpr option that recursively uses oexpr as argument of another type, and on the left of an arrow. We can then use this to lift the function eval:

```
let eval' = lifting eval : oexpr → oexpr option
  with ornament ∗ ← @id
val eval' : expr' → expr' option
```

The annotation **ornament** ∗ ← @id indicates that, for all ornaments that are not otherwise constrained, the identity ornament should be used by default. This is necessary because we create and destruct a tuple in eval, but the type of the tuple does not appear in the signature, so we cannot specify the ornament that should be used through the signature. We give more details in Chapter 12.

Chapter 3

Overview of the lifting process

Whether used for refactoring or refinement, ornaments are about code reuse. Code reuse is usually obtained by modularity, which itself relies on both type and value abstraction mechanisms. Typically, one writes a generic function *gen* that abstracts over the representation details, say described by some structures $\bar{s}$ of operations on types $\bar{\tau}$. Hence, a concrete implementation a is schematically obtained by the application $gen\ \bar{\tau}\ \bar{s}$; changing the representation to small variation $\bar{s}'$ of types $\bar{\tau}'$ of the structures $\bar{s}$, we immediately obtain a new implementation $gen\ \bar{\tau}'\ \bar{s}'$, say a'.

Although the case of ornamentation seems quite different, as we start with a non-modular implementation a, we may still get inspiration from the previous schema: modularity through abstraction and polymorphism is the essence of good programming discipline. Instead of directly going from a to a' on some ad hoc track, we may first find a modular presentation of a as an application $a_{gen}\ \bar{\tau}\ \bar{s}$ so that moving from a to a' is just finding the right parameters $\bar{\tau}'$ and $\bar{s}'$ to pass to a_{gen}.

This is depicted in Figure 3.1. In our case, the *elaboration* that finds the generic term a_{gen} is syntactic and only depends on the source term a. Hence, the same generic term a_{gen} may be used for different liftings of the same source code. The *specialization* process is actually performed in several steps, as we do not want a' to be just the application $a_{gen}\ \bar{\tau}'\ \bar{s}'$, but be presented in a simplified form as close as possible to the term we started with and as similar as possible to the code the programmer would have manually written. Hence, after instan-

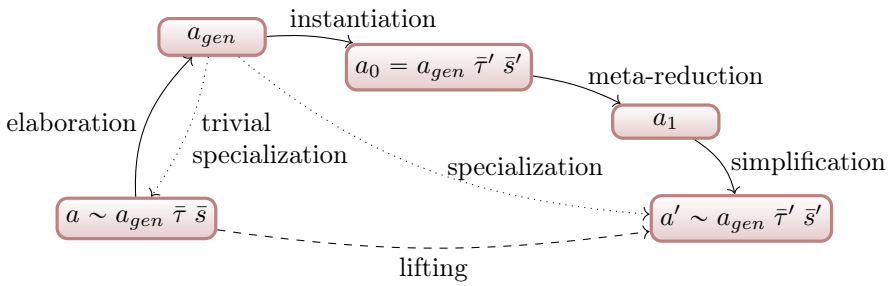

Figure 3.1: Overview of the lifting process

tiation, we perform *meta-reduction*, which eliminates all the abstractions that have been introduced during the elaboration—but not others. This is followed by *simplifications* that will mainly eliminate intermediate pattern matchings.

Having recovered a modular schema, we may use parametricity results, based on logical relations. As long as the arguments s and s' passed to the polymorphic function a_{gen} are related—and they are by the ornamentation relation!—the two applications $a_{gen}\ \bar{\tau}\ \bar{s}$ and $a_{gen}\ \bar{\tau}'\ \bar{s}'$ are also related. Since meta-reduction preserves the relation, it only remains to check that the simplification steps also preserve equivalence to establish a relationship between the bare term a and the lifted term a' (Chapter 7 and 9).

The lifting process is formally described in Chapter 12. In the rest of this section, we present it informally on the example of add and append.

3.1 Encoding ornaments

Ornamentation only affects datatypes, so a program can be lifted by simply inserting some code to translate from and to the ornamented type at occurrences where the base datatype is either constructed or destructed in the original program.

We now explain how this code can be automatically inserted. For sake of illustration, we proceed in several incremental steps. Intuitively, the append function should have the same structure as add, and operate on constructors Nil and Cons similarly to the way add proceeds with constructors S and Z.

To help with this transformation, we may see a list as a nat-like structure where just the head of the list has been transformed. For that purpose, we introduce an hybrid open version of the datatype of Peano naturals, called the *skeleton*, using new constructors Z' and S' corresponding to Z and S but remaining parameterized over the type of the argument of the constructor S:

```
type 'a nat_skel = Z' | S' of 'a
```

We define the head projection of a list into nat_skel[1] where the head is transformed and the tail stays a list:

```
let proj_nat_list : 'a list → 'a list nat_skel = fun m ⇒# match m with
  | Nil → Z'
  | Cons (_, m') → S' m'
```

We use annotated versions of abstractions **fun** x ⇒# a and applications $a\#b$ called *meta-abstractions* and *meta-applications* to keep track of helper code and distinguish it from the original code, but these can otherwise be read as regular functions and applications.

Once an 'a list has been turned into 'a list nat_skel, we can pattern match on it in the same way we match on nat in the definition of add. Hence, the definition of append should look like:

```
let rec append₁ m n = match proj_nat_list # m with
  | Z' → n
  | S' m' → ... S' (append₁ m' n) ...
```

[1] Our naming convention is to use the suffix _nat_list for the functions related to the ornament from nat to list .

In the second branch, we must construct a list out of the nat skeleton containing a list S' (append$_1$ m' n). We use a helper function to inject an 'a list nat_skel into an 'a list :

```
| S' m' → inj_nat_list₁ (S'(append m' n)) ...
```

Of course, inj_nat_list requires some supplementary information x to put in the head of the list:

```
let inj_nat_list : 'a list nat_skel → 'a → 'a list =
  fun n x ⇒ match n with
    | Z' → Nil
    | S' n' → Cons (x, n')
```

As explained earlier (§2.2), we need the patch (**match** m **with** Cons (x, _) → x) to obtain append, and it must be user provided as patch #2. Hence, the lifting of add into lists is:

```
let rec append₂ m n = match proj_nat_list # m with
  | Z' → n
  | S' m' →
      inj_nat_list # (S'(append₂ m' n)) # (match m with Cons (x, _) → x)
```

This version is correct, but not final yet, as it still contains the intermediate hybrid structure, which will eventually be eliminated.

However, before we see how to do so in the next section, we first check that our schema extends to more complex examples of ornaments. Assume, for instance, that we also attach new information to the Z constructor to get lists with some information at the end, which could be defined as:

```
type ('a,'b) listend = Nilend of 'b | Consend of 'a * ('a, 'b) listend
```

We may write encoding and decoding functions as above:

```
let proj_nat_listend = fun l → match l with
    | Nilend _ → Z'
    | Consend (_,l') → S' l'

let inj_nat_listend = fun n x → match n with
    | Z' → Nilend x
    | S' l' → Consend (x,l')
```

However, a new problem appears: we cannot give a valid ML type to the function inj_nat_listend, as the argument x should take different types depending on whether n is zero or a successor. This is solved by adding a form of dependent types to our intermediate language—and finely tuned restrictions to guarantee that the generated code becomes typeable in ML after some simplifications. The general idea of the simplifications is given next.

3.1.1 Eliminating the encoding

The mechanical ornamentation both creates intermediate hybrid data structures and includes extra abstractions and applications. Fortunately, these additional computations can be avoided, which not only removes sources of inefficiencies, but also helps generate code with fewer indirections that is more similar to hand-written code.

We first perform meta-reduction of append$_2$, which removes all helper functions (we actually give different types to ordinary and meta functions so that

meta functions can only be applied using meta-applications and ordinary functions can only be applied using ordinary applications):

```
let rec append3 m n =
  match (match m with Nil → Z' | Cons (x, m') → S' m') with
    | Z' → n
    | S' m' →
        match S'(append3 m' n) with
          | Z' -> Nil
          | S' r' → Cons ((match m with Cons(x, _) → x), r')
```

(The grayed out branch is inaccessible). Still, append_3 computes two pattern matchings that do not appear in the manually written version append. Interestingly, both of them can be eliminated. Extruding the inner match on m in append_3, we get:

```
let rec append4 m n = match m with
  | Nil → (match Z' with Z' → n | S' m' -> b)
  | Cons (x, m') → (match S' m' with Z' -> n | S' m' → b)
```

Since we know that m is equal to Cons(x,m') in the Cons branch, we simplify b to Cons(x, append m' n). After removing all remaining dead branches, we exactly obtain the manually written version append:

```
let rec append = fun m n →
    match m with
      | Nil → n
      | Cons (x, m') → Cons (x, append m' n)
```

3.1.2 Inferring a generic lifting

We have shown a specific ornamentation append of add. However, instead of producing such an ornamentation directly, we first generate a generic lifting of add abstracted over all possible instantiations, and only then specialize it to some specific ornamentation by passing encoding and decoding functions as arguments, as well as a set of *patches* that generate the additional data.

Let us detail this process by building the generic lifting add_gen of add, which we repeat below.

```
let rec add = fun m n → match m with
    | Z → n
    | S m' → S(add m' n)
```

Because they will be passed together to the function, we group the injection and projection into a record:

```
type ('a,'b,'c) orn = { inj : 'a → 'b → 'c; proj : 'c → 'a }
let nat_list = { inj = inj_nat_list; proj = proj_nat_list; }
```

The code of append_2 could have been written as:

```
let rec append2 m n = match nat_list.proj # m with
  | Z' → n
  | S' m' → nat_list.inj # (S'(add m' n)) # (match m with Cons (x, _) → x)
  in append2
```

Instead of using the concrete ornament nat_list, the generic version abstracts over arbitrary ornaments of nats and over the patch:

```
let add_gen = fun m_orn n_orn p1 ⇒
  let rec add_gen' m n = match m_orn.proj # m with
    | Z' → n
    | S' m' → n_orn.inj # S'(add_gen' m' n) # (p1 # add_gen' # m # m' # n)
  in add_gen'
```

While append_2 uses the same ornament nat_list for ornamenting both arguments m and n, this need not be the case in general; hence add_gen has two different ornament arguments m_orn and n_orn. The patch p_1 is abstracted over all variables in scope, *i.e.* m, n and m'.

In general, we ask for a different ornament for each occurrence of a constructor or pattern matching on a datatype. We then apply ML inference on the generic term (ignoring the patches) allowing us to deduce that some ornaments encode to the same datatype. In order to preserve the relation between the bare and lifted terms (see Chapter 11), these ornaments are merged into a single ornament, with a single record. We thus obtain a description of all possible *syntactic* ornaments of the base function, *i.e.* those ornaments that preserve the structure of the original code. The patch p_1 describes how to obtain the missing information from the environment (namely add_gen, m, n, m') when building a value of the ornamented type. While the parameters m_orn and n_orn will be automatically instantiated, the code for patches will have to be user-provided.

The generalized function abstracts over all possible ornaments, and must now be instantiated with some specific ornaments. We may for instance decide to ornament nothing, *i.e.* just lift nat to itself using the *identity ornament* on nat, which amounts to passing to add_gen the following trivial functions:

```
let proj_nat_nat = fun x ⇒                  let inj_nat_nat = fun x () ⇒
  match x with Z → Z' | S x → S' x            match x with Z' → Z | S' x → S x
let orn_nat_nat = { proj=proj_nat_nat; inj = inj_nat_nat }
```

There is no information added, so we may use the following unit_patch for p_1:

```
let unit_patch = fun _ _ _ _ ⇒ ()
let add1 = add_gen # orn_nat_nat # orn_nat_nat # unit_patch
```

As expected, meta-reducing add_1 and simplifying the result returns the original program add.

We may also instantiate the generic lifting with the ornament from nat to lists and the following patch. Meta-reduction of append_5 gives append_2 which can then be simplified to append, as explained above.

```
let orn_nat_list = { proj = proj_nat_list; inj = inj_nat_list }
let append_patch = fun _ m _ _ ⇒ match m with Cons(x, _) → x
let append5 = add_gen # orn_nat_list # orn_nat_list # append_patch
```

The generic lifting is not exposed as is to the user because it is not convenient to use directly. Positional arguments are not practical, because one must reference the generic term to understand the role of each argument. We can solve this problem by attaching the arguments to program locations and exposing the correspondence in the user interface. For example, in the lifting of add to append shown in the previous section, the location #2 corresponds to the argument p_1.

The next two parts formalize the intuition given in this chapter: in Part II, we define a layer over ML allowing to express the encoding of ornaments and reasoning tools that will be used to prove that ornamentation is correct; in

Part III, we exploit this language to define the encoding and prove that lifting is correct.

Part II

A calculus for program transformation

In this part, we build a meta-language on top of ML as a tool to define and reason about ornamentation. We start by formally defining our subset of OCaml (Chapter 4). We then define a *labeling* of function applications to reuse the results of previously computed applications (Chapter 5). We build upon this concept to present eML, an extension of ML with equalities and type-level pattern matching (Chapter 6). On top of eML, we add another layer to represent *meta-abstractions* and *meta-applications* as well as richer type-level computation (Chapter 7). This layer is carefully designed so that it can disappear simply by reduction (Theorem 7.3). We present a logical relation allowing us to reason on mML terms (Chapter 8); later, we will encode ornaments into this relation. Finally, we show that eML terms can be simplified to ML in appropriate context (Chapter 9), preserving the logical relation.

Chapter 4

Core ML

The transformations done during lifiting take an ML program as input, and return an ML program as output. We will consider here a small subset of OCaml [Leroy et al., 2020], featuring data types and polymorphic let bindings. In particular, our core language has no references or module system. We discusses extending ornamentation to a larger language in §15.4.

4.1 Notation

We often need to write tuples and sequences of arguments. We write $(Q_i)^{i \in I}$ for a tuple $(Q_1, .. Q_n)$. We often omit the set I in which i ranges and just write $(Q_i)^i$, using different indices i, j, and k for ranging over different sets I, J, and K; we also write $\overline{Q}$ if we do not have to explicitly mention the components Q_i. In particular, $\overline{Q}$ stands for $(Q, .. Q)$ in syntax definitions. We write $Q[z_i \leftarrow Q_i]^{i \in I}$, or $Q[z_i \leftarrow Q_i]^i$ for short, for the simultaneous substitution of z_i by Q_i in Q for all i in I. We sometimes need to consider dependent indices: we write for example $j \in J_i$ for j ranging on a set J_i depending on the index i, and $(Q_{ij})^{i \in I, j \in J_i}$ for the sequence ranging over the pairs of i and j. In this case, we usually keep the explicit notation. The order of the indices is relevant: we consider that the order relation $<$ orders the indices such that the index of an element is smaller than the indices of elements to its right.

4.2 Types and datatypes

To prepare ML for further extensions, we slightly depart from traditional presentations. Instead of defining type schemes as a generalization of monomorphic types, we do the converse and introduce monotypes as a restriction of type schemes. The reason to do so is to be able to see both ML and eML as sublanguages of mML—the most expressive of the three. We use kinds to distinguish between the types of the different languages: for ML we only need a kind Typ, to classify the monomorphic types, and its superkind Sch, to classify type schemes. The syntax of kinds (noted κ) and types (σ, τ) is given on Figure 4.1. The monotypes include function types $\tau_1 \to \tau_2$ and application of type constructors $d\,\overline{\tau}$, while universal quantification $\forall(\alpha : \mathsf{Typ})\ \tau$, restricted to monotypes, builds a

κ	$::=$	Kinds
	$\mid \mathsf{Typ}$	Monotypes
	$\mid \mathsf{Sch}$	Type schemes
τ, σ	$::=$	Types
	$\mid \alpha$	Type variable
	$\mid \tau \to \tau$	Function type
	$\mid \zeta\ \overline{\tau}$	Datatype
	$\mid \forall(\alpha : \mathsf{Typ})\ \tau$	Universal quantification
Γ	$::=$	Typing environments
	$\mid \varnothing$	Empty
	$\mid \Gamma, x : \tau$	Variable
	$\mid \Gamma, \alpha : \mathsf{Typ}$	Type variable
D	$::=$	Datatype environments
	$\mid \varnothing$	Empty
	$\mid D, \zeta : (\mathsf{Typ}, .. \mathsf{Typ}) \Rightarrow \mathsf{Typ}$	Type constructor
	$\mid D, d : \forall(\alpha_i : \mathsf{Typ})^i\ (\tau_j)^j \to \zeta\ (\alpha_i)^i$	Data constructor

Figure 4.1: Types, kinds and environments for ML

type scheme. Diverging from the traditional ML syntax, we take type constructors as prefix: thus, the type of lists of booleans is noted list bool instead of bool list. Type environments Γ bind a set of type variables α, all of kind Typ, and term variables x_i of types τ_i, which may be type schemes.

We assume given a fixed set of type constructors, written ζ. Each type constructor has a fixed signature of the form $(\mathsf{Typ}, .. \mathsf{Typ}) \Rightarrow \mathsf{Typ}$. We require that type expressions respect the kinds of type constructors and type constructors are always fully applied. We also assume given a set of data constructors. Each data constructor d comes with a type signature, which is a closed type scheme of the form $\forall(\alpha_i : \mathsf{Typ})^i\ (\tau_j)^j \to \zeta\ (\alpha_i)^i$. The mapping between type constructors, data constructors and their signatures is stored in a datatype environment D, whose syntax is given on Figure 4.1.

By convention, we choose lowercase names for type constructors, and capitalized names for data constructors. Some well-known datatypes will be useful in examples: the datatype unit with one constructor Unit, the datatype bool with two constructors False and True, the empty datatype void with no constructors. We will also use the type of natural numbers nat with two constructors Z and S, the type of lists list α with two constructors Nil and Cons, the type option α with two constructors None and Some, the type either (α, β), with constructors Left and Right, and the type pair (α, β) with one constructor Pair. The corresponding type environment is given on Figure 4.2

The well-formedness rules for datatype environments, type environments, and types are given in Figure 4.3. A datatype environment D is well-formed (noted $\vdash D$) if the types given to the fields of data constructors are valid (Wf-Datatype). Note that this definition permits arbitrary recursion between types, including mutually recursive types, non-strictly positive types, and non-regular types [Bird and Meertens, 1998]. A typing environment Γ is well-formed

$$
\begin{array}{lcl}
\mathsf{unit} & : & \mathsf{Typ} \\
\mathsf{Unit} & : & \mathsf{unit} \\
\\
\mathsf{bool} & : & \mathsf{Typ} \\
\mathsf{True} & : & \mathsf{bool} \\
\mathsf{False} & : & \mathsf{bool} \\
\\
\mathsf{void} & : & \mathsf{Typ} \\
\\
\mathsf{nat} & : & \mathsf{Typ} \\
\mathsf{Z} & : & \mathsf{nat} \\
\mathsf{S} & : & \mathsf{nat} \rightarrow \mathsf{nat} \\
\\
\mathsf{list} & : & \mathsf{Typ} \Rightarrow \mathsf{Typ} \\
\mathsf{Nil} & : & \forall(\alpha : \mathsf{Typ})\ \mathsf{list}\ \alpha \\
\mathsf{Cons} & : & \forall(\alpha : \mathsf{Typ})\ (\alpha, \mathsf{list}\ \alpha) \rightarrow \mathsf{list}\ \alpha \\
\\
\mathsf{option} & : & \mathsf{Typ} \Rightarrow \mathsf{Typ} \\
\mathsf{None} & : & \forall(\alpha : \mathsf{Typ})\ \mathsf{option}\ \alpha \\
\mathsf{Some} & : & \forall(\alpha : \mathsf{Typ})\ \alpha \rightarrow \mathsf{option}\ \alpha \\
\\
\mathsf{either} & : & (\mathsf{Typ}, \mathsf{Typ}) \Rightarrow \mathsf{Typ} \\
\mathsf{Left} & : & \forall(\alpha : \mathsf{Typ}, \beta : \mathsf{Typ})\ \alpha \rightarrow \mathsf{either}\ (\alpha, \beta) \\
\mathsf{Right} & : & \forall(\alpha : \mathsf{Typ}, \beta : \mathsf{Typ})\ \beta \rightarrow \mathsf{either}\ (\alpha, \beta) \\
\\
\mathsf{pair} & : & (\mathsf{Typ}, \mathsf{Typ}) \Rightarrow \mathsf{Typ} \\
\mathsf{Pair} & : & \forall(\alpha : \mathsf{Typ}, \beta : \mathsf{Typ})\ (\alpha, \beta) \rightarrow \mathsf{pair}\ (\alpha, \beta)
\end{array}
$$

Figure 4.2: Definition of some datatypes

$$\textsc{EnvEmpty}\quad D \vdash \varnothing$$

$$\textsc{EnvVar}\quad \frac{D \vdash \Gamma \qquad \Gamma \vdash \tau : \mathsf{Sch} \qquad x \# \Gamma}{D \vdash \Gamma, x : \tau}$$

$$\textsc{EnvTVar}\quad \frac{D \vdash \Gamma \qquad \alpha \# \Gamma}{D \vdash \Gamma, \alpha : \mathsf{Typ}}$$

$$\textsc{K-Var}\quad \frac{D \vdash \Gamma \qquad \alpha : \mathsf{Typ} \in \Gamma}{D; \Gamma \vdash \alpha : \mathsf{Typ}}$$

$$\textsc{K-Datatype}\quad \frac{D \vdash \Gamma \qquad (\zeta : (\mathsf{Typ})^i \Rightarrow \mathsf{Typ}) \in D \qquad (D; \Gamma \vdash \tau_i : \mathsf{Typ})^i}{D; \Gamma \vdash \zeta\,(\tau_i)^i : \mathsf{Typ}}$$

$$\textsc{K-Arr}\quad \frac{D; \Gamma \vdash \tau_1 : \mathsf{Typ} \qquad D; \Gamma \vdash \tau_2 : \mathsf{Typ}}{D; \Gamma \vdash \tau_1 \to \tau_2 : \mathsf{Typ}}$$

$$\textsc{K-SubTyp}\quad \frac{D; \Gamma \vdash \tau : \mathsf{Typ}}{D; \Gamma \vdash \tau : \mathsf{Sch}}$$

$$\textsc{K-All}\quad \frac{D; \Gamma, \alpha : \mathsf{Typ} \vdash \tau : \mathsf{Sch}}{D; \Gamma \vdash \forall(\alpha : \mathsf{Typ})\,\tau : \mathsf{Sch}}$$

$$\textsc{Wf-Datatype}\quad \frac{\begin{array}{c}\forall(d : \forall(\alpha_i : \mathsf{Typ})^i\,(\tau_j)^j \to \zeta\,(\alpha_i)^i) \in D\,, \\ (\zeta : (\mathsf{Typ})^i \Rightarrow \mathsf{Typ}) \in D \wedge (D; (\alpha_i : \mathsf{Typ})^i \vdash \tau_j : \mathsf{Typ})^j\end{array}}{\vdash D}$$

Figure 4.3: Well-formedness rules

$$
\begin{array}{rll}
a, b & ::= & \text{Terms} \\
& \mid\ x & \text{Variable} \\
& \mid\ \mathsf{let}\ x = a\ \mathsf{in}\ a & \text{Let binding} \\
& \mid\ \mathsf{fix}\, x\ (x : \tau) : \tau\ .\ a & \text{Function by fixed point} \\
& \mid\ a\ a & \text{Application} \\
& \mid\ a\ \tau & \text{Type application} \\
& \mid\ \Lambda(\alpha : \mathsf{Typ}).\ u & \text{Type abstraction} \\
& \mid\ d\,\overline{\tau}\ \overline{a} & \text{Construction} \\
& \mid\ \mathsf{match}\ a\ \mathsf{with}\ (P \to a \mid ..\, P \to a) & \text{Pattern matching} \\
P & ::= & \text{Patterns} \\
& \mid\ d\,\overline{\tau}\ \overline{x} &
\end{array}
$$

$$
\begin{array}{rl}
u\ ::= & x \mid d\,\overline{\tau}\ \overline{u} \mid \mathsf{fix}\, x\ (x : \tau) : \tau\ .\ a \mid u\ \tau \mid \Lambda(\alpha : \kappa).\ u \mid \mathsf{let}\ x = u\ \mathsf{in}\ u \\
\mid & \mathsf{match}\ u\ \mathsf{with}\ (P \to u \mid ..\, P \to u)
\end{array}
$$

Figure 4.4: Syntax of ML

in a datatype environment D, noted $D \vdash \Gamma$ if it only contains well-formed types and does not bind the same variable twice. Finally, a type τ is well-formed with kind κ in a context $D; \Gamma$, noted $D; \Gamma \vdash \tau : \kappa$ if all applications of type constructors have the right number of arguments (K-Datatype) and any variable that appears in the type is bound in the context or by a preceding universal quantification (K-Var). If it does not contain universal quantification, it has kind Typ. Otherwise, all universal quantification must be at the head of the type (because the kind Sch cannot appear as the argument of the function arrow or a type constructor), and it has kind Sch (K-All). The rule K-SubTyp allows monotypes to be considered as an instance of type schemes. Types (and terms) are considered up to α-conversion.

Since the datatype environment D does not vary, we will almost always leave it implicit.

4.3 The syntax of explicit ML

Since we are mainly interrested in transforming ML programs, we describe an explicitly typed version of ML.

The syntax of ML is given on Figure 4.4. Terms are represented by the meta-variables a and b. Instead of having a special notation for recursive functions, functions are always defined recursively, using the construction $\mathsf{fix}\, f\ (x : \tau_1) : \tau_2\ .\ a$. This avoids having two different syntactic forms for values of function types. For convenience, we still use the standard notation $\lambda(x : \tau_1).\ a$ for non-recursive functions, but we just see it as a shorthand for $\mathsf{fix}\, f\ (x : \tau_1) : \tau_2\ .\ a$ where f does not appear free in a and τ_2 is the function's return type. Type abstraction and type application are explicit, respectively noted $\Lambda(\alpha : \mathsf{Typ}).\ a$ and $a\ \tau$. Constructors d take as arguments both a list of types, instantiating the corresponding type arguments of the datatype, and a list of terms corresponding to their fields.

$$\textsc{Var}\quad \frac{\vdash \Gamma \qquad x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad\qquad \textsc{TAbs}\quad \frac{\Gamma, \alpha : \mathsf{Typ} \vdash u : \sigma}{\Gamma \vdash \Lambda(\alpha : \mathsf{Typ}).\, u : \forall(\alpha : \mathsf{Typ})\, \sigma}$$

$$\textsc{TApp}\quad \frac{\Gamma \vdash \tau : \mathsf{Typ} \qquad \Gamma \vdash a : \forall(\alpha : \mathsf{Typ})\, \sigma}{\Gamma \vdash a\, \tau : \sigma[\alpha \leftarrow \tau]}$$

$$\textsc{Fix}\quad \frac{\Gamma \vdash \tau_1 : \mathsf{Typ} \qquad \Gamma \vdash \tau_2 : \mathsf{Typ} \qquad \Gamma, x : \tau_1 \to \tau_2, y : \tau_1 \vdash a : \tau_2}{\Gamma \vdash \mathsf{fix}\, x\, (y : \tau_1) : \tau_2\ .\, a : \tau_1 \to \tau_2}$$

$$\textsc{App}\quad \frac{\Gamma \vdash b : \tau_1 \qquad \Gamma \vdash a : \tau_1 \to \tau_2}{\Gamma \vdash a\, b : \tau_2} \qquad\qquad \textsc{Let}\quad \frac{\Gamma \vdash a : \tau' \qquad \Gamma, x : \tau' \vdash b : \tau}{\Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ b : \tau}$$

$$\textsc{Con}\quad \frac{\begin{array}{c}\vdash \Gamma \qquad \vdash d : \forall(\alpha_j : \mathsf{Typ})^j\, (\tau_i)^i \to \zeta\, (\alpha_j)^j \\ (\Gamma \vdash \tau_j : \mathsf{Typ})^j \qquad (\Gamma \vdash a_i : \tau_i[\alpha_j \leftarrow \tau_j]^j)^i\end{array}}{\Gamma \vdash d(\tau_j)^j (a_i)^i : \zeta\, (\tau_j)^j}$$

$$\textsc{Match}\quad \frac{\begin{array}{c}\Gamma \vdash \tau : \mathsf{Sch} \qquad (d_i : \forall(\alpha_k : \mathsf{Typ})^k\, (\tau_{ij})^j \to \zeta\, (\alpha_k)^k)^i \\ \Gamma \vdash a : \zeta\, (\tau_k)^k \qquad (\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j \vdash b_i : \tau)^i \qquad \vdash (d_i)^i : \zeta\ \mathsf{complete}\end{array}}{\Gamma \vdash \mathsf{match}\ a\ \mathsf{with}\ (d_i(\tau_k)^k (x_{ij})^j \to b_i)^i : \tau}$$

Figure 4.5: Typing rules for ML

We distinguish a class of *non-expansive terms*, noted u, whose evaluation does not involve reducing function application. We require that the body of type abstractions is non-expansive. This is a slightly relaxed version of value restriction [Wright, 1995]. Non-expansive terms will be useful later on because their reduction terminates. Note that the class of non-expansive terms is not preserved by substitution of variables for arbitrary terms, but only for substitution by non-expansive terms. To preserve the syntax of terms, we must be careful to only substitute non-expansive terms for variables.

The typing judgment $\Gamma \vdash a : \tau$ (which is a shorthand for $D; \Gamma \vdash a : \tau$, with D considered constant), expresses that a is well-typed of type τ in the typing context Γ. Its derivation rules are given on Figure 4.5. We need to ensure that all pattern matchings are non-overlapping and complete. This is done in Match through an auxiliary judgment $\vdash (d_i)^i : \zeta\ \mathsf{complete}$, defined as follows: $\vdash (d_i)^i : \zeta\ \mathsf{complete}$ if and only if $(d_i)^i$ contains once and only once all constructors d such that there exists $(\alpha_j)^j$ and $(\tau_k)^k$ such that $(d_i : \forall(\alpha_k : \mathsf{Typ})^k\, (\tau_{ij})^j \to \zeta\, (\alpha_k)^k) \in D$. The rules of the typing judgment ensure that if $\Gamma \vdash a : \tau$, then Γ is well-formed and $\Gamma \vdash \tau : \mathsf{Sch}$; in rule Match this requires checking explicitly that $\Gamma \vdash \tau : \mathsf{Sch}$, *i.e.* that τ and Γ are well-formed, as the pattern matching can have zero branch if we are matching on a datatype without constructors.

$$
\begin{array}{rcl}
v & ::= & d\,\overline{\tau}\,\overline{v} \mid \mathsf{fix}\,x\,(x:\tau):\tau\,.\,a \mid \Lambda(\alpha:\mathsf{Typ}).\,v \\
E & ::= & [] \mid E\,a \mid v\,E \mid d\,\overline{\tau}\,(a,..\,a,E,v,..\,v) \mid \Lambda(\alpha:\mathsf{Typ}).\,E \mid E\,\tau \\
 & \mid & \mathsf{match}\,E\,\mathsf{with}\,(P \to a \mid ..\,P \to a) \mid \mathsf{let}\,x = E\,\mathsf{in}\,a
\end{array}
$$

$$
\begin{array}{rcl}
(\mathsf{fix}\,x\,(y:\tau'):\tau\,.\,a)\,v & \longrightarrow_\beta^h & a[x \leftarrow \mathsf{fix}\,x\,(y:\tau'):\tau\,.\,a, y \leftarrow v] \\
(\Lambda(\alpha:\mathsf{Typ}).\,v)\,\tau & \longrightarrow_\beta^h & v[\alpha \leftarrow \tau] \\
\mathsf{let}\,x = v\,\mathsf{in}\,a & \longrightarrow_\beta^h & a[x \leftarrow v] \\
\mathsf{match}\,d_j\,\overline{\tau}\,(v_i)^i\,\mathsf{with}\,(d_j\,\overline{\tau}\,(x_{ji})^i \to a_j)^j & \longrightarrow_\beta^h & a_j[x_{ji} \leftarrow v_i]^i
\end{array}
$$

$$
\textsc{Context-Beta}\quad \frac{a \longrightarrow_\beta^h b}{E[a] \longrightarrow_\beta E[b]}
$$

Figure 4.6: Reduction for ML

4.4 Evaluation

The language is equipped with a weak (*i.e.* we disallow reduction under binders), call-by-value reduction semantics, expressed in small-step style. To match the behavior of OCaml, we evaluate applications left-to-right and constructor arguments right-to-left. The definitions of values v, of the evaluation contexts E are given in Figure 4.6. The reduction $\longrightarrow_\beta$ is defined as performing head-reductions $\longrightarrow_\beta^h$ in evaluation contexts E. While reduction does not occur under term abstractions, we reduce under type abstractions: thus evaluation occurs as if the type abstractions were absent. This is important because abstractions are not visible in the implicitly-typed language that is presented to the programmer. If type abstractions did block reduction, the programmer would have to guess where abstractions are inserted to understand the semantics of the program they wrote.

Reduction always substitutes variables by values, which are a subset of non-expansive terms. Thus, the syntax of terms is preserved. Given the evaluation contexts we have defined, the reduction is deterministic, and values are indeed irreducible.

Lemma 4.1 (Values are irreducible for $\longrightarrow_\beta$)**.** *Consider a value v. Then, there is no term a such that $v \longrightarrow_\beta a$.*

Proof. By contradiction, consider a reduction of v. There exists a decomposition $v = E[b]$ where E is an evaluation context, and $b \longrightarrow_\beta^h b'$, and $a = E[b']$.

Let us prove by structural induction of values that this is impossible. There is no head-reduction on values (because values start with a constructor, while head-reduction reduces only terms that start with a destructor). Thus, the evaluation context E is non-empty. Let us consider the different cases for values:

- Suppose $v = d(\tau)^i(v'_i)^i$. Then E is of the form $d(\tau)^i((v'_j)^j, E', (v'_k)^k)$, therefore there is some value v' in $(v'_i)^i$ that decomposes as $v' = E'[b]$ and reduces to $E'[b']$. This is impossible by induction hypothesis.

- Suppose $v = \mathsf{fix}\,x\,(y:\tau'):\tau\,.\,a$. There is no possible non-empty context E such that $v = E[b]$.

- Suppose $v = \Lambda(\alpha : \mathsf{Typ}).\, v'$ for some value v'. Then $E = \Lambda(\alpha : \mathsf{Typ}).\, E'$, thus $v' = E'[b]$ reduces to $E'[b']$. This is impossible by induction hypothesis. □

Lemma 4.2 ($\longrightarrow_\beta$ is deterministic)**.** *Suppose $a \longrightarrow_\beta b$ and $a \longrightarrow_\beta b'$. Then $b = b'$.*

Proof. Let us show that any term a decomposes in at most one way as $E[a']$ where a' is head-reducible. This implies the lemma, because head-reduction is deterministic.

We proceed by induction on a.

- If the term is a variable, the reduction is blocked.
- If it is a let binding $a = \mathsf{let}\ x = b\ \mathsf{in}\ b'$ there are two cases to consider. If b is not a value, a does not head-reduce. Then, the decomposition must of the form $E = \mathsf{let}\ x = E'\ \mathsf{in}\ b'$ and we apply the induction hypothesis on b. If b is a value, the decomposition cannot be of this form, since that would imply that b reduces. Thus, E is necessarily the empty context.
- If the term is a function definition by fixed point, it is a value, and thus irreducible.
- If $a = b\ b'$ is an application, there are three cases to consider. If b is not a value, the term does not head-reduce, and the only possible decomposition is of the form $E = E'\ b'$. Then we can apply the inductive hypothesis to b. If b is a value but b' is not, the term does not head-reduce, and the only possible decomposition is of the form $E = b\ E'$ (b does not decompose as it is a value). The we can apply the inductive hypothesis to b'. Otherwise, we necessarily have $E = []$.
- Similarly, for type application $a = b\ \tau$, either b is not a value and a does not head-reduce, or b is a value, does not decompose further, and E is the empty context.
- For type abstraction the context is necessarily of the form $E = \Lambda(\alpha : \mathsf{Typ}).\, E'$, and we can apply the induction hypothesis.
- For constructors we find the first field (starting from the right) that is not a value, and apply the inductive hypothesis.
- For pattern matching, either the scrutinee is not a value and the term does not head-reduce, then we can apply the induction hypothesis, or it is a value and does not decompose so the empty context is the only possible context. □

4.5 Type soundness

As usual, weakening allows us to transport a term to another context, which is useful to prove that substitution preserves types. We have two ways to enrich an environment: either add a type variable $\alpha : \mathsf{Typ}$, or add a term variable $a : \tau$. Similarly, we have three judgments where a typing environment appears: well-formedness of environments $\vdash \Gamma$, well-formedness of types $\Gamma \vdash \tau : \kappa$ and typing $\Gamma \vdash a : \tau$. Thus, we have six statements of weakening:

Lemma 4.3 (Weakening)**.** *Suppose* $\vdash \Gamma, \Gamma'$*. Then,*

- *if* $\alpha \# \Gamma, \Gamma'$*, we have* $\vdash \Gamma, \alpha : \mathsf{Typ}, \Gamma'$*;*
- *if* $x \# \Gamma, \Gamma'$ *and* $\Gamma \vdash \tau : \mathsf{Sch}$*, then* $\vdash \Gamma, x : \tau, \Gamma'$*.*

Suppose $\Gamma, \Gamma' \vdash \tau : \kappa$*. Then,*

- *if* $\alpha \# \Gamma, \Gamma'$*, we have* $\Gamma, \alpha : \mathsf{Typ}, \Gamma' \vdash \tau : \kappa$*;*
- *if* $x \# \Gamma, \Gamma'$ *and* $\Gamma \vdash \tau' : \mathsf{Sch}$*, then* $\Gamma, x : \tau', \Gamma' \vdash \tau : \kappa$*.*

Suppose $\Gamma, \Gamma' \vdash a : \tau$*. Then,*

- *if* $\alpha \# \Gamma, \Gamma'$*, then* $\Gamma, \alpha : \mathsf{Typ}, \Gamma' \vdash a : \tau$*;*
- *if* $x \# \Gamma, \Gamma'$ *and* $\Gamma \vdash \tau : \mathsf{Sch}$*, then* $\Gamma, x : \tau, \Gamma' \vdash a : \tau$*.*

Proof. By induction on the derivation. For weakening on type variables, since the variable is free for the context, the context stays valid (EnvTVar), and the only rule that accesses the context for type variables (K-Var) is monotonic with respect to the context. Similarly for term variables, with EnvVar and Var. □

Similarly, substitution has six different statements:

Lemma 4.4 (Substitution)**.** *Suppose* $\Gamma \vdash \sigma : \mathsf{Typ}$*. Then,*

- *if* $\vdash \Gamma, \alpha : \mathsf{Typ}, \Gamma'$*, then* $\vdash \Gamma, \Gamma'[\alpha \leftarrow \sigma]$*;*
- *if* $\Gamma, \alpha : \mathsf{Typ}, \Gamma' \vdash \tau : \kappa$*, then* $\Gamma, \Gamma'[\alpha \leftarrow \sigma] \vdash \tau[\alpha \leftarrow \sigma] : \kappa$*;*
- *if* $\Gamma, \alpha : \mathsf{Typ}, \Gamma' \vdash a : \tau$*, then* $\Gamma, \Gamma'[\alpha \leftarrow \sigma] \vdash a[\alpha \leftarrow \sigma] : \tau[\alpha \leftarrow \sigma]$*.*

Suppose $\Gamma \vdash u : \tau$*.*

- *if* $\vdash \Gamma, x : \tau, \Gamma'$*, then* $\vdash \Gamma, \Gamma'$*;*
- *if* $\Gamma, x : \tau, \Gamma' \vdash \tau : \kappa$*, then* $\Gamma, \Gamma' \vdash \tau : \kappa$*;*
- *if* $\Gamma, x : \tau, \Gamma' \vdash a : \tau$*, then* $\Gamma, \Gamma' \vdash a[x \leftarrow u] : \tau$*.*

Proof. By induction on the derivations. For type substitution, substitute in subderivations by induction, replacing K-Var by the derivation of $\Gamma \vdash \sigma : \mathsf{Typ}$. For term substitution, substitute in subderivations by induction, replacing rule Var by the derivation of $\Gamma \vdash u : \tau$. □

The reduction preserves typing.

Lemma 4.5 (Subject reduction for head reduction)**.** *Suppose* $\Gamma \vdash a : \tau$*. Then, if* $a \longrightarrow^h_\beta a'$*, we have* $\Gamma \vdash a' : \tau$*.*

Proof. Consider the different redexes.

- If $a = (\mathsf{fix}\, x\, (y : \tau_1') : \tau_2' \,.\, b)\; v$, then the last derivation rule is necessarily App. Thus, there exists τ_1 such that $\Gamma \vdash \mathsf{fix}\, x\, (y : \tau_1') : \tau_2' \,.\, b : \tau_1 \to \tau_2$ and $\Gamma \vdash v : \tau_1$. The last derivation rule of $\Gamma \vdash \mathsf{fix}\, x\, (y : \tau_1') : \tau_2' \,.\, b : \tau_1 \to \tau_2$ is necessarily Fix. Thus, we have $\Gamma, x : \tau_1 \to \tau, y : \tau_1 \vdash b : \tau$. By substitution (Lemma 4.4), $\Gamma \vdash a[x \leftarrow \mathsf{fix}\, x\, (y : \tau') : \tau \,.\, b, y \leftarrow v] : \tau$.

- Proceed similarly for the other redexes (type application and abstraction, match and constructor, and let and value), by inverting the typing derivation and applying substitution. □

Theorem 4.1 (Subject reduction)**.** *Suppose* $\Gamma \vdash a : \tau$*. Then, if* $a \longrightarrow_\beta a'$*, we have* $\Gamma \vdash a' : \tau$*.*

Proof. By induction on the context where head reduction takes place.

- If the context is empty, apply Lemma 4.5.
- Otherwise, notice that typing derivations are not dependent: we can change subterms as long as we keep the same type. Then apply the induction hypothesis on the reduced subterm. □

Moreover, well-typed irreducible terms are values.

Lemma 4.6 (Inversion)**.** *Consider a value* v*, such that* $\Gamma \vdash v : \tau$*.*

- *If* $\tau = \tau_1 \to \tau_2$*, there exists* a *such that* $v = \mathsf{fix}\, x\, (y : \tau_1) : \tau_2\ .\, a$*.*
- *If* $\tau = \forall(\alpha : \mathsf{Typ})\, \tau'$*, there exists* w *such that* $v = \Lambda(\alpha : \mathsf{Typ}).\, w$*.*
- *If* $\tau = \zeta\, (\tau_i)^i$*, there exists a constructor* $\vdash d : \forall(\alpha_i : \mathsf{Typ})^i\, (\tau_j)^j \to \zeta\, (\alpha_i)^i$ *and terms* $(a_j)^j$ *such that* $a = d(\tau_i)^i(a_j)^j$*.*

Proof. By examining the different cases for values, and inverting the typing derivation. □

Theorem 4.2 (Progress)**.** *Consider* $\varnothing \vdash a : \tau$*. Then, either* a *reduce, or it is a value.*

Proof. We proceed by induction on the term, as in the proof of Lemma 4.2. In the induction, we will reduce under type abstractions. For the proof by induction to work, we need to prove a strengthened result where the typing environment can bind any number of type variables: consider a set of type variables $(\alpha_i)^i$. Then, if $(\alpha_i : \mathsf{Typ})^i \vdash a : \tau$, either a is a value or a reduces.

- Term variables are not well-typed in the empty context.
- A function definition by fixed point is a value.
- Each argument of the constructor is well-typed in Γ, thus is either a value or reduces. If all arguments are values, a is a value. Otherwise, we can reduce the rightmost non-value argument.
- The term inside a well-typed type abstraction is well-typed in an environment $\Gamma, \alpha : \mathsf{Typ}$ enriched with one type variable, thus is either a value or reduces. If it reduces, the abstraction reduces. Otherwise a is a value.
- If a is a let binding $\mathsf{let}\ x = b\ \mathsf{in}\ b'$, apply the induction hypothesis on b, well-typed in Γ. If b reduces to b'', a reduces to $\mathsf{let}\ x = b''\ \mathsf{in}\ b'$. Otherwise, b is a value, and a reduces to $b'[x \leftarrow b]$.

- In the case of applications $a = b\ b'$, by inverting the last typing rule, b and b' are well-typed. If b' reduces, a reduces. Otherwise, it is a value. Then, if b reduces, a reduces. Let us assume b and b' are values. By inverting the type derivation, we know that there exists a type τ_1 such that $\varnothing \vdash b' : \tau_1$ and $\varnothing \vdash b : \tau_1 \to \tau$. Then, by Lemma 4.6, we know that there exists a' such that $b = \mathsf{fix}\, x\ (y : \tau_1) : \tau\ .\ a'$. Then, a reduces to $a'[x \leftarrow b, y \leftarrow b']$.

- Proceed similarly for pattern matching and type application. $\square$

Chapter 5

Labelling ML terms

We introduce a labeling system for ML: expansive computations are *labeled* so that their result can be reused without recomputation later in the term, without having to introduce explicit bindings. This will be used in *e*ML, presented in the next chapter, to refer to results of previous computations that were useful in determining, *e.g.* which branch is taken. *e*ML only requires the labels being available at type-level. The language we present here also has labels at value level (and, indeed, is mostly untyped).

5.1 Overview

When lifting an ML program, we provide patches to fill in the missing parts of the term. When writing these patches, we might need to access a previously computed result after it has been computed. For example, suppose we are lifting the following term by lifting the output boolean to an integer:

$$\mathsf{match\ random_int_option\ ()\ with\ Some\ _ \rightarrow True \mid None \rightarrow False}$$

We obtain a program of the form

$$\mathsf{match\ random_int_option\ ()\ with\ Some\ _ \rightarrow Some\ \#1 \mid None \rightarrow None}$$

To provide a value for the hole #1 we need to access the result of the call to random_int_option, but we dropped some information. We cannot call the function a second time, as it may have side-effects, or in this case return a different result altogether. If we can recover the result of the call, we can simply match on it again to finish the transformation.

This is already achievable in ML by program transformation: in that case, we simply have to *bind* the result of the computation to a variable that we can reuse later on:

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{random_int_option}\ ()\ \mathsf{in}\\ \mathsf{match}\ x\ \mathsf{with\ Some\ _} \rightarrow \mathsf{True} \mid \mathsf{None} \rightarrow \mathsf{False}\end{array}$$

The lifted term becomes:

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{random_int_option}\ ()\ \mathsf{in}\\ \mathsf{match}\ x\ \mathsf{with\ Some\ _} \rightarrow \mathsf{Some}\ \#1 \mid \mathsf{None} \rightarrow \mathsf{None}\end{array}$$

The hole #1 can then be patched with match x with Some $y \to y$ | None $\to 0$.

Note that we have another issue there: we know from the branch we are in that x is necessarily Some _, but we have to provide a complete pattern matching. In this example, we simply insert a placeholder value there. We solve this problem in Chapter 6 by teaching the type system how to use branch information to decide that some branches are unreachable. This re-uses the labelling system to refer to previously performed computations.

We don't need to give names to computations that don't involve function application: we consider them both safe and cheap to duplicate if needed.

More complicated cases can also be handled by program transformation, although at increased cost. Consider for example the program

```
let x1 = match f () with
  | True → g () != 2
  | False → False
in
C[#1]
```

We can only bind g () to a variable if it is actually evaluated, which means we have to extrude the match around the binding of x_1 and duplicate the context $C[_]$ so we can reuse the result in #1:

```
let y1 = f () in
match y1 with
  | True →
  let y2 = g () in
  let x1 = y2 != 2 in
  C[#1]
  | False →
  let x1 = False in
  C[#2]
```

With more pattern matching, we would have to duplicate even more code, nest even more pattern matchings, and generate even more patches. Moreover, our lifting strategy does not allow us to perform only the necessary extrusions: since we want to create a completely generic term that can support any lifting of the base function, we have to perform all extrusions, even if they are unneeded.

We would rather take the inverse approach: only perform the necessary extrusions at the latest possible moment, once the term has been instantiated and we are reducing it back to an ML term. We label each application in the execution of a program with an unique label p, noted above the application as in $f^p\ x$. When we want to reuse the result of a previously executed application, we access the value of the label, noted $*p$. Our first example becomes: match random_int_option^1 () with Some _ $\to$ Some #1 | None $\to$ None, and we can patch it with match $*1$ with Some $y \to y$ | None $\to 0$.

Labels are substituted as soon as the corresponding expansive computation is evaluated: thus, labels are accessible in all subterms evaluated after the application that binds them in the ML evaluation order.

We want applications to have unique names, and application of a function can create new applications to be evaluated later by duplicating the code inside the body of the function. Therefore, we need a way for each separate application to generate its own labels. Instead of using abstract names for labels, we use

paths constructed by concatenating abstract labels. We want a function application labelled by a (unique) label p to only generate applications prefixed by p with a unique suffix. This guarantees that the labels generated throughout the execution are unique. We add to each term abstraction a label variable π: all applications in execution position inside the abstraction will then have a label prefixed by π.

Our presentation here is untyped, we only add some well-labeling checks to ensure unicity of the generated labels (and later confluence, see §7.3.1). In particular, we do not worry about using labels that are only defined in some branches: the problem of knowing whether a particular label is valid to use or not is solved by the extension to eML in Chapter 6.

When reducing an application labelled with p, we immediately substitute all instances of $*p$ with the result of the application. This requires some care: consider $\mathsf{let}\ x = (\lambda^{\pi}\,(y : \mathsf{unit}).\ \mathrm{random}^{\pi\cdot 1}\ ())^{1}\ ()\ \mathsf{in}\ (x, *1)$. If we substituted all instances of $*1$ with the result of the application when reducing the lambda, we would obtain $\mathsf{let}\ x = \mathrm{random}^{1\cdot 1}\ ()\ \mathsf{in}\ (x, \mathrm{random}^{1\cdot 1}\ \mathsf{Unit})$. This is wrong: we duplicated the call to random and we will probably end up with different results for the two calls. We also duplicated a label, while we want labels to uniquely identify applications. Instead, we substitute a *reusable* version of the term, obtained by replacing all new applications by a reuse of their label: here, this reusable version is simply $*1 \cdot 1$. The reduced term is thus: $\mathsf{let}\ x = \mathrm{random}^{1\cdot 1}\ ()\ \mathsf{in}\ (x, *1 \cdot 1)$.

5.2 Labelled reduction

In this section, we define the syntax of labelled terms and their reduction (Figure 5.1).

Result names, noted p and q are formed by appending a sequence of path components n to either the root name ε or a path variable π. When writing a path starting from the root component, we may omit it: we write $n_1 \cdot n_2$ for $\varepsilon \cdot n_1 \cdot n_2$. If q is a path starting from the root, we note $p \cdot q$ the path formed by appending all path components of q to p. We say a path q is a suffix of p, noted $p \leqslant q$, if q can be formed by adding some path components to p. It is a strict suffix if it is different from p.

The syntax of terms a is identical to the syntax of ML, with a few deviations: first, all applications are labelled: $a^p\ b$ is read as the application of a to b labelled with p. Second, we write $*p$ the reuse of the result of the application labelled p. When the application is reduced, $*p$ is replaced with the result of the reduction. Third, lambda abstractions also bind a path variable π. This variable can be used as a prefix for the labels appearing in the variable (either being defined or used). When an application is reduced, the label variable on the lambda is substituted with the label of the application in the body of the abstraction. The grammar of non-expansive terms u is also extended. As before, applications are expansive. Conversely, reused results $*p$ are always non-expansive. The grammar of values v is unchanged, except for the label variables added to abstractions.

The definition of the evaluation contexts, the head reduction, and the context rule are given on Figure 5.2. We define evaluation from elementary context layers $\mathcal{E}$: they represent the possible right-to-left evaluation contexts in ML,

a, b	$::=$	Terms
	$\mid x$	Variable
	$\mid \mathsf{let}\ x = a\ \mathsf{in}\ a$	Let binding
	$\mid \mathsf{fix}^{\pi}\ x\ (x : \tau) : \tau\ .\ a$	Function by fixed point
	$\mid a^{p}\ a$	Application
	$\mid a\ \tau$	Type application
	$\mid \Lambda(\alpha : \mathsf{Typ}).\ u$	Type abstraction
	$\mid d\,\overline{\tau}\ \overline{a}$	Construction
	$\mid \mathsf{match}\ a\ \mathsf{with}\ (P \to a \mid ..\, P \to a)$	Pattern matching
	$\mid *p$	Reuse
P	$::=$	Patterns
	$\mid d\,\overline{\tau}\,(x)^{i}$	
p, q	$::=$	Result names
	$\mid \varepsilon$	Root
	$\mid \pi$	Variable
	$\mid p \cdot n$	Path

$$u ::= x \mid d\,\overline{\tau}\ \overline{u} \mid \mathsf{fix}^{\pi}\ x\ (x : \tau) : \tau\ .\ a \mid u\ \tau \mid \Lambda(\alpha : \kappa).\ u \mid \mathsf{let}\ x = u\ \mathsf{in}\ u$$
$$\mid \mathsf{match}\ u\ \mathsf{with}\ (P \to u \mid ..\, P \to u) \mid *p$$
$$v ::= x \mid d\,\overline{\tau}\ \overline{v} \mid \mathsf{fix}^{\pi}\ x\ (x : \tau) : \tau\ .\ a \mid \Lambda(\alpha : \kappa).\ v$$

Figure 5.1: Syntax of labelled ML

$$\begin{array}{rcl}\mathcal{E} & ::= & \mathsf{let}\ x = [\,]\ \mathsf{in}\ a \mid [\,]^p\ a \mid v^p\ [\,] \mid d\,\overline{\tau}\,(a, ..\,a, [\,], v, ..\,v) \mid \Lambda(\alpha : \mathsf{Typ}).\ [\,] \mid [\,]\ \tau \\ & \mid & \mathsf{match}\ [\,]\ \mathsf{with}\ (P \to a \mid ..\,P \to a)\end{array}$$

$$(\Lambda(\alpha : \mathsf{Typ}).\ v)\ \tau \xrightarrow{0} v[\alpha \leftarrow \tau]$$

$$\begin{array}{l}(\mathsf{fix}^\pi\ x\ (y : \tau_1') : \tau_2'\ .\ a)^p\ v \\ \qquad \xrightarrow{*p \leftarrow \mathsf{reuse}(a[\pi \leftarrow p, x \leftarrow \mathsf{fix}^\pi\ x\ (y:\tau_1'):\tau_2'\ .\ a, y \leftarrow \tau_1'])} \\ a[\pi \leftarrow p, x \leftarrow \mathsf{fix}^\pi\ x\ (y : \tau_1') : \tau_2'\ .\ a, y \leftarrow \tau_1']\end{array} \qquad \mathsf{let}\ x = v\ \mathsf{in}\ a \xrightarrow{0} a[x \leftarrow v]$$

$$\mathsf{match}\ d_j\,\overline{\tau}\,(v_i)^i\ \mathsf{with}\ (d_j\,\overline{\tau'}\,(x_{ji})^i \to a_j)^j \xrightarrow{0} a_j[x_{ji} \leftarrow v_i]^i$$

$$\textsc{Red-NoLabel}\ \frac{a \xrightarrow{0} b}{\mathcal{E}[a] \xrightarrow{0} \mathcal{E}[b]} \qquad \textsc{Red-Label}\ \frac{a \xrightarrow{*p \leftarrow u} b}{\mathcal{E}[a] \xrightarrow{*p \leftarrow u} (\mathcal{E}[*p \leftarrow u])[b]}$$

Figure 5.2: Reduction for labeled ML

and embed constrains about the fact that terms to the right must already be reduced to value for a subterm to be in evaluation position.

The reduction $\longrightarrow_\beta$ is similar to the ML reduction, except that the reduction is labelled: a reduction $\xrightarrow{0}$ indicates that the head-reduction was performed on a non-expansive term, while a reduction $\xrightarrow{*p \leftarrow u}$ labelled with a substitution indicates that an expansive term with label p has been reduced to some expression to which u is a representation. Then, all occurrences of $*p$ in the evaluation context are substituted by the non-expansive term u. This is done in the context rules Red-NoLabel (propagating the empty label) and Red-Label (applying the substitution and propagating it to the enclosing context).

The evaluation contexts $\mathcal{E}$ do not bind term variables, and only non-expansive terms occur under type abstraction; thus we do not need to take precautions to ensure that the terms substituted for labels stay well-scoped while moving up the context layers.

The head reduction rules are almost identical to the ML reduction rules. They are also labelled: reductions of non-expansive redexes (type application, let binding and pattern matching) have an empty label. Reductions of applications are labelled with a substitution $*p \leftarrow u$ of the label p of the application with a non-expansive representation u of the result of the application. The non-expansive representation of a term a is noted $\mathsf{reuse}(a)$. It is formed by replacing all expansive computations in evaluation position (not under a term abstraction) by a reuse of their label. This makes it equivalent to the term, as long as the term is already evaluated once before. The definition is given in Figure 5.3: the important rule is given first, the others simply descend on the structure of the terms.

Lemma 5.1 (Values do not reduce)**.** *Values are irreducible for* $\longrightarrow_\beta^h$.

Proof. By structural induction on values. Values are never head-reducible (because they start with a constructor), and if $v = \mathcal{E}[a]$, then a is a value too. □

$$
\begin{array}{lcl}
\mathsf{reuse}(a^p\ b) & = & *p \\
\mathsf{reuse}(x) & = & x \\
\mathsf{reuse}(*p) & = & *p \\
\mathsf{reuse}(\Lambda(\alpha : \mathsf{Typ}).\ u) & = & \Lambda(\alpha : \mathsf{Typ}).\ u \\
\mathsf{reuse}(\mathsf{fix}^\pi\ x\ (y : \tau') : \tau\ .\ a) & = & \mathsf{fix}^\pi\ x\ (y : \tau') : \tau\ .\ a \\
\mathsf{reuse}(\mathsf{let}\ x = a\ \mathsf{in}\ b) & = & \mathsf{let}\ x = \mathsf{reuse}(a)\ \mathsf{in}\ \mathsf{reuse}(b) \\
\mathsf{reuse}(a\ \tau) & = & \mathsf{reuse}(a)\ \tau \\
\mathsf{reuse}(d_i(\tau_j)^j(a_k)^k) & = & d_i(\tau_j)^j(\mathsf{reuse}(a_k))^k \\
\mathsf{reuse}(\mathsf{match}\ a\ \mathsf{with}\ (d_i(\tau_k)^k(x_{ij})^j \to b_i)^i) & & \\
\quad = \mathsf{match}\ \mathsf{reuse}(a)\ \mathsf{with}\ (d_i(\tau_k)^k(x_{ij})^j \to \mathsf{reuse}(b_i))^i & &
\end{array}
$$

Figure 5.3: Making ML terms reusable

Lemma 5.2 (Unique decomposition)**.** *Consider a reducible term a. Then, either a is not head-reducible and there is a unique decomposition as $a = \mathcal{E}[b]$ where b is reducible, or a is head-reducible and there is no such decomposition.*

Proof. Consider the different cases for a.

- Variables x and reused results $*p$ are not reducible.
- Suppose a is $\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$. If a_1 is not a value, the term is not head-reducible, and the only possible reduction context is $\mathcal{E} = \mathsf{let}\ x = []\ \mathsf{in}\ a_2$. Then a_1 must be reducible (otherwise a would not be reducible), and we have $a = \mathcal{E}[a_1]$. If a_1 is a value, a head-reduces. The only possible reduction context is $\mathcal{E} = \mathsf{let}\ x = []\ \mathsf{in}\ a_2$ but it does not produce a decomposition because a_1 is not reducible (by Lemma 5.1).
- We proceed similarly for terms starting with destructors (application, pattern matching).
- Term abstractions are always values and do not reduce.
- For other constructors: we consider the case of data constructors, with $a = d(\tau)^i(a_j)^j$. Since a is reducible, there exists a decomposition of $(a_j)^j = (b_k)^k b(v_\ell)^\ell$ such that b reduces. By Lemma 5.1, b is not a value. Other possible contexts are of the form $\mathcal{E} = d(\tau)^i(b_k)^k b(v_\ell)^{\ell<n}[](v_\ell)^{\ell>n}$ with $a = \mathcal{E}[v_n]$. But v_n is irreducible.

□

Lemma 5.3 (Determinism)**.** *Suppose a is reducible. There exists a unique b and l such that $a \xrightarrow{l}_h b$.*

Proof. By induction on the term, using unique decomposition: if there were two reductions, there would be two decompositions. □

5.3 Full reduction

In this section, we introduce a full reduction for labeled ML. While we will not use it directly, it will serve to explain how the labeling, that seems very

$$(\Lambda(\alpha : \mathsf{Typ}).\, u)\ \tau \overset{0}{\mapsto} u[\alpha \leftarrow \tau]$$

$$(\mathsf{fix}^{\pi}\ x\ (y : \tau_1') : \tau_2'\ .\ a)^p\ u \xmapsto{*p \leftarrow \mathsf{reuse}(a[\pi \leftarrow p, x \leftarrow \mathsf{fix}^{\pi}\ x\ (y:\tau_1'):\tau_2'\ .\ a, y \leftarrow \tau_1'])} a[\pi \leftarrow p, x \leftarrow \mathsf{fix}^{\pi}\ x\ (y : \tau_1') : \tau_2'\ .\ a, y \leftarrow \tau_1']$$

$$\mathsf{let}\ x = u\ \mathsf{in}\ a \overset{0}{\mapsto} a[x \leftarrow u]$$

$$\mathsf{match}\ d_j(\tau_k'')^k(v_i)^i\ \mathsf{with}\ (d_j(\tau_k)^k(x_{ji})^i \rightarrow a_j)^j \overset{0}{\mapsto} a_j[x_{ji} \leftarrow v_i]^i$$

Figure 5.4: Full head reduction for labeled ML

$$\begin{aligned}
\mathcal{C} \ ::=\ & \mathsf{let}\ x = []\ \mathsf{in}\ a \mid \mathsf{let}\ x = a\ \mathsf{in}\ [] \mid a^p\ [] \mid []^p\ a \mid d\,\overline{\tau}(a, ..\, a, [], a, ..\, a) \\
\mid\ & \Lambda(\alpha : \mathsf{Typ}).\, [] \mid \mathsf{fix}^{\pi}\ x\ (x : \tau) : \tau\ .\ [] \mid []\ \tau \\
\mid\ & \mathsf{match}\ []\ \mathsf{with}\ (P \rightarrow a \mid ..\, P \rightarrow a) \mid \mathsf{match}\ a\ \mathsf{with}\ (P \rightarrow [] \mid \ldots P \rightarrow a)
\end{aligned}$$

Figure 5.5: Contexts for full reduction

dependent on the evaluation order of ML, is actually compatible with full reduction. We will build upon the concepts defined here to provide full reduction for meta-abstractions in mML (Chapter 7).

The full reduction allows reduction in any position, including under a lambda. Moreover, instead of requiring values in redexes, we only require terms to be non-expansive. The definition of the head reductions is given in Figure 5.4. Other than requiring non-expansive terms instead of values, they are identical to the head reductions of the weak reduction.

The contexts of the reduction are all shallow contexts $\mathcal{C}$. We enumerate them in Figure 5.5.

There is an additional wrinkle to take care of when performing full reduction. Consider the reduction of $\mathsf{let}\ z = \mathsf{let}\ x = \mathsf{True}\ \mathsf{in}\ (\lambda(y : \mathsf{bool}).\, y)^p\ x\ \mathsf{in}\ *p$. If we start by reducing the abstraction, we need to propagate a result $*p \leftarrow x$. This is fine until we leave the scope of x, but if we continue propagating this to the toplevel, we end up with $\mathsf{let}\ z = \mathsf{let}\ x = \mathsf{True}\ \mathsf{in}\ x\ \mathsf{in}\ x$, which reduces to x. If we follow the weak reduction order on the same term, we end up with True: our reduction is not confluent. We need a way to keep the value of x available in the term, even if x has not been evaluated yet. Fortunately, we have exactly that: when $*p \leftarrow u$ passes through a $\mathsf{let}\ x = a\ \mathsf{in}\ []$ layer, we transform it into $*p \leftarrow u[x \leftarrow \mathsf{reuse}(a)]$.

We encounter similar problems with pattern matching. We can perform the same pattern matching to get the values of the variables in a branch, but we are blocked in other branches: we know locally which branch has been taken, but there is no way to use this information to create terms to fill the other branches. We work around this (until Chapter 6 gives us a better solution) by allowing incomplete pattern matchings.

Lambda abstractions also bind term variables. There is no easy way to recover a value for these variables because lambda abstractions are detached from the term that will be used as argument. For this reason, when reducing inside of a lambda abstraction, we hide the label on the reduction from the context outside of the abstraction: the reduction becomes unlabeled. Lastly, type abstractions contain only non-expansive terms, and reducing non-expansive

$$
\begin{array}{lcl}
(*p \leftarrow u) \uparrow \mathsf{fix}^x\ \tau\ [] : (y : \tau')\ . & = & 0 \\
(*p \leftarrow u) \uparrow \mathsf{match}\ a\ \mathsf{with}\ P \to [] \mid \ldots & = & *p \leftarrow \mathsf{match}\ \mathsf{reuse}(a)\ \mathsf{with}\ P \to u \\
(*p \leftarrow u) \uparrow \mathsf{let}\ x = a\ \mathsf{in}\ [] & = & *p \leftarrow u[x \leftarrow \mathsf{reuse}(a)] \\
(*p \leftarrow u) \uparrow \mathsf{let}\ x = []\ \mathsf{in}\ a & = & *p \leftarrow u \\
(*p \leftarrow u) \uparrow a^q\ [] & = & *p \leftarrow u \\
(*p \leftarrow u) \uparrow []^q\ a & = & *p \leftarrow u \\
(*p \leftarrow u) \uparrow d\,\overline{\tau}\,((a_i)^i, [], (a_j)^j) & = & *p \leftarrow u \\
(*p \leftarrow u) \uparrow []\ \tau & = & *p \leftarrow u \\
(*p \leftarrow u) \uparrow \mathsf{match}\ []\ \mathsf{with}\ (P \to a \mid ..\, P \to a) & = & *p \leftarrow u
\end{array}
$$

Figure 5.6: Translation

$$
\textsc{Red-NoLabel}\ \frac{a \overset{0}{\mapsto} b}{\mathcal{C}[a] \overset{0}{\mapsto} \mathcal{C}[b]}
\qquad
\textsc{Red-Label}\ \frac{a \overset{*p \leftarrow u}{\longmapsto} b}{\mathcal{C}[a] \overset{(*p \leftarrow u) \uparrow \mathcal{C}}{\longmapsto} (\mathcal{C}[*p \leftarrow u])[b]}
$$

Figure 5.7: Full reduction

terms will not produce a label.

We express these transformations on labels by defining a translation function $(*p \leftarrow u) \uparrow \mathcal{C}$ that returns either a new label $*p \leftarrow u'$ or 0. Its definition is given in Figure 5.6. For weak reduction contexts (*i.e.* the last six cases), the label is propagated as-is, which is consistent with the fact that we did not need label translation in weak reduction contexts.

Then, the full reduction is defined using these two functions in Figure 5.7. It generalizes the previous reduction:

Lemma 5.4 (Superset)**.** *Suppose $a \xrightarrow{l} b$. Then $a \overset{l}{\mapsto} b$.*

Proof. The evaluation contexts and redexes of the weak reduction are a subset of the evaluation contexts and redexes of the full reduction, the weak head reduction is a subset of the full head reduction, and the context rules are identical for evaluation contexts. □

The reduction preserves non-expansivity. Moreover, reduction of non-expansive terms does not produce a label:

Lemma 5.5 (Non-expansivity)**.** *Suppose $u \overset{l}{\mapsto} a$. Then a is non-expansive and $l = 0$.*

Proof. By induction on the contexts. Contexts that yield non-expansive terms either have a non-expansive term in the hole or remove the label. Non-expansive redexes do not produce a label when reducing. □

5.4 An attempt at well-labelling

Unfortunately, the full reduction is not confluent. There are three causes for this. The first cause is that we can have two redexes with the same label.

Depending on which we reduce first, we get two different results. Consider for example the following two reduction paths from the same term:

$$\begin{array}{l} \quad (*p, (\lambda^\pi x.\, x)^p\ 1, (\lambda^\pi y.\, y^p\ )2) \\ \xmapsto{*p\leftarrow 1} (1, 1, (\lambda^\pi y.\, y)^p\ 2) \\ \xmapsto{*p\leftarrow 2} (1, 1, 2) \end{array} \qquad \begin{array}{l} \quad (*p, (\lambda^\pi x.\, x)^p\ 1, (\lambda^\pi y.\, y)^p\ 2) \\ \xmapsto{*p\leftarrow 2} (2, (\lambda^\pi x.\, x)^p\ 1, 2) \\ \xmapsto{*p\leftarrow 1} (2, 1, 2) \end{array}$$

Secondly, depending on the order of evaluation, a label may or may not be substituted. Consider the following example, where $*1 \cdot 1$ is substituted or not depending on whether we start by reducing the outer or the inner thunk:

$$\begin{array}{l} (*1 \cdot 1, (\lambda^\pi y.\, (\lambda^\pi z.\, x)^{\pi\cdot 1}\ ())^1\ ()) \\ \xmapsto{*1\leftarrow *1\cdot 1} (*1 \cdot 1, (\lambda^\pi z.\, x)^{1\cdot 1}\ ()) \\ \xmapsto{*1\cdot 1\leftarrow x} (x, x) \end{array} \qquad \begin{array}{l} (*1 \cdot 1, (\lambda^\pi y.\, (\lambda^\pi z.\, x)^{\pi\cdot 1}\ ())^1\ ()) \\ \xmapsto{\varnothing} \quad (*1 \cdot 1, (\lambda^\pi y.\, x)^1\ ()) \\ \xmapsto{*1\leftarrow x} (*1 \cdot 1, x) \end{array}$$

Third, under branching constructs, evaluation might simply discard the expression generating a label before it is substituted: for example, in the following example, the label 1 is generated in a dead branch. But nothing prevents the full reduction from evaluating inside this dead branch:

$$\begin{array}{l} (*1, \mathsf{if\ False\ then}\ (\lambda^\pi z.\, x)^1\ 2\ \mathsf{else}\ 2) \\ \xmapsto{*1\leftarrow 2} (2, \mathsf{if\ False\ then}\ 2\ \mathsf{else}\ 2) \\ \xmapsto{\varnothing} \quad (2, 2) \end{array} \qquad \begin{array}{l} (*1, \mathsf{if\ False\ then}\ (\lambda^\pi z.\, x)^1\ 2\ \mathsf{else}\ 2) \\ \xmapsto{\varnothing} (*1, 2) \end{array}$$

We can solve the first two issues by defining a set of well-labeled terms, stable by reduction. The third issue is solved by the introduction of eML in Chapter 6. We have two constraints. First, the labels produced by the reductions must be distinct. We cannot merely ask the labels present in a term to be distinct, because reduction of expansive redexes creates new redexes. To control the new labels, we require that the labels generated by reduction of a redex labeled p have labels prefixed by p. Then, a term should produce a set of labels such that no label is a prefix of another: this guarantees that replacing p with a set $(p \cdot n_i)^i$ will never create a conflict. Second, the order of evaluation should not influence when a label gets substituted. This is a constraint on the uses of labels: we must be certain that whatever the order of evaluation, a label will be substituted, or that whatever the order of evaluation it won't be. The second case is useful for subject reduction, because we need to handle the case where the expression producing a label was under a branch that disappeared during reduction.

To control the unicity of label, we need to define *orthogonal* sets of paths.

Definition 5.1 (Prefix)**.** *A path p is a prefix of q, noted $p \leqslant q$, if there a exists a (possibly empty) sequence of labels $(n_i)^i$ such that $q = p \cdot n_1 \cdot \ldots \cdot n_n$. This defines a partial order on the set of paths.*

If neither $p \leqslant q$ nor $q \leqslant p$, we say that p and q are orthogonal, noted $p \perp q$. ♢

Definition 5.2 (Orthogonal set, union)**.** *A set S of paths is orthogonal, noted* orthogonal(S)*, if there are no distinct elements p and q such that $p \leqslant q$.*

If S_1 and S_2 are orthogonal, and for all $p_1 \in S_1, p_2 \in S_2$, $p_1 \perp p_2$, then S_1 and S_2 are said to be orthogonal, and there union $S_1 \cup S_2$ is an orthogonal set. ♢

$$\textsc{L-Var}\quad \frac{\text{orthogonal}(\Gamma)}{\Sigma;\Gamma \vdash_l^p x \Rightarrow \varnothing} \qquad \textsc{L-TAbs}\quad \frac{\Sigma;\Gamma \vdash_l^p u \Rightarrow \varnothing}{\Sigma;\Gamma \vdash_l^p \Lambda(\alpha : \mathsf{Typ}).\, u \Rightarrow \varnothing} \qquad \textsc{L-TApp}\quad \frac{\Sigma;\Gamma \vdash_l^p a \Rightarrow \Delta}{\Sigma;\Gamma \vdash_l^p a\ \tau \Rightarrow \Delta}$$

$$\textsc{L-Fix}\quad \frac{\pi \# \Sigma \qquad \Sigma,\pi;\Gamma \vdash_l^\pi a \Rightarrow \Delta}{\Sigma;\Gamma \vdash_l^p \mathsf{fix}^\pi\ x\ (y : \tau_1) : \tau_2\ .\ a \Rightarrow \varnothing}$$

$$\textsc{L-App}\quad \frac{\Sigma;\Gamma \vdash_l^p a \Rightarrow \Delta_1 \qquad \Sigma;\Gamma,\Delta_1 \vdash_l^p b \Rightarrow \Delta_2 \qquad p \leqslant q \qquad q \perp \Gamma,\Delta_1,\Delta_2}{\Sigma;\Gamma \vdash_l^q a^q\ b \Rightarrow \Delta_1,\Delta_2,q}$$

$$\textsc{L-Let}\quad \frac{\Sigma;\Gamma \vdash_l^p a \Rightarrow \Delta_1 \qquad \Sigma;\Gamma,\Delta_1 \vdash_l^p b \Rightarrow \Delta_2}{\Sigma;\Gamma \vdash_l^p \mathsf{let}\ x = a\ \mathsf{in}\ b \Rightarrow \Delta_1,\Delta_2}$$

$$\textsc{L-Con}\quad \frac{\text{orthogonal}(\Gamma) \qquad (\Sigma;\Gamma,(\Delta_k)^{k>j} \vdash_l^p a_j \Rightarrow \Delta_j)^j}{\Sigma;\Gamma \vdash_l^p d(\tau_i)^i(a_j)^j \Rightarrow (\Delta_j)^j}$$

$$\textsc{L-Match}\quad \frac{\Sigma;\Gamma \vdash_l^p a \Rightarrow \Delta \qquad (\Sigma;\Gamma,\Delta \vdash_l^p a_i \Rightarrow \Delta_i)^i \qquad \text{orthogonal}(\Delta_i)^i}{\Sigma;\Gamma \vdash_l^p \mathsf{match}\ a\ \mathsf{with}\ (P_i \to a_i)^i \Rightarrow \Delta,(\Delta_i)^i}$$

$$\textsc{L-Reuse}\quad \frac{\text{orthogonal}(\Gamma) \qquad p \in \Gamma \vee p \perp \Gamma}{\Sigma;\Gamma \vdash_l^p *p \Rightarrow \varnothing}$$

Figure 5.8: Well-labeling

We define a well-labeling judgment $\Sigma; \Gamma \vdash_l^p a \Rightarrow \Delta$, asserting that under an environment with label variables Σ and that produces the set of labels Γ, and when the current prefix for introducing new labels is p, the term a is well-labeled and may produce labels Δ. The rules of the judgment are given in Figure 5.8. They are essentially a miniature type-system that is only concerned with the existence of labels. Information flows following the evaluation order: for example, in application (L-App), the labels obtained evaluating the left-hand expression are added as input to the right-hand expression. The well-labeling judgment is only very weakly concerned with the existence of reused labels: it just wants them to be unambiguously bound, or unambiguously unbound. The latter case can happen if a reduction removes the code that would have emitted the label. A latter reduction might then remove the use of the label. For this reason, we allow reused labels (L-Reuse) to be either in Γ or independent from Γ.

The rules maintain some invariants.

Lemma 5.6 (Scoping of path variables)**.** *Suppose* $\Sigma; \Gamma \vdash_l^p a \Rightarrow \Delta$. *Then,* Σ *does not contain duplicate variables, and all variables occurring in* Γ, Δ, p, a *are in* Σ.

We write the reduction rules such that if $\Sigma; \Gamma \vdash_l^p a \Rightarrow \Delta$, Γ and Δ are orthogonal. This is enforced by requiring Γ to be orthogonal in L-Var, and requiring the different outputs to be orthogonal in L-Match. The rest is implicitly enforced:

Lemma 5.7 (Disjoint labels)**.** *Suppose* $\Sigma; \Gamma \vdash_l^p a \Rightarrow \Delta$. *Then,* $\Gamma \perp \Delta$ *and all labels in* Δ *are prefixed by* p.

Proof. Γ is orthogonal: all derivation rules have a premise whose set of input labels is Γ or a superset of Γ (then, proceed by induction), except L-Var where we check explicitly that Γ is orthogonal.

$\Gamma \perp \Delta$: the only rule that grows $\Gamma \cup \Delta$ is L-App, and L-App checks that the union of input and output labels remains independent.

Δ is orthogonal: for most rules, Δ is obtained by the union of part of the input of a premise and its output. L-App checks that the new label is independent of the already-existing output. L-Match explicitly checks that the output is orthogonal.

All labels in Δ are prefixed by p: labels are only introduced by L-App and it checks that the label is prefixed by p. □

Lemma 5.8 (Non-expansivity)**.** *Suppose* $\Sigma; \Gamma \vdash_l^p a \Rightarrow \Delta$. *If* a *is non-expansive, then* $\Delta = \varnothing$, *and for all* q, $\Sigma; \Gamma \vdash_l^q a \Rightarrow \varnothing$.

Proof. By induction on the derivation. No rules can inspect p or add something in Δ until passing through L-Fix because function application is expansive, and L-Fix erases the label and Δ. □

We need to prove weakening and substitution:

Lemma 5.9 (Weakening)**.** *Suppose* $\Sigma; \Gamma \vdash_l^p a \Rightarrow \Delta$. *Then:*

- *Suppose* $q \leqslant p$, *and* q *independent of* Γ, Δ. *Then,* $\Sigma; \Gamma \vdash_l^q a \Rightarrow \Delta$.
- *Suppose* $\pi \notin \Sigma$. *Then,* $\Sigma, \pi; \Gamma \vdash_l^p a \Rightarrow \Delta$.

- *Suppose* $q \perp p, \Gamma, \Delta$. *Then,* $\Sigma; \Gamma, q \vdash_l^p a \Rightarrow \Delta$.

Proof. By induction on the derivation: these changes translate to weakening in the premises. The independence side-conditions stay true in the derivations. □

Lemma 5.10 (Substitution)**.** *Suppose* $\Sigma; \Gamma \vdash_l^p a \Rightarrow \Delta$.

- *Consider* $\pi \in \Sigma$, *and* $q \perp \Gamma, \Delta$. *Then,* $\Sigma - \pi; \Gamma[\pi \leftarrow q] \vdash_l^{p[\pi \leftarrow q]} a[\pi \leftarrow q] \Rightarrow \Delta[\pi \leftarrow q]$.

- *Consider* x *and a non-expansive term* u *such that* $\Sigma; \Gamma \vdash_l^p u \Rightarrow \varnothing$. *Then,* $\Sigma; \Gamma \vdash_l^p a[x \leftarrow u] \Rightarrow \Delta$.

Proof. By induction on the derivation. □

We can prove that well-labeling is preserved by reduction (this is a form of subject reduction). Reduction can emit a label, which changes the set of possibly emitted labels: the emitted label p must be in Δ, and it is replaced by a set of labels prefixed by p. If no label is emitted, Δ does not change.

Lemma 5.11 (Subject reduction for well-labeling)**.** *Suppose* $\Sigma; \Gamma \vdash_l^p a \Rightarrow \Delta$. *Moreover, suppose* $a \overset{l}{\mapsto} b$. *Then:*

- *If* $l = 0$, *we have* $\Sigma; \Gamma \vdash_l^p b \Rightarrow \Delta'$ *where* Δ' *is a subset of* Δ.

- *If* $l = *q \leftarrow u$, *we have* $\Sigma; \Gamma \vdash_l^p b \Rightarrow \Delta'$ *where* $\Delta' = \Delta - \{q\} \uplus S$ *where* S *is an orthogonal set of labels prefixed by* p.

Proof. Let us first prove the result for head reductions:

- For reduction of an application $(\mathsf{fix}^\pi\, x\, (y : \tau) : \tau' \,.\, a)^q\, u$: the application reduces to $a[\pi \leftarrow q, x \leftarrow (\mathsf{fix}^\pi\, x\, (y : \tau) : \tau' \,.\, a), y \leftarrow u]$, with label $l = *q \leftarrow \mathsf{reuse}(a[x \leftarrow u])$. Inverting rule L-App, we obtain that $p \leqslant q$ and there exists Δ_1 and Δ_2 such that $\Delta = \Delta_1 \uplus \Delta_2 \uplus \{q\}$, with $\Sigma; \Gamma \vdash_l^p \mathsf{fix}^\pi\, x(y : \tau) : \tau' \,.\, a \Rightarrow \Delta_1$ and $\Sigma; \Gamma, \Delta_1 \vdash_l^p u \Rightarrow \Delta_2$. Since both $\mathsf{fix}^\pi\, x\, (y : \tau) : \tau' \,.\, a$ and u are non-expansive, $\Delta_1 = \Delta_2 = \varnothing$ (Lemma 5.8). Further inverting the first subderivation (that must use L-Fix), we obtain that there exists Δ_3 such that: $\Sigma, \pi; \Gamma \vdash_l^\pi a \Rightarrow \Delta_3$. By substitution of term variables and path variables (Lemma 5.10), we have $\Sigma; \Gamma[\pi \leftarrow q] \vdash_l^q a[\pi \leftarrow q, x \leftarrow (\mathsf{fix}^\pi\, x\, (y : \tau) : \tau' \,.\, a), y \leftarrow u] \Rightarrow \Delta_3[\pi \leftarrow q]x$. The path variable π does not appear in Γ (the free variables of Γ are included in Σ, thus $\Gamma[\pi \leftarrow q] = \Gamma$. By Lemma 5.7, all labels of $\Delta_3[\pi \leftarrow q]$ are prefixed by q. By weakening of the path prefix (Lemma 5.9), we have $\Sigma; \Gamma \vdash_l^p a[\pi \leftarrow q, x \leftarrow (\mathsf{fix}^\pi\, x\, (y : \tau) : \tau' \,.\, a), y \leftarrow u] \Rightarrow \Delta_3[\pi \leftarrow q]$, which is the result we are looking for (we have $\Delta = \{q\}$, and $S = \Delta_3[\pi \leftarrow q]$).

- Consider the reduction of a type application: $(\Lambda(.\, \alpha : \mathsf{Typ})u)\, \tau$ reduces to $u[\alpha \leftarrow \tau]$ with label 0. Inverting the last rule (L-TApp), we get $\Sigma; \Gamma \vdash_l^p \Lambda(\alpha : \mathsf{Typ}).\, u \Rightarrow \varnothing$ and $\Delta = \varnothing$. Inverting rule L-TAbs, we get $\Sigma; \Gamma \vdash_l^p u \Rightarrow \varnothing$. By substitution of the type variable (Lemma 5.10), $\Sigma; \Gamma \vdash_l^p u[\alpha \leftarrow \tau] \Rightarrow \varnothing$,

- Consider the reduction of a let binding: $\mathsf{let}\ x = u\ \mathsf{in}\ a$ reduces to $a[x \leftarrow u]$ with label 0. By inverting the last derivation, we get that there exists Δ_1, Δ_2 such that $\Delta = \Delta_1 \uplus \Delta_2$, and $\Sigma; \Gamma \vdash_l^p u \Rightarrow \Delta_1$ and $\Sigma; \Gamma, \Delta_1 \vdash_l^p a \Rightarrow \Delta_2$. Since u is non-expansive, $\Delta_1 = \varnothing$ and $\Delta = \Delta_2$. Then, by substitution (Lemma 5.10), $\Sigma; \Gamma \vdash_l^p a[x \leftarrow u] \Rightarrow \Delta_2$.

- Consider reduction of a pattern matching:

$$\mathsf{match}\ d_j(\tau''_k)^k(u_i)^i\ \mathsf{with}\ (d_j(\tau_k)^k(x_{ji})^i \to a_j)^j$$

  It reduces with label 0 to $a_j[x_{ji} \leftarrow u_i]^i$. Let us invert the rule L-Match. There exists $\Delta', (\Delta_j)^j$ such that $\Delta = \Delta' \uplus (\Delta_j)^j$, $\Sigma; \Gamma \vdash_l^p d_j(\tau''_k)^k(v_i)^i \Rightarrow \Delta'$, and, for all j, $\Sigma; \Gamma, \Delta' \vdash_l^p a_j \Rightarrow \Delta_j$. We have $\Delta = \varnothing$ because $d_j(\tau''_k)^k(v_i)^i$ is non-expansive. By inverting the rule L-Con, we get that, for all i, $\Sigma; \Gamma \vdash_l^p u_i \Rightarrow \varnothing$. By substitution, $\Sigma; \Gamma \vdash_l^p a_j[x_{ji} \leftarrow u_i]^i \Rightarrow \Delta_j$. This satisfies subject reduction, since Δ_j is a subset of Δ.

Then, we proceed by induction on the reduction context. Let us consider a few different cases for context (other cases are similar):

- Suppose $\mathcal{C} = \mathsf{let}\ x = [\,]\ \mathsf{in}\ b$, and $a_0 \overset{l}{\mapsto} a_1$. Then, $\mathsf{let}\ x = a_0\ \mathsf{in}\ b \overset{l}{\mapsto} \mathsf{let}\ x = a_0\ \mathsf{in}\ b[l]$. Suppose $\mathsf{let}\ x = a_0\ \mathsf{in}\ b$ is well-labeled: then the last rule of the well-labeling derivation is L-Let and there exists $\Sigma, \Gamma, p, \Delta_1, \Delta_2$ such that $\Sigma; \Gamma \vdash_l^p a_0 \Rightarrow \Delta_1$, $\Sigma; \Gamma, \Delta_1 \vdash_l^p b \Rightarrow \Delta_2$, and $\Sigma; \Gamma \vdash_l^p \mathsf{let}\ x = a_0\ \mathsf{in}\ b \Rightarrow \Delta_1, \Delta_2$. By induction hypothesis, there exists Δ'_1 such that $\Sigma; \Gamma \vdash_l^p a_1 \Rightarrow \Delta'_1$. We have $\Delta'_1 \perp \Delta_2$: this is true for Δ_1 and the new elements of Δ'_1 are suffixes of elements of Δ_1: thus, if an element of Δ'_1 is a prefix of an element of Δ_2, then an element of Δ_1 was a prefix of an element of Δ_2. Conversely, if an element of Δ_2 is a prefix of an element of Δ'_1, either it is a prefix or a suffix of the corresponding element of Δ_1. Moreover, we have $\Sigma; \Gamma, \Delta'_1 \vdash_l^p b[l] \Rightarrow \Delta_2$. This is true if $\Delta_1 = \Delta'_1$. Otherwise, the emitted label is q. By substitution, we have $\Sigma; \Gamma, \Delta_1 - q \vdash_l^{b[l]} \Delta_2 \Rightarrow$. Consider p such that p is either included in Δ_1 or independent of Δ_1. If p is q, it is eliminated by substitution. Otherwise, p is either included in Δ'_1 or independent of Δ'_1, as Δ'_1 is a subset of $\Delta_1 - q$, and we obtain the result by weakening. Thus, we have $\Sigma; \Gamma \vdash_l^p \mathsf{let}\ x = a_1\ \mathsf{in}\ b[l] \Rightarrow \Delta'_1, \Delta_2$.

- Suppose $\mathcal{C} = \mathsf{fix}^\pi\ x\ (y : \tau') : \tau\ .\ []$. Suppose $a_0 \overset{l}{\mapsto} a_1$. Then, $\mathsf{fix}^\pi\ x\ (y : \tau') : \tau\ .\ a_0 \overset{\varnothing}{\longmapsto} \mathsf{fix}^\pi\ x\ (y : \tau') : \tau\ .\ a_1$. Suppose $\mathsf{fix}^\pi\ x\ (y : \tau') : \tau\ .\ a_0$: there exists Γ, p, Δ such that $\Sigma, \pi; \Gamma \vdash_l^\pi a_0 \Rightarrow \Delta$ and $\Sigma; \Gamma \vdash_l^p \mathsf{fix}^\pi\ x(y : \tau') : \tau\ .a_0 \Rightarrow \varnothing$. By induction hypothesis, there exists Δ'_1 such that $\Sigma; \Gamma \vdash_l^p a_1 \Rightarrow \Delta'_1$. Then, $\Sigma; \Gamma \vdash_l^p \mathsf{fix}^\pi\ x\ (y : \tau') : \tau\ .\ a_1 \Rightarrow \varnothing$.

□

As mentioned before, the full reduction is not confluent, even on well-labeled terms. We could prove that two reduction paths either converge or one of them gets blocked on an unbound label. Instead, we will use equalities to develop a richer type system that allows us to encode precisely when labels are available in Chapter 6.

Chapter 6

A language for equalities

In this chapter, we construct an extension of ML with type-level pattern matching, dependent on the variables in the context: for example, a term of type $\mathsf{match}\ x\ \mathsf{with}\ \mathsf{True} \to \tau \mid \mathsf{False} \to \sigma$ must have a value of type τ if x is true, and of type σ otherwise. To allow the construction of terms of these types, term-level pattern matching introduces an equality witnessing the information learned by matching the term: the term that is matched upon is equal to the pattern of the current branch. Then, this equality (for instance $x = \mathsf{True}$) can be used to reduce type-level pattern matching: under this equality, the type $\mathsf{match}\ x\ \mathsf{with}\ \mathsf{True} \to \tau \mid \mathsf{False} \to \sigma$ is equivalent to τ, and terms with these types are freely interconvertible.

This extension, called eML, is carefully designed so that it can be simplified back to ML by some well-identified code transformations.

6.1 Design constraints

The introduction of eML is motivated by the fact that, when doing ornamentation, the programmer decorates a value of the base type with some supplementary information that will be used to construct a value of the ornamented type. The type of this supplementary information depends on the original value: for example, when lifting natural numbers to lists, one has to provide an element for lifting S into a Cons, but no information (*i.e.* a value of type unit) is needed for lifting Z into Nil. In that case, the supplementary information depends only on the constructor. In general, it could depend on information found deeper into the term: for example, one could add a value to Some True and leave Some False as-is.

Type-level pattern matching provides an expressive way of handling the typing of the supplementary information: the ornament writer can use an arbitrarily deep (but finite) view of the base term to determine the appropriate type for the patch.

To write a term of the appropriate type, the writer of the patch can use the local information in the function: for example, if the patch is to be applied below a pattern matching Z, the type of the information required is equivalent to unit. This can be proved to the type system using an equality introduced in the context by the pattern matching, witnessing the branch that is currently

being executed. It is important to the design of eML that this information is local, *i.e.* is not propagated through function calls and returns: one key requirement on eML is that we still want to be able to output ML programs after ornamentation. Exploiting the fact that the information used to interpret type is local, it is possible (under some typing conditions) to simplify any eML term to an equivalent ML term.

The equivalence between types defined by eML depends on an equivalence between terms: one can reduce a type-level pattern matching (such as match a with Z → unit | S _ → bool) when one knows that, in the current context, the term that is matched is equivalent to a term of the shape of one of the branches (for example, a being equivalent to Z). This equivalence between terms needs to be appropriately limited: first, it is more convenient if it is decidable (this allows, for example, to easily eliminate dead branches in eML code when translating it to ML). A simple way to do that is to not recognize the equality between a function application and its reduction. A second reason to not recognize this equality is that, in OCaml, functions may have side-effects. If an application returns True the first time it is called, there is no guarantee that it will not return False the second time. The presence of side-effects also means that it is not sound to replace a call to a function by its result, since the call may have other effects. Equalities between an application and its result may be introduced as a witness of the result of a pattern matching: match rand_bool Unit with True → a | False → b introduces when typing a an equality between rand_bool Unit and True. A pattern matching on the same expression could introduce the equality between rand_bool Unit and False. Taken together, these two equalities give the equivalence of True and False, which notably proves the equivalence of any two types.

The equalities we actually mean to introduce in this case are that a particular call to rand_bool Unit returned a particular boolean, without any implications on the result of other evaluations of this expression. We express this by using the *labels* that we introduced to refer to the result of previous applications: each application executed in the program is attributed a different label, and one can refer using this label to the result of this specific application.

6.2 Description of eML

This section builds on the presentation of labeled ML (§5). We combine the typing rules of ML with the well-labeling rules of labeled ML, and add *constraints* on labels to ensure that well-typed terms do not get stuck on unsubstituted labels (Lemma 6.24).

6.2.1 Extended syntax

The syntax of eML is identical to the syntax of labeled ML, except for the introduction of type-level pattern matching. The new syntax of types is given on Figure 6.1: type-level pattern matching is identical to term-level pattern matching, except that it can only match on non-expansive terms, and the branches are types instead of terms. For example, match u with True → unit | False → bool is a valid eML type.

$$
\begin{array}{lll}
\tau, \sigma \ ::= & & \text{Types} \\
\quad | & \alpha & \text{Type variable} \\
\quad | & \tau \to \tau & \text{Function type} \\
\quad | & \zeta\ \overline{\tau} & \text{Datatype} \\
\quad | & \forall(\alpha : \mathsf{Typ})\ \tau & \text{Universal quantification} \\
\quad | & \mathsf{match}\ u\ \mathsf{with}\ \overline{P \to \tau} & \text{Pattern matching}
\end{array}
$$

Figure 6.1: Types and kinds for eML

$$\mathsf{match}\ d_j(\tau''_k)^k(v_i)^i\ \mathsf{with}\ (d_j(\tau_k)^k(x_{ji})^i \to \sigma_j)^j \overset{0}{\mapsto} \sigma_j[x_{ji} \leftarrow v_i]^i$$

Figure 6.2: Additional head reduction rule for eML

6.2.2 Extended labeled reduction

The weak reduction stays the same as in labeled ML: it only reduces the terms, whose syntax is unchanged.

Since redexes can now appear in types, it is necessary to extend the full reduction to also reduce these types: this introduces new (non-expansive) evaluation contexts and places where a potential substitution is allowed. The additional reduction rule is given in Figure 6.2: it is the reduction of a type-level pattern matching. It is identical to the reduction of a term-level pattern matching, except in that the branches are types. The other rules are identical to those given in Figure 5.7. The modified contexts are given in Figure 6.3. Since none of the new contexts may contain expansive expressions, the translation function is identical to its definition in Figure 5.6.

We could also define a well-labeling relation for eML terms. Instead, we provide a typing judgment that refines the well-labeling judgment.

6.2.3 Combining typing and labeling

The definitions of datatypes stay unchanged. In particular, we require that they do not include type-level pattern matching. This is necessary so that they are still ML datatypes, so that we can simplify eML types to ML types.

The typing judgment for eML combines the typing of ML with the well-labeling judgment of labeled ML. The syntax of typing environments is given on Figure 6.4. They include term and type variables as in ML, but also path variables π and input labels $(u)*p : \tau$. Input labels are guarded with a non-expansive term u: this is a boolean expression that guards the use of the label. The typing judgment will ensure that the guard condition is true whenever we try to refer to a label. We also indicate the type τ of the computed result. The typing environments are also extended with witnesses of equalities between non-expansive terms u_1 and u_2 at a type σ, written $(u_1 \simeq u_2) : \sigma$. The equalities are introduced and used implicitly, thus are unnamed.

$$\mathcal{C}\ ::=\ \ldots \mid d(\tau, .. \tau, [], \tau, .. \tau)(a, .. a) \mid \mathsf{fix}^{\pi}\, x\ (x : \tau) : []\ .\ a \mid \mathsf{fix}^{\pi}\, x\ (x : []) : \tau\ .\ a$$

Figure 6.3: Additional contexts for eML reduction

Γ ::=		Typing environments
	$\mid \varnothing$	Empty
	$\mid \Gamma, x : \tau$	Variable
	$\mid \Gamma, \alpha : \mathsf{Typ}$	Type variable
	$\mid \Gamma, \pi$	Path variable
	$\mid \Gamma, (u)*p : \tau$	Precomputed result
	$\mid \Gamma, (u \simeq u) : \tau$	Equality
Δ ::=		Computed applications
	$\mid \varnothing$	Empty
	$\mid \Delta, (u)*p : \tau$	Result

Figure 6.4: Typing environments for $e\mathsf{ML}$

The typing judgment is of the form $\Gamma \vdash^p a : \tau \Rightarrow \Delta$. It can be read similarly to a ML typing judgment $\Gamma \vdash a : \tau$ by ignoring the parts related to labeling: under context Γ, the term a has type τ. It can also be read as well-labeling judgment $\Sigma; \Gamma \vdash_l^p a \Rightarrow \Delta$ where Σ is the set of label variables defined in Γ, and Γ represents only the labels that are defined in Γ (Γ plays both roles in the $e\mathsf{ML}$ judgment), and under a path prefix p, the term a emits the labels specified in Δ. The output specification Δ is similar to Δ in the well-labeling judgment. Its syntax is defined on Figure 6.4: it is a subset of the syntax of Γ where only precomputed results are allowed. As with Γ, it associates to each output label p a condition u, and the type τ of the computed result.

For a non-expensive term u, if $\Gamma \vdash^p u : \tau \Rightarrow \Delta$, the set of computed results Δ is always empty, and p is irrelevant. In this case, we may instead write $\Gamma \vdash u : \tau$.

The definition of the typing judgment is given in Figure 6.5. It follows the structure of both the ML typing judgment and the well-labeling judgment.

We need to keep track of which abstractions have been reduced so we can reduce the results. This is done through the output Δ of the typing judgment: Δ contains a list of paths and the type of the result of the application. This is not enough to guarantee that a term can be reused: in code with branches, whether or not an application has actually been evaluated may depend on which branch was taken. For instance, in $\mathsf{let}\ x = \mathsf{match}\ u\ \mathsf{with}\ \mathsf{True} \to \mathsf{True} \mid \mathsf{False} \to f^p\ y\ \mathsf{in}\ a$, when typing a, a subexpression $*p$ would be valid if and only if u were equal to False. This condition is added (in Match) to Δ: a typing of $b = \mathsf{match}\ u\ \mathsf{with}\ \mathsf{True} \to \mathsf{True} \mid \mathsf{False} \to f^p\ y$ could be $\Gamma \vdash^\pi b : \mathsf{bool} \Rightarrow (\mathsf{match}\ u\ \mathsf{with}\ \mathsf{True} \to \mathsf{False} \mid \mathsf{False} \to \mathsf{True})*p : \mathsf{bool}$, indicating that $*p$ is valid only when the expression $\mathsf{match}\ u\ \mathsf{with}\ \mathsf{True} \to \mathsf{False} \mid \mathsf{False} \to \mathsf{True}$ is equal to True, *i.e.* when u is equal to False. This mechanism works well for matching on non-expansive terms. Matching on expansive terms is problematic, because we need the condition u to be non-expansive. We exploit the notion of reusable terms defined in Chapter 5: since the constraint will only be used after the evaluation of the abstraction in the expansive term, $\mathsf{reuse}(a)$ is a valid non-expansive substitute for a. Similarly, the type of the introduced path may use variables that are only bound in the place where the application is present in the term. To be able to use these types somewhere else, we must transform them as we leave the contexts of pattern matchings and let bindings: this is done in

$$\textsc{Var}\quad \frac{\vdash \Gamma \qquad x : \sigma \in \Gamma}{\Gamma \vdash^p x : \sigma \Rightarrow \varnothing}$$

$$\textsc{TAbs}\quad \frac{\Gamma, \alpha : \mathsf{Typ} \vdash^p u : \sigma \Rightarrow \varnothing}{\Gamma \vdash^p \Lambda(\alpha : \mathsf{Typ}).\, u : \forall(\alpha : \mathsf{Typ})\, \sigma \Rightarrow \varnothing}$$

$$\textsc{TApp}\quad \frac{\Gamma \vdash \tau : \mathsf{Typ} \qquad \Gamma \vdash^p a : \forall(\alpha : \mathsf{Typ})\, \sigma \Rightarrow \Delta}{\Gamma \vdash^p a\ \tau : \sigma[\alpha \leftarrow \tau] \Rightarrow \Delta}$$

$$\textsc{Fix}\quad \frac{\Gamma, \pi, x : \tau_1 \to \tau_2, y : \tau_1 \vdash^\pi a : \tau_2 \Rightarrow \Delta}{\Gamma \vdash^p \mathsf{fix}^\pi\, x\, (y : \tau_1) : \tau_2\ .\, a : \tau_1 \to \tau_2 \Rightarrow \varnothing}$$

$$\textsc{App}\quad \frac{\Gamma \vdash^p a : \tau_1 \to \tau_2 \Rightarrow \Delta_1 \qquad \Gamma, \Delta_1 \vdash^p b : \tau_1 \Rightarrow \Delta_2 \qquad p \leqslant q \qquad q \perp \Gamma, \Delta_1, \Delta_2}{\Gamma \vdash^p a^q\ b : \tau_2 \Rightarrow \Delta_1, \Delta_2, (\mathsf{True}) {*} q : \tau_2}$$

$$\textsc{Let}\quad \frac{\Gamma \vdash^p a : \tau' \Rightarrow \Delta_1 \qquad \Gamma, \Delta_1, x : \tau', (x \simeq \mathsf{reuse}(a)) : \tau' \vdash^p b : \tau \Rightarrow \Delta_2}{\Gamma \vdash^p \mathsf{let}\ x = a\ \mathsf{in}\ b : \tau \Rightarrow \Delta_1, \Delta_2[x \leftarrow \mathsf{reuse}(a)]}$$

$$\textsc{Con}\quad \frac{\begin{array}{c}\vdash \Gamma \qquad \vdash d : \forall(\alpha_j : \mathsf{Typ})^j\ (\tau_i)^i \to \zeta\ (\alpha_j)^j \\ (\Gamma \vdash \tau_j : \mathsf{Typ})^j \qquad (\Gamma, (\Delta_k)^{k>i} \vdash^p a_i : \tau_i[\alpha_j \leftarrow \tau_j]^j \Rightarrow \Delta_i)^i\end{array}}{\Gamma \vdash^p d(\tau_j)^j (a_i)^i : \zeta\ (\tau_j)^j \Rightarrow (\Delta_i)^i}$$

$$\textsc{Match}\quad \frac{\begin{array}{c}\vdash (d_i)^i : \zeta\ \mathsf{complete} \\ \Gamma \vdash \tau : \mathsf{Sch} \qquad (d_i : \forall(\alpha_k : \mathsf{Typ})^k\ (\tau_{ij})^j \to \zeta\ (\alpha_k)^k)^i \qquad \Gamma \vdash^p a : \zeta\ (\tau_k)^k \Rightarrow \Delta \\ (\Gamma, \Delta, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, (d_i(\tau_k)^k (x_{ij})^j \simeq \mathsf{reuse}(a)) : \zeta\ (\tau_k)^k \vdash^p b_i : \tau \Rightarrow \Delta_i)^i\end{array}}{\begin{array}{c}\Gamma \vdash^p \mathsf{match}\ a\ \mathsf{with}\ (d_i(\tau_k)^k (x_{ij})^j \to b_i)^i : \tau \\ \Rightarrow \Delta, (\mathsf{match}\ \mathsf{reuse}(a)\ \mathsf{with}\ d_i(\tau_k)^k (x_{ij})^j \to \Delta_i)^i\end{array}}$$

$$\textsc{Coerce}\quad \frac{\Gamma \vdash^p a : \tau' \Rightarrow \Delta \qquad \Gamma \vdash \tau' : \kappa' \simeq \tau : \kappa}{\Gamma \vdash^p a : \tau \Rightarrow \Delta}$$

$$\textsc{Reuse-Present}\quad \frac{(u){*}q : \tau \in \Gamma \qquad \Gamma \vdash u : \mathsf{bool} \simeq \mathsf{True} : \mathsf{bool}}{\Gamma \vdash^p {*}q : \tau \Rightarrow \varnothing}$$

$$\textsc{Reuse-Absurd}\quad \frac{q \perp \Gamma \qquad \Gamma \vdash \mathsf{False} : \mathsf{bool} \simeq \mathsf{True} : \mathsf{bool} \qquad \Gamma \vdash \tau : \mathsf{Typ}}{\Gamma \vdash^p {*}q : \tau \Rightarrow \varnothing}$$

Figure 6.5: Typing rules for eML

rules Match and Let. The transformations are similar to these done to pass the values associated to labels through let and match contexts. The notation $\mathsf{match}\ u_0\ \mathsf{with}\ d_i(\tau_k)^k(x_{ij})^j \to \Delta$ is to be interpreted as transforming each element $(u)*p : \tau$ of Δ into $(\mathsf{match}\ u_0\ \mathsf{with}\ d_i(\tau_k)^k(x_{ij})^j \to u \mid (d_\ell(\tau_k)^k(x_{\ell j})^j \to \mathsf{False})^{\ell \neq i})*p : \mathsf{match}\ u_0\ \mathsf{with}\ d_i(\tau_k)^k(x_{ij})^j \to \tau \mid (d_\ell(\tau_k)^k(x_{\ell j})^j \to \mathsf{void})^{\ell \neq i}$ (the type void could be replaced by any other type).

Function definitions also introduce variables in the scope of their bodies, but the results from evaluating the body do not escape the abstraction (they are blocked by the reduction rule, and removed by the typing rule), so there is no need to retain information about these variables. Abstractions on types can also avoid doing this for another reason: the body of the abstraction is non-expansive (by syntactic guarantee) and thus its evaluation cannot involve reducing an application.

In the judgment $\Gamma \vdash^p a : \tau \Rightarrow \Delta$, p is a prefix that constrains the labels that can be returned in Δ. Most rules simply leave p unchanged. The application rule App checks for an application $a^q\ b$ that the label q is a suffix of p, and that it is independent from the labels in Γ and the labels in Δ_1 and Δ_2 generated by a and b. The returned set of results is composed of Δ_1, Δ_2, and the newly introduced result $(\mathsf{True})*q : \tau_2$: the condition is True because in this context the application must be reduced. The fixed-point rule Fix types the body a of an abstraction $\mathsf{fix}^\pi\ x\ (y : \tau) : \sigma\ .\ a$ under the path variable π introduced by the application. The computed results Δ are discarded: they are not available outside of the body of the abstraction.

When re-using a result q (Reuse-Present and Reuse-Absurd) we need to check that the evaluation of this application did take place. As in the well-labeling judgment, there are two possibilities: either the label is present in the context as $(u)*q : \tau$, in which case we look up its condition u and check that it is equal to True in the current context (by using the equality $\Gamma \vdash u : \mathsf{bool} \simeq \mathsf{True} : \mathsf{bool}$): then the type of the expression is τ. The other possibility (Reuse-Absurd) is that q is not present in the context. In that case, we require, as in the L-Reuse, that q is independent of Γ, and that the branch is dead, as execution would be stuck if we had to reduce the term. This is expressed by asking that True be equal to False in the current context. Then, we accept to type this term with any well-formed type.

The features of eML rely on the ability of the type system to track branches. This is done by adding equalities in the typing context: when under a branch, for example in the term a in the expression $\mathsf{match}\ u\ \mathsf{with}\ \mathsf{Z} \to a \mid \mathsf{S}\ x \to b$, the context contains an equality $(u \simeq \mathsf{Z}) : \mathsf{nat}$. This equality witnesses that u must be Z when evaluating a, which allows for example reusing the results of applications computed below such a branch in a previous location in the term. When matching on a term c, not necessarily non-expansive, we introduce an equality between $\mathsf{reuse}(c)$ and (*e.g.*) Z instead. This is done in Match. For convenience, we also introduce an equality for let bindings (Let): in $\mathsf{let}\ x = a\ \mathsf{in}\ b$, if a has type τ, the equality $(x \simeq \mathsf{reuse}(a)) : \tau$ is available when typing b. The equalities in the context are always between non-expansive terms. These equalities are not used directly in the typing judgment, but in type and term equality judgments, written $\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$ and $\Gamma \vdash a_1 : \tau_1 \simeq a_2 : \tau_2$. The definition of these judgments is given in the next section (§6.2.4).

Equality judgments are used in two places in the typing derivations. First, they are used to *coerce* terms from one type to another (Coerce): if two types

$$\textsc{EnvEmpty}\quad \vdash \varnothing$$

$$\textsc{EnvPathVar}\quad \frac{\vdash \Gamma \qquad \pi \# \Gamma}{\vdash \Gamma, \pi}$$

$$\textsc{EnvVar}\quad \frac{\vdash \Gamma \qquad \Gamma \vdash \tau : \mathsf{Sch} \qquad x \# \Gamma}{\vdash \Gamma, x : \tau}$$

$$\textsc{EnvTVar}\quad \frac{\vdash \Gamma \qquad \alpha \# \Gamma}{\vdash \Gamma, \alpha : \kappa}$$

$$\textsc{EnvEq}\quad \frac{\vdash \Gamma \qquad \Gamma \vdash u_1 : \tau \qquad \Gamma \vdash u_2 : \tau}{\vdash \Gamma, (u_1 \simeq u_2) : \tau}$$

$$\textsc{EnvPtr}\quad \frac{\vdash \Gamma \qquad p \perp \Gamma \qquad (p = \pi \cdot q \implies \pi \in \Gamma) \qquad \Gamma \vdash u : \mathsf{bool} \qquad \Gamma \vdash \tau : \mathsf{Sch}}{\vdash \Gamma, (u)*p : \tau}$$

$$\textsc{K-Var}\quad \frac{\vdash \Gamma \qquad \alpha : \mathsf{Typ} \in \Gamma}{\Gamma \vdash \alpha : \mathsf{Typ}}$$

$$\textsc{K-Datatype}\quad \frac{\vdash \Gamma \qquad \vdash \zeta : (\mathsf{Typ})^i \Rightarrow \mathsf{Typ} \qquad (\Gamma \vdash \tau_i : \mathsf{Typ})^i}{\Gamma \vdash \zeta\,(\tau_i)^i : \mathsf{Typ}}$$

$$\textsc{K-Arr}\quad \frac{\Gamma \vdash \tau_1 : \mathsf{Typ} \qquad \Gamma \vdash \tau_2 : \mathsf{Typ}}{\Gamma \vdash \tau_1 \to \tau_2 : \mathsf{Typ}}$$

$$\textsc{K-All}\quad \frac{\Gamma, \alpha : \kappa \vdash \tau : \mathsf{Sch}}{\Gamma \vdash \forall(\alpha : \mathsf{Typ})\,\tau : \mathsf{Sch}}$$

$$\textsc{K-SubTyp}\quad \frac{\Gamma \vdash \tau : \mathsf{Typ}}{\Gamma \vdash \tau : \mathsf{Sch}}$$

$$\textsc{K-Match}\quad \frac{\begin{array}{c}\vdash (d_i)^i : \zeta \ \mathsf{complete} \qquad (d_i : \forall(\alpha_k : \mathsf{Typ})^k\,(\tau_{ij})^j \to \zeta\,(\alpha_k)^k)^i \qquad \Gamma \vdash u : \zeta\,(\tau_k)^k \\ (\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, (d_i(\tau_k)^k(x_{ij})^j \simeq u) : \zeta\,(\tau_k)^k \vdash \sigma_i : \kappa)^i\end{array}}{\Gamma \vdash \mathsf{match}\ u\ \mathsf{with}\ (d_i(\tau_k)^k(x_{ij})^j \to \sigma_i)^i : \kappa}$$

Figure 6.6: Well-formedness rules of eML

are equal (in a given environment), terms of one type can be used as terms of the other type. In our system, this rule is mostly to introduce and eliminate pattern matching in types. We chose to make coercion invisible in the syntax of terms: this makes the syntax much lighter and allow us to express simply transformations that would have to perform complicated transformations on equalities: indeed, we will introduce a full reduction for non-expansive parts $\longrightarrow_\iota$ that would need to perform complex transformations on explicit equalities. Second, they are used in Reuse-Present to check that the result that we are attempting to reuse has been computed.

Before proceeding with describing equality, we describe in Figure 6.6 the well-formedness rules for type environments ($\vdash \Gamma$) and for types ($\Gamma \vdash \tau : \kappa$).

The well-formedness judgment for Γ, as well as the typing rules, reproduce the assumption in labeled ML that the set of available names is orthogonal: EnvPathVar checks that path variable names are unique, and EnvPtr ensures that the introduced path is neither a prefix nor a suffix of an existing path, and that if it starts with a variable, this variable is present in Γ.

6.2.4 The non-expansive equality judgment

We define the equality judgment on type and non-expansive terms. These terms do not emit labels: the equality judgment can then be built around a simpler

version of the typing judgment. We want equality to be symmetric, reflexive (on well-formed types and on well-typed non-expansive terms) and transitive. It also needs to be a congruence, and compatible with reduction of type-level and term-level pattern matching, let binding, and type application.

We start with the type equality judgment $\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$, stating that under the environment Γ, the types τ_1 and τ_2 of kinds κ_1 and κ_2 are equal. Its rules are given on Figure 6.7. We include a rule for transitivity (TEq-Trans) and for symmetry (TEq-Sym). Reflexivity can be proved using the congruence rules. For each well-formedness rules for types, we include a corresponding *congruence* rule that states that two types of a given syntactic shape are equal if their subtypes are equal: TEq-Var for K-Var, TEq-Datatype for K-Datatype, TEq-Arr for K-Arr, TEq-All for K-All. Since the kind of the types appear in the judgment, we also include a rule TEq-SubTyp that allows converting the left-hand side of an equality from the kind of monotypes to the kind of schemes. We can do the same transformation on the right-hand side using symmetry. The rule TEq-Match corresponding to K-Match is particular: since it compares two pattern matchings where two different (but equal) terms are matched on, we need to chose whether to include an equality with the term on the left- or right-hand side of the equality. We choose the left-hand side. The reduction of type-level pattern matching is handled through a rule TEq-ReduceMatch. Note that the reduction can be done even when the term we are matching on is not fully reduced to a value. Finally, we allow case-splitting on non-expansive terms of a datatype (TEq-Split). For each constructor, we can try to prove the equality assuming that the term starts with this constructor. If we can prove the equality in all of these cases, then it is true. This implies that, if we have a term of a datatype without constructors such as void, we can prove the equality of any two types.

The non-expansive term equality judgment $\Gamma \vdash u_1 : \tau_1 \simeq u_2 : \tau_2$ is defined by the structural and reduction rules in Figure 6.8 and the congruence rules in Figure 6.9. While it is apparently an heterogeneous equality, we prove (Lemma 6.15) that it actually only proves equality between terms of equal types. Its structure is similar to the type equality judgment: it has a transitivity (Eq-Trans), symmetry (Eq-Sym) and case-splitting rule (Eq-Split), congruence rules for each syntactic construct of $e\mathsf{ML}$ terms (Figure 6.9), and reduction rules for let binding (Eq-ReduceLet), type application (Eq-ReduceTApp) and pattern matching (Eq-ReduceMatch). The rule Eq-Coerce allows substituting the types on both side of the equality by equal types. Finally, the rule Eq-Subst allows using equalities in the context.

Note that when there is a contradiction in the context (such as $\mathsf{True} = \mathsf{False}$), we can deduce the equality of any two types or non-expansive terms: consider types τ_1 and τ_2. Then, by TEq-Match, the types $\mathsf{match\ True\ with\ True} \to \tau_1 \mid \mathsf{False} \to \tau_2$ and $\mathsf{match\ False\ with\ True} \to \tau_1 \mid \mathsf{False} \to \tau_2$ are equal. Reducing these two equal types, we get τ_1 for the first and τ_2 for the second. By transitivity, we have proved $(\mathsf{True} \simeq \mathsf{False}) : \mathsf{bool} \vdash \tau_1 : \mathsf{Sch} \simeq \tau_2 : \mathsf{Sch}$. Similarly, $(\mathsf{True} \simeq \mathsf{False}) : \mathsf{bool} \vdash u_1 : \tau_1 \simeq u_2 : \tau_2$ for any u_1, u_2, τ_1, τ_2.

6.2.5 Full term equality

There is a wrinkle in this definition: we only define equality on non-expansive terms, but arbitrary terms can appear under lambda abstractions. In order

TEq-Trans
$$\frac{\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2 \qquad \Gamma \vdash \tau_2 : \kappa_2 \simeq \tau_3 : \kappa_3}{\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_3 : \kappa_3}$$

TEq-Sym
$$\frac{\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2}{\Gamma \vdash \tau_2 : \kappa_2 \simeq \tau_1 : \kappa_1}$$

TEq-Split
$$\frac{\begin{array}{c} \vdash (d_i)^i : \zeta \ \mathsf{complete} \qquad (\vdash d_i : \forall(\alpha_k : \mathsf{Typ})^k\,(\tau_{ij})^j \to \zeta\,(\alpha_k)^k)^i \\ \Gamma \vdash u : \zeta\,(\tau_k)^k \qquad \Gamma \vdash \sigma_1 : \kappa_1 \qquad \Gamma \vdash \sigma_2 : \kappa_2 \\ (\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, (d_i(\tau_k)^k(x_{ij})^j \simeq u) : \zeta\,(\tau_k)^k \vdash \sigma_1 : \kappa_1 \simeq \sigma_2 : \kappa_2)^i \end{array}}{\Gamma \vdash \sigma_1 : \kappa_1 \simeq \sigma_2 : \kappa_2}$$

TEq-SubTyp
$$\frac{\Gamma \vdash \tau : \mathsf{Typ} \simeq \tau' : \kappa}{\Gamma \vdash \tau : \mathsf{Sch} \simeq \tau' : \kappa}$$

TEq-Var
$$\frac{\vdash \Gamma \qquad \alpha : \mathsf{Typ} \in \Gamma}{\Gamma \vdash \alpha : \mathsf{Typ} \simeq \alpha : \mathsf{Typ}}$$

TEq-Datatype
$$\frac{\vdash \Gamma \qquad \vdash \zeta : (\mathsf{Typ})^i \Rightarrow \mathsf{Typ} \qquad (\Gamma \vdash \tau_i : \mathsf{Typ} \simeq \tau_i' : \mathsf{Typ})^i}{\Gamma \vdash \zeta\,(\tau_i)^i : \mathsf{Typ} \simeq \zeta\,(\tau_i')^i : \mathsf{Typ}}$$

TEq-Arr
$$\frac{\Gamma \vdash \tau_1 : \mathsf{Typ} \simeq \tau_1' : \mathsf{Typ} \qquad \Gamma \vdash \tau_2 : \mathsf{Typ} \simeq \tau_2' : \mathsf{Typ}}{\Gamma \vdash \tau_1 \to \tau_2 : \mathsf{Typ} \simeq \tau_1' \to \tau_2' : \mathsf{Typ}}$$

TEq-All
$$\frac{\Gamma, \alpha : \mathsf{Typ} \vdash \tau : \mathsf{Sch} \simeq \tau' : \mathsf{Sch}}{\Gamma \vdash \forall(\alpha : \mathsf{Typ})\,\tau : \mathsf{Sch} \simeq \forall(\alpha : \mathsf{Typ})\,\tau' : \mathsf{Sch}}$$

TEq-Match
$$\frac{\begin{array}{c} \vdash (d_i)^i : \zeta \ \mathsf{complete} \\ (d_i : \forall(\alpha_k : \mathsf{Typ})^k\,(\tau_{ij})^j \to \zeta\,(\alpha_k)^k)^i \qquad \Gamma \vdash u_1 : \zeta\,(\tau_{1k})^k \simeq u_2 : \zeta\,(\tau_{2k})^k \\ (\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_{1k}]^k)^j, (d_i(\tau_{1k})^k(x_{ij})^j \simeq u_1) : \zeta\,(\tau_{1k})^k \vdash \sigma_{1i} : \kappa_1 \simeq \sigma_{2i} : \kappa_2)^i \end{array}}{\begin{array}{c} \Gamma \vdash \mathsf{match}\ u_1\ \mathsf{with}\ (d_i(\tau_{1k})^k(x_{ij})^j \to \sigma_{1i})^i : \kappa_1 \\ \simeq \mathsf{match}\ u_2\ \mathsf{with}\ (d_i(\tau_{2k})^k(x_{ij}')^j \to \sigma_{2i})^i : \kappa_2 \end{array}}$$

TEq-ReduceMatch
$$\frac{\begin{array}{c} \vdash (d_i)^i : \zeta \ \mathsf{complete} \\ (d_i : \forall(\alpha_k : \mathsf{Typ})^k\,(\tau_{ij})^j \to \zeta\,(\alpha_k)^k)^i \qquad (\Gamma \vdash u_j : \tau_{ij}[\alpha_k \leftarrow \tau_k']^k)^j \\ \Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, (d_i(\tau_k)^k(x_{ij})^j \simeq d_i(\tau_k')^k(u_j)^j) : \zeta\,(\tau_k)^k \vdash \sigma_i : \kappa \end{array}}{\Gamma \vdash \mathsf{match}\ d_i(\tau_k')^k(u_j)^j\ \mathsf{with}\ (d_i(\tau_k)^k(x_{ij})^j \to \sigma_i)^i : \kappa \simeq \sigma_i[x_{ij} \leftarrow u_j]^j : \kappa}$$

Figure 6.7: Type equality

$$\textsc{Eq-Trans}\quad \frac{\Gamma \vdash u_1 : \tau_1 \simeq u_2 : \tau_2 \qquad \Gamma \vdash u_2 : \tau_2 \simeq u_3 : \tau_3}{\Gamma \vdash u_1 : \tau_1 \simeq u_3 : \tau_3}$$

$$\textsc{Eq-Sym}\quad \frac{\Gamma \vdash u_1 : \tau_1 \simeq u_2 : \tau_2}{\Gamma \vdash u_2 : \tau_2 \simeq u_1 : \tau_1}$$

$$\textsc{Eq-Split}\quad \frac{\begin{array}{c} \vdash (d_i)^i : \zeta \ \mathsf{complete} \qquad (d_i : \forall(\alpha_k : \mathsf{Typ})^k\ (\tau_{ij})^j \to \zeta\ (\alpha_k)^k)^i \\ \Gamma \vdash u : \zeta\ (\tau_k)^k \qquad \Gamma \vdash u_1 : \tau_1 \qquad \Gamma \vdash u_2 : \tau_2 \\ (\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, (d_i(\tau_k)^k(x_{ij})^j \simeq u) : \zeta\ (\tau_k)^k \vdash u_1 : \tau_1 \simeq u_2 : \tau_2)^i \end{array}}{\Gamma \vdash u_1 : \tau_1 \simeq u_2 : \tau_2}$$

$$\textsc{Eq-Subst}\quad \frac{\vdash \Gamma \qquad ((u_1 \simeq u_2) : \tau) \in \Gamma}{\Gamma \vdash u_1 : \tau \simeq u_2 : \tau}$$

$$\textsc{Eq-ReduceLet}\quad \frac{\Gamma \vdash u : \tau_1 \qquad \Gamma, x : \tau_1, (x \simeq u) : \tau_1 \vdash u' : \tau_2}{\Gamma \vdash \mathsf{let}\ x = u\ \mathsf{in}\ u' : \tau_2 \simeq u'[x \leftarrow u] : \tau_2}$$

$$\textsc{Eq-ReduceTApp}\quad \frac{\Gamma, \alpha : \mathsf{Typ} \vdash u : \sigma \qquad \Gamma \vdash \tau : \mathsf{Typ}}{\Gamma \vdash (\Lambda(\alpha : \mathsf{Typ}).\ u)\ \tau_0 : \sigma[\alpha \leftarrow \tau_0] \simeq u[\alpha \leftarrow \tau_0] : \sigma[\alpha \leftarrow \tau_0]}$$

$$\textsc{Eq-ReduceMatch}\quad \frac{\begin{array}{c} \vdash (d_i)^i : \zeta \ \mathsf{complete} \\ (d_i : \forall(\alpha_k : \mathsf{Typ})^k\ (\tau_{ij})^j \to \zeta\ (\alpha_k)^k)^i \qquad (\Gamma \vdash u_j : \tau_{ij}[\alpha_k \leftarrow \tau'_k]^k)^j \\ \Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, (d_i(\tau_k)^k(x_{ij})^j \simeq d_i(\tau'_k)^k(u_j)^j) : \zeta\ (\tau_k)^k \vdash u'_i : \sigma \end{array}}{\Gamma \vdash \mathsf{match}\ d_i(\tau'_k)^k(u_j)^j\ \mathsf{with}\ (d_i(\tau_k)^k(x_{ij})^j \to u'_i)^i : \sigma \simeq u'_i[x_{ij} \leftarrow u_j]^j : \sigma}$$

Figure 6.8: Non-expansive term equality

Eq-Coerce

$$\frac{\Gamma \vdash u_1 : \tau_1' \simeq u_2 : \tau_2' \qquad \Gamma \vdash \tau_1' : \kappa_1' \simeq \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2' : \kappa_2' \simeq \tau_2 : \kappa_2}{\Gamma \vdash u_1 : \tau_1 \simeq u_2 : \tau_2}$$

Eq-Var

$$\frac{\vdash \Gamma \qquad x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma \simeq x : \sigma}$$

Eq-TAbs

$$\frac{\Gamma, \alpha : \mathsf{Typ} \vdash u_1 : \sigma_1 \simeq u_2 : \sigma_2}{\Gamma \vdash \Lambda(\alpha : \mathsf{Typ}).\, u_1 : \forall(\alpha : \mathsf{Typ})\, \sigma_1 \simeq \Lambda(\alpha : \mathsf{Typ}).\, u_2 : \forall(\alpha : \mathsf{Typ})\, \sigma_2}$$

Eq-TApp

$$\frac{\Gamma \vdash \tau_1 : \mathsf{Typ} \simeq \tau_2 : \mathsf{Typ} \qquad \Gamma \vdash u_1 : \forall(\alpha : \mathsf{Typ})\, \sigma_1 \simeq u_2 : \forall(\alpha : \mathsf{Typ})\, \sigma_2}{\Gamma \vdash u_1\ \tau_1 : \sigma_1[\alpha \leftarrow \tau_1] \simeq u_2\ \tau_2 : \sigma_2[\alpha \leftarrow \tau_2]}$$

Eq-Fix

$$\frac{\begin{array}{c}\Gamma \vdash \tau_1 : \mathsf{Typ} \simeq \tau_2 : \mathsf{Typ} \\ \mathsf{TermsEqual}(\Gamma, x : \tau_1 \to \tau_1', y : \tau_1 \vdash^{\pi} a_1 : \tau_1' \simeq a_2 : \tau_2')\end{array}}{\Gamma \vdash \mathsf{fix}^{\pi}\, x\, (y : \tau_1) : \tau_1'\ .\, a_1 : \tau_1 \to \tau_1' \simeq \mathsf{fix}^{\pi}\, x\, (y : \tau_2) : \tau_2'\ .\, a_2 : \tau_2 \to \tau_2'}$$

Eq-Let

$$\frac{\Gamma \vdash u_1 : \sigma_1 \simeq u_2 : \sigma_2 \qquad \Gamma, x : \sigma_1 \vdash u_1' : \tau_1 \simeq u_2' : \tau_2}{\Gamma \vdash \mathsf{let}\ x = u_1\ \mathsf{in}\ u_1' : \tau_1 \simeq \mathsf{let}\ x = u_2\ \mathsf{in}\ u_2' : \tau_2}$$

Eq-Con

$$\frac{\begin{array}{c}\vdash \Gamma \qquad \vdash d : \forall(\alpha_j : \mathsf{Typ})^j\, (\tau_i)^i \to \zeta\, (\alpha_j)^j \qquad (\Gamma \vdash \tau_{1j} : \mathsf{Typ} \simeq \tau_{2j} : \mathsf{Typ})^j \\ (\Gamma \vdash u_{1i} : \tau_i[\alpha_j \leftarrow \tau_{1j}]^j \simeq u_{2i} : \tau_i[\alpha_j \leftarrow \tau_{2j}]^j)^i\end{array}}{\Gamma \vdash d(\tau_{1j})^j (u_{1i})^i : \zeta\, (\tau_{1j})^j \simeq d(\tau_{2j})^j (u_{2i})^i : \zeta\, (\tau_{2j})^j}$$

Eq-Match

$$\frac{\begin{array}{c}\vdash (d_i)^i : \zeta\ \mathsf{complete} \\ (d_i : \forall(\alpha_k : \mathsf{Typ})^k\, (\tau_{ij})^j \to \zeta\, (\alpha_k)^k)^i \qquad \Gamma \vdash u_1 : \zeta\, (\tau_{1k})^k \simeq u_2 : \zeta\, (\tau_{2k})^k \\ (\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, (d_i(\tau_{1k})^k (x_{ij})^j \simeq u_1) : \zeta\, (\tau_{1k})^k \vdash u_{1i}' : \sigma_1 \simeq u_{2i}' : \sigma_2)^i\end{array}}{\begin{array}{c}\Gamma \vdash \mathsf{match}\ u_1\ \mathsf{with}\ (d_i(\tau_{1k})^k (x_{ij})^j \to u_{1i}')^i : \sigma_1 \\ \simeq \mathsf{match}\ u_2\ \mathsf{with}\ (d_i(\tau_{2k})^k (x_{ij}')^j \to u_{2i}')^i : \sigma_2\end{array}}$$

Eq-Reuse-Present

$$\frac{(u)*p : \tau \in \Gamma \qquad \Gamma \vdash u : \mathsf{bool} \simeq \mathsf{True} : \mathsf{bool}}{\Gamma \vdash *p : \tau \simeq *p : \tau}$$

Eq-Reuse-Absurd

$$\frac{p \perp \Gamma \qquad \Gamma \vdash \mathsf{False} : \mathsf{bool} \simeq \mathsf{True} : \mathsf{bool}}{\Gamma \vdash *p : \tau \simeq *p : \tau}$$

Figure 6.9: Non-expansive term equality (congruence rules)

not to get stuck, we delegate checking of equality for terms to a judgment $\mathsf{TermsEqual}(\Gamma \vdash^p a_1 : \tau_1 \simeq a_2 : \tau_2)$: this judgment assert that the terms a_1 and a_2 are equal at types τ_1 and τ_2 under an environment Γ and a label prefix p. This judgment does not compare the results computed when evaluating a_1 and a_2. This does not prevent us from defining the rule Eq-Fix since Δ is removed by the Fix typing rule, and many reasonable program transformations change labels.

Decomposition of terms

To define the equality on term, we want to separate the expansive part of a term from its non-expansive parts, and reason by equality on these two components separately. We define an *extraction* of expansive computations from an expansive term a. This extraction returns a mapping from computation labels to *expansive atoms*, *i.e.* applications of the form $u\ u'$, where both sides of the application are non-expansive terms. For example, for the term $a \triangleq \mathsf{Some}\ ((u_1{}^1\ u_2)^2\ (\lambda^\pi\ (x : \tau).\ u_3{}^{\pi\cdot 1}\ u_4))$, its decomposition is:

$$\begin{array}{lcl} *1 & \mapsto & u_1\ u_2 \\ *2 & \mapsto & *1\ (\lambda^\pi\ (x : \tau).\ u_3{}^{\pi\cdot 1}\ u_4) \\ \mathsf{reuse}(a) & \mapsto & \mathsf{Some}\ (*2) \end{array}$$

We need to add typing information to the decomposition: suppose $\Gamma \vdash^p a : \tau \Rightarrow \Delta$. Then, we want every expansive atom to be typed in Γ, Δ', with Δ' a (strict) subset of Δ, under a condition $(u \simeq \mathsf{True}) : \mathsf{bool}$. Consider for example Γ equal to $f : \mathsf{bool} \to \mathsf{bool}, x : \mathsf{bool}$, $\Delta = (\mathsf{True})*1 : \mathsf{bool}, (x)*2 : \mathsf{bool}$ and the following term:

$$\Gamma \vdash^\epsilon \left(\begin{array}{l} \mathsf{let}\ y = \mathsf{True}\ \mathsf{in}\ f^1\ y, \\ \mathsf{match}\ x\ \mathsf{with}\ \mathsf{True} \to f^2\ *1 \mid \mathsf{False} \to x \end{array} \right) : \mathsf{bool} \times \mathsf{bool} \Rightarrow \Delta$$

The expression labeled 2 depends on the expression labeled 1, thus its hypotheses will be $\Delta' = (\mathsf{True})*1 : \mathsf{bool}$. Moreover, the expansive atom labeled 1 is under a let binding. We use context translation (as in reduction) to express it in Γ. Finally, the expression labeled 2 is only computed if $x = \mathsf{True}$, so it is protected by a condition x. The full decomposition is then the following. Note that the expansive terms are ordered in the same order they would be evaluated in the original term.

$$\begin{array}{lll} \Gamma & \vdash (\mathsf{True})*1 & := f\ \mathsf{True} : \mathsf{bool} \\ \Gamma, \Delta' & \vdash (x)*2 & := f\ *1 : \mathsf{bool} \\ \Gamma, \Delta & \vdash \mathsf{reuse}(a) & := \left(\begin{array}{l} \mathsf{let}\ y = \mathsf{True}\ \mathsf{in}\ *1, \\ \mathsf{match}\ x\ \mathsf{with}\ \mathsf{True} \to *2 \mid \mathsf{False} \to x \end{array} \right) : \mathsf{bool} \times \mathsf{bool} \end{array}$$

The general decomposition of a term, noted $\mathsf{atoms}(a)$, is defined in Figure 6.10: it returns the list of atoms and the set of labels that they depend on (the final non-expansive expression for the term is given by $\mathsf{reuse}(a)$). We note $\mathsf{labels}(a)$ the set of labels defined in a term, and $\mathsf{addLabels}(S, (*p_i \leftarrow (T_i, u_i\ u'_i))^i) = (*p_i \leftarrow (S \cup T_i, u_i\ u'_i))^i$ the addition of the labels in S to the dependencies of all atoms of the second argument. We use this for example in the definition of $\mathsf{atoms}(a^p\ b)$: the decomposition is obtained by concatenating

$$
\begin{aligned}
&\mathsf{atoms}(u) = \varnothing \\
&\mathsf{atoms}(a\ \tau) = \mathsf{atoms}(a) \\
&\mathsf{atoms}(\mathsf{let}\ x = a\ \mathsf{in}\ b) = \\
&\quad \mathsf{atoms}(a), (\mathsf{addLabels}(\mathsf{labels}(a), \mathsf{atoms}(b)))[x \leftarrow \mathsf{reuse}(a)] \\
&\mathsf{atoms}(\mathsf{match}\ a\ \mathsf{with}\ (c_i\, \overline{\tau}\, (x_{ik})^k \rightarrow b_i)^i) = \\
&\quad \mathsf{atoms}(a), (\mathsf{addLabels}(\mathsf{labels}(a), \mathsf{match}\ a\ \mathsf{with}\ c_i(\tau_j)^j (x_{ik})^k \rightarrow \mathsf{atoms}(b_i))^i \\
&\mathsf{atoms}(c\, \overline{\tau}\, (a_j)^j) = (\mathsf{addLabels}((\mathsf{labels}(b_k))^{k>j}, \mathsf{atoms}(a_j)))^j \\
&\mathsf{atoms}(a^p\ b) = \\
&\quad \mathsf{atoms}(a), \\
&\quad \mathsf{addLabels}(\mathsf{labels}(a), \mathsf{atoms}(b)), \\
&\quad *p \leftarrow (\mathsf{labels}(a) \cup \mathsf{labels}(b), \mathsf{reuse}(a)\ \mathsf{reuse}(b))
\end{aligned}
$$

Figure 6.10: Decomposition of terms

the decomposition $\mathsf{atoms}(a)$ of a, then the decomposition $\mathsf{atoms}(b)$ of b, with all labels of a added as dependencies, then the label $*p$ corresponding to the application. The decomposition of non-expansive terms is empty. For general terms, it obeys the following typing property:

Lemma 6.1. *Suppose* $\Gamma \vdash^p a : \tau \Rightarrow \Delta$, *and* $\mathsf{atoms}(a) = (*p_i \leftarrow (S_i, u_i\ u'_i))^i$. *Then, the labels of* Δ *are the* $(p_i)^i$ *and for all* i, *there exists* τ_i, σ_i *such that* $\Gamma, \Delta \cap S_i \vdash u_i : \tau_i \rightarrow \sigma_i$ *and* $\Gamma, \Delta \cap S_i \vdash u'_i : \tau_i$.

Proof. By induction on the typing derivation. □

Reasoning on decompositions

We define an equality on decompositions that preserves the fact that equivalent expansive atoms are evaluated in the same order in both terms. We still want to allow the evaluation of a subexpression to be split, and subexpressions to be renamed: the terms $f^1\ x$ and $\mathsf{match}\ y\ \mathsf{with}\ \mathsf{True} \rightarrow f^2\ x \mid \mathsf{False} \rightarrow f^3$ should be considered equivalent, as in the $y = \mathsf{True}$ case, 1 on the left will match 2 on the right, and 3 on the right will not be evaluated, and in the case of $y = \mathsf{False}$, 1 will match 3 and 2 will not be evaluated. Thus, an equality should allow to provide a *mapping* between the labels on the left-hand side and on the right-hand side.

The evaluations are ordered and we have conditions available telling us if an expansive atom is evaluated or not: thus, we simply want the first evaluated atom on the left to match the first evaluated atom, and so on until the last evaluated atom.

We first define an *indexing* function that takes a list of non-expansive conditions and associated terms or types and returns the term or type corresponding to the k-th true condition.

Definition 6.1 (Indexing)**.** *Consider a type* end *with one zero-argument constructor* End. *We use this type and this constructor as a special marker when*

indexing goes past the end. We define $\textsc{Index}_k(u_i \to u'_i)^i$ *as follows:*

$$
\begin{array}{lcl}
\textsc{Index}_k(\emptyset) & = & \mathsf{End} \\
\textsc{Index}_0(u_i \to u'_i)^{i\in\{1,\ldots,n\}} & = & \mathsf{match}\ u_1\ \mathsf{with} \\
 & & \quad \mid \mathsf{True} \to u'_1 \\
 & & \quad \mid \mathsf{False} \to \textsc{Index}_0(u_i \to u'_i)^{i\in\{2,\ldots,n\}} \\
\textsc{Index}_{n+1}(u_i \to u'_i)^{i\in\{1,\ldots,n\}} & = & \mathsf{match}\ u_1\ \mathsf{with} \\
 & & \quad \mid \mathsf{True} \to \textsc{Index}_n(u_i \to u'_i)^{i\in\{2,\ldots,n\}} \\
 & & \quad \mid \mathsf{False} \to \textsc{Index}_{n+1}(u_i \to u'_i)^{i\in\{2,\ldots,n\}}
\end{array}
$$

Similarly, we can define for types:

$$
\begin{array}{lcl}
\textsc{Index}_k(\emptyset) & = & \mathsf{end} \\
\textsc{Index}_0(u_i \to \tau_i)^{i\in\{1,\ldots,n\}} & = & \mathsf{match}\ u_1\ \mathsf{with} \\
 & & \quad \mid \mathsf{True} \to \tau_1 \\
 & & \quad \mid \mathsf{False} \to \textsc{Index}_0(u_i \to \tau_i)^{i\in\{2,\ldots,n\}} \\
\textsc{Index}_{n+1}(u_i \to \tau_i)^{i\in\{1,\ldots,n\}} & = & \mathsf{match}\ u_1\ \mathsf{with} \\
 & & \quad \mid \mathsf{True} \to \textsc{Index}_n(u_i \to \tau_i)^{i\in\{2,\ldots,n\}} \\
 & & \quad \mid \mathsf{False} \to \textsc{Index}_{n+1}(u_i \to \tau_i)^{i\in\{2,\ldots,n\}}
\end{array}
$$

$\diamondsuit$

We have the following typing lemma for indexing:

Lemma 6.2 (Typing of indexing)**.** *Consider an environment* Γ *and suppose* $(\Gamma \vdash u_i : \mathsf{bool})^i$ *and* $(\Gamma, (u_i \simeq \mathsf{True}) : \mathsf{bool} \vdash u'_i : \sigma_i)^i$*. Then for all* n*,* $\Gamma \vdash \textsc{Index}_n(u_i \to u'_i)^i : \textsc{Index}_n(u_i \to \sigma_i)^i$*.*

Proof. By induction on the list: at each stage we add a pattern matching in each of the branches to match the indexing function. $\square$

We then use the following definition:

Definition 6.2 (Term equality)**.** *Consider an environment* Γ*, two terms* a*,* b*, and two types* τ_a*,* τ_b *such that there exists* p*,* Δ_a*,* Δ_b *such that* $\Gamma \vdash^p a : \tau_a \Rightarrow \Delta_a$ *and* $\Gamma \vdash^p b : \tau_b \Rightarrow \Delta_b$*. Let* $S_a = \mathsf{labels}(a)$*,* $S_b = \mathsf{labels}(b)$

Suppose $\mathsf{atoms}(a) = (*p_i \leftarrow (T_{a,i}, u_{a,i}\ u'_{a,i}))^{p_i\in\mathsf{labels}(a)}$ *and* $\mathsf{atoms}(b) = (*q_j \leftarrow (T_{b,j}, u_{b,j}\ u'_{b,j}))^{q_j\in\mathsf{labels}(b)}$*. Moreover, suppose that for all* i*,* $\Gamma, \Delta_a \cup T_{a,i} \vdash u_{a,i} : \tau_{a,i} \to \sigma_{a,i}$ *and* $\Gamma, \Delta_a \cup T_{a,i} \vdash u'_{a,i} : \tau_{a,i}$ *and similarly for all* j*,* $\Gamma, \Delta_b \cup T_{b,j} \vdash u_{b,j} : \tau_{b,j} \to \sigma_{b,j}$ *and* $\Gamma, \Delta_b \cup T_{b,j} \vdash u'_{b,j} : \tau_{b,j}$*. Finally, suppose that* $\Delta_a = ((u^c_{a,i}) * p_i : \sigma_{a,i})^i$ *and* $\Delta_b = ((u^c_{b,j}) * q_j : \sigma_{b,j})^j$*.*

We define a type $\mathsf{app}\ (\tau, \sigma)$ *with one constructor* $\mathsf{App} : (\tau \times (\tau \to \sigma)) \to \mathsf{app}\ (\tau, \sigma)$ *to represent function application atoms. Then, we say that* a *and* b *are equal, noted* $\mathsf{TermsEqual}(\Gamma \vdash^p a : \tau_a \simeq b : \tau_b)$ *if and only if:*

- *For all* k*:*

$$
\begin{array}{l}
\Gamma, \Delta_a, \Delta_b, ((\textsc{Index}_\ell(u^c_{a,i} \to p_i)^i \simeq \textsc{Index}_\ell(u^c_{b,j} \to q_j)^j) : \textsc{Index}_\ell(u^c_{a,i} \to \tau_i)^i)^{\ell<k} \\
\quad \vdash \textsc{Index}_k(u^c_{a,i} \to \mathsf{App}(u_{a,i}, u'_{a,i}))^i : \textsc{Index}_k(u^c_{a,i} \to \mathsf{app}\ (\tau_{a,i}, \sigma_{a,i}))^i \\
\quad \simeq \textsc{Index}_k(u^c_{b,j} \to \mathsf{App}(u_{b,j}, u'_{b,j}))^j : \textsc{Index}_k(u^c_{b,j} \to \mathsf{app}\ (\tau_{b,j}, \sigma_{b,j}))^j
\end{array}
$$

  i.e. *the* k*-th application on the left-hand side has equal function and arguments to the* k*-th application on the right-hand side.*

- *The final expressions are equal:*

$$\Gamma, \Delta_a, \Delta_b,$$
$$((\textsc{Index}_\ell(u^c_{a,i} \to p_i)^i \simeq \textsc{Index}_\ell(u^c_{b,j} \to q_j)^j) : \textsc{Index}_\ell(u^c_{a,i} \to \tau_i)^i)^{\ell < \mathsf{max}(n,m)}$$
$$\vdash \mathsf{reuse}(a) : \tau_a \simeq \mathsf{reuse}(b) : \tau_b$$

where n and m are the number of labels appearing in a and b respectively.

$\diamondsuit$

Note that if two non-expansive terms are equal, they are equal as terms: all indexing operation will rewrite to simply End (or end), and their reusable versions will be themselves, equal by hypothesis.

6.3 Metatheory of *e*ML

As for ML, the reduction $\longrightarrow_\beta$ is deterministic ($\longrightarrow_\beta$ does not change between ML and *e*ML).

6.3.1 Basic properties

We prove some basic properties of the typing judgments of *e*ML.

First, we need to prove a weakening lemma: we can freely add variables, type variables, path variables, computed results and equalities to the environment and preserve the typing and equality judgments.

Lemma 6.3 (Weakening)**.** *Consider Γ, Γ' and Δ, and let J be a judgment of the following forms:*

- $\vdash \Gamma, \Gamma'$
- $\Gamma, \Gamma' \vdash \tau : \kappa$
- $\Gamma, \Gamma' \vdash^p a : \tau \Rightarrow \Delta$
- $\Gamma, \Gamma' \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$
- $\Gamma, \Gamma' \vdash u_1 : \tau_1 \simeq u_2 : \tau_2$
- $\mathsf{TermsEqual}(\Gamma, \Gamma' \vdash^p a_1 : \tau_1 \simeq a_2 : \tau_2)$

Suppose J holds. Then:

- *Suppose $\alpha \# \Gamma, \Gamma'$. Then J also holds with Γ, Γ' replaced by $\Gamma, \alpha : \mathsf{Typ}, \Gamma'$.*
- *Suppose $x \# \Gamma, \Gamma'$ and $\Gamma \vdash \sigma : \mathsf{Sch}$. Then J also holds with Γ, Γ' replaced by $\Gamma, x : \sigma, \Gamma'$.*
- *Consider u'_1, u'_2 and σ such that $\Gamma \vdash u'_1 : \sigma$ and $\Gamma \vdash u'_2 : \sigma$. Then J also holds with Γ, Γ' replaced by $\Gamma, (u'_1 \simeq u'_2) : \sigma, \Gamma'$.*
- *Consider q, u' and σ such that $\Gamma \vdash u' : \mathsf{bool}$, $\Gamma \vdash \tau : \mathsf{Sch}$, and suppose q starts at the root or a variable in Γ and $q \perp \Gamma, \Gamma', \Delta$. Then J also holds with Γ, Γ' replaced by $\Gamma, (u')*q : \sigma, \Gamma'$.*

- *Consider a path variable* π *such that* $\pi \notin \Gamma, \Gamma'$*. Then* J *also holds with* Γ, Γ' *replaced by* Γ, π, Γ'*.*

Proof. Let us look at the proof in the case of type variables:

By mutual induction on the derivations. For rules that grow the environment, adding Γ'', note that for any Γ'', $\Gamma, \alpha : \mathsf{Typ}, \Gamma', \Gamma''$ is still a weakening of $\Gamma, \Gamma', \Gamma''$, so we can use the induction hypothesis on the subderivations. Then we copy the rule with the new environment. The only rules that access the environment instead of growing it are the variable rules Var, Eq-Var, K-Var, TEq-Var, and Eq-Subst but they only check membership in the environment so they stay true when the environment grows.

In the case of results, we also need to check that the newly introduced path q does not clash with any existing path: this is true because all results not introduced under lambdas are already in Γ, Γ' or Δ, and paths introduced under lambdas start with a variable that is not in Γ.

For $\mathsf{TermsEqual}(\Gamma \vdash^p a_1 : \tau_1 \simeq a_2 : \tau_2)$: all the judgments used in the proof are compatible with weakening. □

For proving substitution, we first need reflexivity of the equality judgment on well-typed terms and types:

Lemma 6.4 (Reflexivity)**.** *Consider an environment* Γ*, a kind* κ*, a type* τ*, and a non-expansive term* a*.*

- *If* $\Gamma \vdash \tau : \kappa$*, then* $\Gamma \vdash \tau : \kappa \simeq \tau : \kappa$*.*
- *If* $\Gamma \vdash a : \tau$*, then* $\Gamma \vdash a : \tau \simeq a : \tau$*.*
- *If* $\Gamma \vdash^p a : \tau \Rightarrow \Delta$*, then* $\mathsf{TermsEqual}(\Gamma \vdash^p a : \tau \simeq a : \tau)$*.*

Proof. For type and non-expansive equality: translate each typing rule to the equivalent equality rule and apply reflexivity inductively to the subtypes/subterms.

For full term equality: all the required conditions hold by reflexivity. □

Even though it does not have a transitivity rule, full term equality is transitive:

Lemma 6.5 (Term equality is transitive)**.** *Suppose* $\mathsf{TermsEqual}(\Gamma \vdash^p a_1 : \tau_1 \simeq a_2 : \tau_2)$ *and* $\mathsf{TermsEqual}(\Gamma \vdash^p a_2 : \tau_2 \simeq a_3 : \tau_3)$*. Then,* $\mathsf{TermsEqual}(\Gamma \vdash^p a_1 : \tau_1 \simeq a_3 : \tau_3)$*.*

Proof. In all three terms, the k-th evaluated atoms are equal. □

We can then show substitution for type and term variables and for equalities. For equalities, substitution means that we can remove from the environment an equality that can be derived from the rest of the environment.

Lemma 6.6 (Substitution, type variables)**.** *Consider* Γ, Γ' *and* α *such that* $\vdash \Gamma, \alpha : \mathsf{Typ}, \Gamma'$*. Let* σ *be a type such that* $\Gamma \vdash \sigma : \mathsf{Typ}$*. Then,*

- *we have* $\vdash \Gamma, \Gamma'[\alpha \leftarrow \sigma]$*;*
- *if* $\Gamma, \alpha : \mathsf{Typ}, \Gamma' \vdash \tau : \kappa$*, then* $\Gamma, \Gamma'[\alpha \leftarrow \sigma] \vdash \tau[\alpha \leftarrow \sigma] : \kappa[\alpha \leftarrow \sigma]$*;*

- *if* $\Gamma, \alpha : \mathsf{Typ}, \Gamma' \vdash^p a : \tau \Rightarrow \Delta$, *then* $\Gamma, \Gamma'[\alpha \leftarrow \sigma] \vdash^p a[\alpha \leftarrow \sigma] : \tau[\alpha \leftarrow \sigma] \Rightarrow \Delta[\alpha \leftarrow \sigma]$;

Moreover, consider σ_1, σ_2 *such that* $\Gamma \vdash \sigma_1 : \kappa_1$, $\Gamma \vdash \sigma_2 : \kappa_2$ *and* $\Gamma \vdash \sigma_1 : \mathsf{Typ} \simeq \sigma_2 : \mathsf{Typ}$. *Then,*

- *if* $\Gamma, \alpha : \mathsf{Typ}, \Gamma' \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$, *then* $\Gamma, \Gamma'[\alpha \leftarrow \sigma_1] \vdash \tau_1[\alpha \leftarrow \sigma_1] : \kappa_1[\alpha \leftarrow \sigma_1] \simeq \tau_2[\alpha \leftarrow \sigma_2] : \kappa_2[\alpha \leftarrow \sigma_2]$;
- *if* $\Gamma, \alpha : \mathsf{Typ}, \Gamma' \vdash u_1 : \tau_1 \simeq u_2 : \tau_2$, *then* $\Gamma, \Gamma'[\alpha \leftarrow \sigma_1] \vdash u_1[\alpha \leftarrow \sigma_1] : \tau_1[\alpha \leftarrow \sigma_1] \simeq u_2[\alpha \leftarrow \sigma_2] : \tau_2[\alpha \leftarrow \sigma_2]$;
- *if* $\mathsf{TermsEqual}(\Gamma, \alpha : \mathsf{Typ}, \Gamma' \vdash^p a_1 : \tau_1 \simeq a_2 : \tau_2)$, *then* $\mathsf{TermsEqual}(\Gamma, \Gamma'[\alpha \leftarrow \sigma_1] \vdash^p a_1[\alpha \leftarrow \sigma_1] : \tau_1[\alpha \leftarrow \sigma_1] \simeq a_2[\alpha \leftarrow \sigma_2] : \tau_2[\alpha \leftarrow \sigma_2])$.

Lemma 6.7 (Substitution, term variables)**.** *Consider* Γ, Γ' *and* x *such that* $\vdash \Gamma, x : \sigma, \Gamma'$. *Let* u *be a type such that* $\Gamma \vdash u : \sigma$. *Then,*

- *we have* $\vdash \Gamma, \Gamma'[x \leftarrow b]$;
- *if* $\Gamma, x : \sigma, \Gamma' \vdash \tau : \kappa$, *then* $\Gamma, \Gamma'[x \leftarrow b] \vdash \tau[x \leftarrow b] : \kappa[x \leftarrow b]$;
- *if* $\Gamma, x : \sigma, \Gamma' \vdash^p a : \tau \Rightarrow \Delta$, *then* $\Gamma, \Gamma'[x \leftarrow b] \vdash^p a[x \leftarrow b] : \tau[x \leftarrow b] \Rightarrow \Delta[x \leftarrow b]$;

Moreover, consider u_1, u_2 *such that* $\Gamma \vdash u_1 : \tau$, $\Gamma \vdash u_2 : \tau$ *and* $\Gamma \vdash u_1 : \tau \simeq u_2 : \tau$. *Then,*

- *if* $\Gamma, \alpha : \mathsf{Typ}, \Gamma' \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$, *then* $\Gamma, \Gamma'[\alpha \leftarrow u_1] \vdash \tau_1[\alpha \leftarrow u_1] : \kappa_1[\alpha \leftarrow u_1] \simeq \tau_2[\alpha \leftarrow u_2] : \kappa_2[\alpha \leftarrow u_2]$;
- *if* $\Gamma, \alpha : \mathsf{Typ}, \Gamma' \vdash u'_1 : \tau_1 \simeq u'_2 : \tau_2$, *then* $\Gamma, \Gamma'[\alpha \leftarrow u_1] \vdash u'_1[\alpha \leftarrow u_1] : \tau_1[\alpha \leftarrow u_1] \simeq u'_2[\alpha \leftarrow u_2] : \tau_2[\alpha \leftarrow u_2]$;
- *if* $\mathsf{TermsEqual}(\Gamma, \alpha : \mathsf{Typ}, \Gamma' \vdash^p a_1 : \tau_1 \simeq a_2 : \tau_2)$, *then* $\mathsf{TermsEqual}(\Gamma, \Gamma'[\alpha \leftarrow u_1] \vdash^p a_1[\alpha \leftarrow u_1] : \tau_1[\alpha \leftarrow u_1] \simeq a_2[\alpha \leftarrow u_2] : \tau_2[\alpha \leftarrow u_2])$.

Lemma 6.8 (Substitution, equalities)**.** *Consider* Γ, Γ' *and* u_1, u_2 *such that* $\vdash \Gamma, (u_1 : \sigma_1) \simeq (u_2 : \sigma_2), \Gamma'$. *Suppose* $\Gamma \vdash u_1 : \sigma_1 \simeq u_2 : \sigma_2$. *Then,*

- *we have* $\vdash \Gamma, \Gamma'$;
- *if* $\Gamma, (u_1 : \sigma_1) \simeq (u_2 : \sigma_2), \Gamma' \vdash \tau : \kappa$, *then* $\Gamma, \Gamma' \vdash \tau : \kappa$;
- *if* $\Gamma, (u_1 : \sigma_1) \simeq (u_2 : \sigma_2), \Gamma' \vdash^p a : \tau \Rightarrow \Delta$, *then* $\Gamma, \Gamma' \vdash^p a : \tau \Rightarrow \Delta$;
- *if* $\Gamma, (u_1 : \sigma_1) \simeq (u_2 : \sigma_2), \Gamma' \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$, *then* $\Gamma, \Gamma' \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$;
- *if* $\Gamma, (u_1 : \sigma_1) \simeq (u_2 : \sigma_2), \Gamma' \vdash u'_1 : \tau_1 \simeq u'_2 : \tau_2$, *then* $\Gamma, \Gamma' \vdash u'_1 : \tau_1 \simeq u'_2 : \tau_2$;
- *if* $\mathsf{TermsEqual}(\Gamma, (u_1 : \sigma_1) \simeq (u_2 : \sigma_2), \Gamma' \vdash^p a_1 : \tau_1 \simeq a_2 : \tau_2)$, *then* $\mathsf{TermsEqual}(\Gamma, \Gamma' \vdash^p a_1 : \tau_1 \simeq a_2 : \tau_2)$.

Lemma 6.9 (Substitution, computed results)**.** *Consider* Γ, Γ' *and* p, u *and* τ *such that* $\vdash \Gamma, (u)*p : \tau, \Gamma'$. *Suppose* $\Gamma, (u \simeq \mathsf{True}) : \mathsf{bool} \vdash u_0 : \tau$. *Then,*

- *we have* $\vdash \Gamma, \Gamma'[*p \leftarrow u_0]$*;*
- *if* $\Gamma, (u)*p : \tau, \Gamma' \vdash \tau : \kappa$*, then* $\Gamma, \Gamma'[*p \leftarrow u_0] \vdash \tau[*p \leftarrow u_0] : \kappa[*p \leftarrow u_0]$*;*
- *if* $\Gamma, (u)*p : \tau, \Gamma' \vdash^q a : \tau \Rightarrow \Delta$*, then* $\Gamma, \Gamma'[*p \leftarrow u_0] \vdash q : a[*p \leftarrow u_0]\tau[*p \leftarrow u_0]\Delta[*p \leftarrow u_0]$.

Moreover, consider u_1*,* u_2 *such that* $\Gamma, (u \simeq \mathsf{True}) : \mathsf{bool} \vdash u_1 : \tau$*,* $\Gamma, (u \simeq \mathsf{True}) : \mathsf{bool} \vdash u_2 : \tau$ *and* $\Gamma \vdash u_1 : \tau \simeq u_2 : \tau$*. Then,*

- *if* $\Gamma, (u)*p : \tau, \Gamma' \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$*, then* $\Gamma, \Gamma'[*p \leftarrow u_1] \vdash \tau_1[*p \leftarrow u_1] : \kappa_1[*p \leftarrow u_1] \simeq \tau_2[*p \leftarrow u_2] : \kappa_2[*p \leftarrow u_2]$*;*
- *if* $\Gamma, (u)*p : \tau, \Gamma' \vdash u_1' : \tau_1 \simeq u_2' : \tau_2$*, then* $\Gamma, \Gamma'[*p \leftarrow u_1] \vdash u_1'[*p \leftarrow u_1] : \tau_1[*p \leftarrow u_1] \simeq u_2'[*p \leftarrow u_2] : \tau_2[*p \leftarrow u_2]$*;*
- *if* $\mathsf{TermsEqual}(\Gamma, (u)*p : \tau, \Gamma' \vdash^q a_1 : \tau_1 \simeq a_2 : \tau_2)$*, then* $\mathsf{TermsEqual}(\Gamma, \Gamma'[*p \leftarrow u_1] \vdash^q a_1[*p \leftarrow u_1] : \tau_1[*p \leftarrow u_1] \simeq a_2[*p \leftarrow u_2] : \tau_2[*p \leftarrow u_2])$.

Lemma 6.10 (Substitution, path variables)**.** *Consider* Γ, Γ' *and* π *such that* $\vdash \Gamma, \pi, \Gamma'$*. Suppose* $p \perp \Gamma, \Gamma'$*. Then,*

- *we have* $\vdash \Gamma, \Gamma'[\pi \leftarrow p]$*;*
- *if* $\Gamma, \pi, \Gamma' \vdash \tau : \kappa$*, then* $\Gamma, \Gamma'[\pi \leftarrow p] \vdash \tau[\pi \leftarrow p] : \kappa[\pi \leftarrow p]$*;*
- *if* $\Gamma, \pi, \Gamma' \vdash^q a : \tau \Rightarrow \Delta$*, then* $\Gamma, \Gamma'[\pi \leftarrow p] \vdash^{q[\pi \leftarrow p]} a[\pi \leftarrow p] : \tau[\pi \leftarrow p] \Rightarrow \Delta[\pi \leftarrow p]$.
- *if* $\Gamma, \pi, \Gamma' \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$*, then* $\Gamma, \Gamma'[\pi \leftarrow p] \vdash \tau_1[\pi \leftarrow p] : \kappa_1[\pi \leftarrow p] \simeq \tau_2[\pi \leftarrow p] : \kappa_2[\pi \leftarrow p]$*;*
- *if* $\Gamma, \pi, \Gamma' \vdash u_1' : \tau_1 \simeq u_2' : \tau_2$*, then* $\Gamma, \Gamma'[\pi \leftarrow p] \vdash u_1'[\pi \leftarrow p] : \tau_1[\pi \leftarrow p] \simeq u_2'[\pi \leftarrow p] : \tau_2[\pi \leftarrow p]$*;*
- *if* $\mathsf{TermsEqual}(\Gamma, \pi, \Gamma' \vdash^q a_1 : \tau_1 \simeq a_2 : \tau_2)$*, then* $\mathsf{TermsEqual}(\Gamma, \Gamma'[\pi \leftarrow p] \vdash^{q[\pi \leftarrow p]} a_1[\pi \leftarrow p] : \tau_1[\pi \leftarrow p] \simeq a_2[\pi \leftarrow p] : \tau_2[\pi \leftarrow p])$.

Proof. By induction on the derivations. When extending the environment, use weakening on the thing we are substituting. When reaching a variable rule for the substituted variable (*e.g.* Var), replace the rule by the typing derivation. For the corresponding equality rule (*e.g.* Eq-Var), use reflexivity and replace the rule by the derivation obtained.

For computed results, use substitution with the equality in Rule Reuse-Present after weakening. □

6.3.2 Extraction

We can now prove a useful result: the environments and types appearing in type well-formedness and typing derivations are well-formed and the types given in equality judgments are correct.

Lemma 6.11 (Extraction for environment well-formedness)**.** *The following all imply* $\vdash \Gamma$*:*

- $\Gamma \vdash \tau : \kappa$

- $\Gamma \vdash^{\pi} a : \tau \Rightarrow \Delta$ *(this additionally implies* $\vdash \Gamma, \Delta$*)*
- $\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$
- $\Gamma \vdash u_1 : \tau_1 \simeq u_2 : \tau_2$
- $\mathsf{TermsEqual}(\Gamma \vdash^{p} a_1 : \tau_1 \simeq a_2 : \tau_2)$

Proof. By induction on the derivation. There is always an hypothesis that implies $\vdash \Gamma, \Gamma'$ with Γ' some (optional) extra bindings. From this we can derive $\vdash \Gamma$ by inverting the derivation. □

The following two lemmas need to be proved by mutual induction:

Lemma 6.12 (Extraction for type well-formedness)**.** *Suppose* $\Gamma \vdash^{\pi} a : \tau \Rightarrow \Delta$*. Then, there exists* κ *such that* $\Gamma \vdash \tau : \kappa$*. Suppose* $\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$*. Then,* $\Gamma \vdash \tau_1 : \kappa_1$ *and* $\Gamma \vdash \tau_2 : \kappa_2$*. Suppose* $\Gamma \vdash u_1 : \tau_1 \simeq u_2 : \tau_2$*. Then, there exist* κ_1 *and* κ_2 *such that* $\Gamma \vdash \tau_1 : \kappa_1$ *and* $\Gamma \vdash \tau_2 : \kappa_2$*.*

Lemma 6.13 (Extraction for well-typing)**.** *Suppose* $\Gamma \vdash u_1 : \tau_1 \simeq u_2 : \tau_2$*. Then,* $\Gamma \vdash u_1 : \tau_1$ *and* $\Gamma \vdash u_2 : \tau_2$*.*

Proof. By mutual induction, using substitution when needed. □

6.3.3 Subject reduction

We can now proceed to proving subject reduction for eML. Compared to subject reduction for ML, we need to handle both labels and equalities. Compared to subject reduction for labeled ML, we need to ensure that results are substituted with terms of the right type and that the reuse conditions stay true.

The main difficulty is to prove that the coercions we distribute in the subterms when reducing actually correspond to valid equalities. For example, we need to prove that when $\Gamma \vdash \tau_1 \to \tau_2 : \kappa \simeq \tau_1' \to \tau_2' : \kappa'$, then $\Gamma \vdash \tau_1 : \mathsf{Typ} \simeq \tau_2 : \mathsf{Typ}$ and $\Gamma \vdash \tau_1' : \mathsf{Typ} \simeq \tau_2' : \mathsf{Typ}$.

Lemma 6.14 (Projection)**.** *Consider an environment* Γ *and types* $(\tau_i)^i$ *and* $(\tau_i')^i$*.*

- *If* $\Gamma \vdash \tau_1 \to \tau_2 : \kappa \simeq \tau_1' \to \tau_2' : \kappa'$*, then* $\Gamma \vdash \tau_1 : \mathsf{Typ} \simeq \tau_1' : \mathsf{Typ}$ *and* $\Gamma \vdash \tau_2 : \mathsf{Typ} \simeq \tau_2' : \mathsf{Typ}$*.*
- *If* $\Gamma \vdash \forall(\alpha : \mathsf{Typ})\, \tau : \kappa \simeq \forall(\alpha : \mathsf{Typ})\, \tau' : \kappa'$*, then* $\Gamma, \alpha : \mathsf{Typ} \vdash \tau : \mathsf{Sch} \simeq \tau' : \mathsf{Sch}$*.*
- *If* $\Gamma \vdash \zeta\,(\tau_i)^i : \kappa \simeq \zeta\,(\tau_i')^i : \kappa'$*, then* $(\Gamma \vdash \tau_i : \mathsf{Typ} \simeq \tau_i' : \mathsf{Typ})^i$*.*

Proof. The three cases are similar. We give the proof for universal quantification.

Let us define a function tail as follows:

$$\begin{array}{rcl} \mathsf{tail}(\forall(\alpha : \mathsf{Typ})\, \tau) & = & \tau \\ \mathsf{tail}(\mathsf{match}\ a\ \mathsf{with}\ (P_i \to \tau_i)^i) & = & \mathsf{match}\ a\ \mathsf{with}\ (P_i \to \mathsf{tail}(\tau_i))^i \\ \mathsf{tail}(\tau) & = & \mathsf{Any} \end{array}$$

Here, Any stands for some well-formed typed. For example, we may take $\forall(\alpha : \mathsf{Typ})\, \alpha$ for Any. The function tail returns the type below the universal

quantifier, but it is also able to look under pattern matching, as if the quantifier were lifted out of the pattern matching.

Then, we show by induction on type equality derivations that whenever $\Gamma \vdash \tau : \kappa \simeq \tau' : \kappa'$, then $\Gamma \vdash \mathsf{tail}(\tau) : \mathsf{Sch} \simeq \mathsf{tail}(\tau') : \mathsf{Sch}$ (or Typ for the case of type constructors and function types). This implies the result.

- For TEq-Trans, apply transitivity on the tails.
- For TEq-Split, split on the same term to prove the equality of the tails.
- For TEq-Sym, apply symmetry on the tails.
- For TEq-Match, apply the induction hypothesis on the branches. The tails will start with a pattern matching on the same two equal terms. Then, we can use TEq-Match to prove that the tails are equal.
- For TEq-ReduceMatch, note that the pattern matching in the left-hand tail will reduce in the same way as the pattern matching in the original term.
- TEq-All has equality of the tails as a premise.
- For the other congruence rules, the tails of the two terms are Any, which is equal to itself by reflexivity.
- For TEq-SubTyp, apply induction to the subderivation.

□

We are now able to prove that two equal terms have equal types:

Lemma 6.15 (Equal terms have equal types)**.** *Suppose* $\Gamma \vdash u_1 : \tau_1 \simeq u_2 : \tau_2$. *Then, there exists* κ_1 *and* κ_2 *such that* $\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$.

Proof. By induction on the derivation. This is immediate for most rules by applying the lemma inductively on the hypotheses. For Eq-Coerce and Eq-Trans, use transitivity. For Eq-Split and Eq-Match, split on the same term using TEq-Split. For Eq-TApp, use Lemma 6.14 to deduce equality of σ_1 and σ_2 from the equality of $\forall(\alpha : \mathsf{Typ})\, \sigma_1$ and $\forall(\alpha : \mathsf{Typ})\, \sigma_1$, then apply substitution. □

Now we'll prove subject reduction. To pass to context, we need to prove that the equalities introduced using one term are preserved when the term reduces. For non-expansive reductions, we will show that reduction preserves equality of the reusable versions of the terms. For expansive reductions, they are represented in other terms by their label: we can then use substitution on the label.

The typing derivation are syntax-directed, except for the rule Coerce. Thus, we will often resort to the following pattern of reasoning:

Lemma 6.16 (Extracting a coercion)**.** *Suppose we have a derivation* $\Gamma \vdash a : \tau$. *Then, it has a subderivation* $\Gamma \vdash a : \tau'$ *whose first rule is not* Coerce *(so, is syntax-directed), and* $\Gamma \vdash \tau : \kappa \simeq \tau' : \kappa'$.

Proof. By induction:

- Suppose the derivation does not start with Coerce. Then, we can take $\tau' = \tau$. Since $\Gamma \vdash a : \tau$, there exists a kind κ such that $\Gamma \vdash \tau : \kappa$. Then, by reflexivity (Lemma 6.4), τ is equal to itself: $\Gamma \vdash \tau : \kappa \simeq \tau : \kappa$.

- Otherwise, the derivation is an application of Coerce to a derivation of $\Gamma \vdash a : \tau'$, and we have $\Gamma \vdash \tau : \kappa \simeq \tau' : \kappa'$. By induction hypothesis on the subderivation, there exists τ'' and a subderivation of $\Gamma \vdash a : \tau''$ starting with a syntax-directed rule. Then, it is a subderivation of the initial derivation too, and we have $\Gamma \vdash \tau' : \kappa' \simeq \tau'' : \kappa''$. We obtain $\Gamma \vdash \tau : \kappa \simeq \tau'' : \kappa''$ by transitivity. □

When reducing a term typed $\Gamma \vdash^{\pi} a : \tau \Rightarrow \Delta$, the reduction might change the conditions in the output Δ, the types, and some labels may disappear. For instance, reducing $\mathsf{match\ True\ with\ True} \to f^{p_1}\ x \mid \mathsf{False} \to g^{p_2}\ y$ to $f^{p_1}\ x$ removes the label p_2 and changes the condition for label p_1 from $\mathsf{match\ True\ with\ True} \to \mathsf{True} \mid \mathsf{False} \to \mathsf{False}$ to True. We ensure that the outputs are still equivalent: notably, we need to ensure that the type for non-eliminated labels stay consistent (*i.e.* are equivalent for type equality), and that, for eliminated labels, their conditions were already equivalent to False. This ensures that the Reuse-Present and Reuse-Absurd typing rules still apply in the context where the reduced subterm appears.

Definition 6.3 (Reduced outputs)**.** *Let Δ_1 and Δ_2 be outputs. We say that Δ_2 is a reduced output of Δ_1 in context Γ, noted $\Gamma \vdash \Delta_1 \supseteq \Delta_2$, if for all $p \in \Delta_2$, $p \in \Delta_1$ and for all $p \in \Delta_1$ such that $(u_1)*p : \tau_1 \in \Delta_1$, either $p \notin \Delta_2$ and $\Gamma \vdash u : \mathsf{bool} \simeq \mathsf{False} : \mathsf{bool}$ or $(u_2)*p : \tau_2 \in \Delta_2$ and we have both $\Gamma \vdash u_1 : \mathsf{bool} \simeq u_2 : \mathsf{bool}$ and $\Gamma, (u_1 \simeq \mathsf{True}) : \mathsf{bool} \vdash \tau_1 : \mathsf{Sch} \simeq \tau_2 : \mathsf{Sch}$.* ♢

Lemma 6.17 (Substitution of outputs)**.** *Suppose $\Gamma \vdash \Delta_1 \supseteq \Delta_2$. Then,*

- $\vdash \Gamma, \Delta_1, \Gamma'$ *implies* $\vdash \Gamma, \Delta_2, \Gamma'$*;*

- $\Gamma, \Delta_1, \Gamma' \vdash^{p} a : \tau \Rightarrow \Delta'$ *implies* $\Gamma, \Delta_2, \Gamma' \vdash^{p} a : \tau \Rightarrow \Delta'$*;*

- *and similarly for other judgments.*

Proof. By induction on the derivations. If something is independent of Δ_1, it is also independent of Δ_2. When the results are read (in Reuse-Present, Reuse-Absurd and their equivalent in the equality judgment), in the "present" case, the conditions are both true, so the types are equal and we can use coercion; in the "absent" case, the result stays absent. □

Lemma 6.18 (Subject reduction)**.** *Let Γ be an environment, a a term such that $\Gamma \vdash^{\pi} a : \tau \Rightarrow \Delta$. Suppose $a \overset{l}{\mapsto} b$. We can interpret l as a substitution. Then there exists Δ' such that $\Gamma \vdash^{\pi} b : \tau[l] \Rightarrow \Delta'$ and:*

- *If $l = 0$, we have $\Gamma \vdash \Delta \supseteq \Delta'$.*

- *If $l = q \mapsto u$, we have $\Delta' = (\Delta - q)[*q \leftarrow u] \cup S$, where S is an environment of labels with all labels prefixed by q.*

Moreover, $\Gamma, \Delta' \vdash \mathsf{reuse}(a)[l] : \tau \simeq \mathsf{reuse}(b) : \tau$.

Proof. First, let us prove this for head reductions: we need to use projection (Lemma 6.14). Take for example reduction of a type-level application: we have $\Gamma \vdash (\Lambda(\alpha : \mathsf{Typ}).\, v)\ \tau_0 : \sigma$. Invert the derivation rules until we reach a rule that is not Coerce. Then, the last rule is necessarily TApp. By inverting the type derivation, we obtain $\Gamma \vdash \tau_0 : \mathsf{Typ}$, $\sigma = \tau[\alpha \leftarrow \tau_0]$ and $\Gamma \vdash \Lambda(\alpha : \mathsf{Typ}).\, v : \forall(\alpha : \mathsf{Typ})\, \tau$. After extracting a coercion, and inverting the last syntax directed rule, we obtain that there exists τ' such that $\Gamma, \alpha : \mathsf{Typ} \vdash v : \tau'$ and $\Gamma \vdash \forall(\alpha : \mathsf{Typ})\ \tau : \kappa \simeq \forall(\alpha : \mathsf{Typ})\ \tau : \kappa'$. Then we apply projection on this equality: we obtain $\Gamma, \alpha : \mathsf{Typ} \vdash \tau : \mathsf{Sch} \simeq \tau' : \mathsf{Sch}$. Thus, applying Coerce, we get $\Gamma, \alpha : \mathsf{Typ} \vdash v : \tau$. Then we conclude by substitution: $\Gamma \vdash v[\alpha \leftarrow \tau_0] : \tau[\alpha \leftarrow \tau_0]$. The equality of the reusable versions of the terms is guaranteed by Eq-ReduceTApp.

Let us also examine the reduction of a pattern matching, as it removes labels: if it reduces, the conditions for the output labels reduce too, and they reduce to False in the branches that are not selected, so the new outputs Δ' with the results in.discarded branches removed are reduced outputs of Δ.

For context: by induction on the context. The subterm that is reduced does not appear in the resulting type, but its reusable version may appear in an equality for typing another subterm that is not reduced. But we know by induction hypothesis that the reusable versions stay equal. The output changes, but stays equivalent, so any well-typed term with the old output is also well-typed with the new output. We need to show equality of the reused versions. This is immediate by congruence (for Eq-Fix, we require the equality on expansive terms to embed the equality on non-expansive terms). Finally, we need to show equivalence of the computed result sets. This is immediate by substituion/congruence (for Eq-Let and Eq-Match), and the fact that we combine equivalent computed result sets. □

6.3.4 Soudness via a logical relation for equality

In order to prove soundness, we need to show that, in evaluation contexts, there will never be an equality between, *e.g.*, an arrow type and a datatype, or between True and False. This is not an obvious result because it depends on the equalities in the environment: for example, if we have $\mathsf{True} = \mathsf{False}$ in the environment we can prove any equality.

We prove that these equalities cannot be proved in inhabited environments, *i.e.* typing environments for which we can provide an instanciation of the type and term variables with types and terms that satisfies the equality in the context. For this, we explain the meaning of equality in the empty environment.

We define a terminating subset of the reduction $\longrightarrow_\iota$ on non-expansive terms that only does non-expansive reduction. It is defined in Figure 6.11 and does not emit labels.

The reduction $\longrightarrow_\iota$ does not involve labels, since applications are not reduced. It is confluent. It also preserves types, since its head reduction is a subset of $\longrightarrow_\beta$.

We need to prove that $\longrightarrow_\iota$ terminates. Then, this gives a meaning to $\|a\|$, the normal form of a for $\longrightarrow_\iota$.

Lemma 6.19. *The reduction $\longrightarrow_\iota$ is strongly normalizing: there is no infinite reduction path for $\longrightarrow_\iota$ starting from a term or type (even ill-typed).*

$$(\Lambda(\alpha : \mathsf{Typ}).\, u)\ \tau \longrightarrow_\iota u[\alpha \leftarrow \tau] \qquad \mathsf{let}\ x = u\ \mathsf{in}\ a \longrightarrow_\iota a[x \leftarrow u]$$

$$\mathsf{match}\ d_j(\tau''_k)^k(u_i)^i\ \mathsf{with}\ (d_j(\tau_k)^k(x_{ji})^i \to a_j)^j \longrightarrow_\iota a_j[x_{ji} \leftarrow u_i]^i$$

$$\mathsf{match}\ d_j(\tau''_k)^k(u_i)^i\ \mathsf{with}\ (d_j(\tau_k)^k(x_{ji})^i \to \sigma_j)^j \longrightarrow_\iota \sigma_j[x_{ji} \leftarrow u_i]^i$$

$$\textsc{Context-Iota}\quad \frac{a \longrightarrow_\iota b}{\mathcal{C}[a] \longrightarrow_\iota \mathcal{C}[b]}$$

Figure 6.11: Non-expansive reduction

Proof. Note that this result is not restricted to well-typed terms and types. Implicitly, every reduction mentioned in this proof will be $\longrightarrow_\iota$.

We define the set $\mathcal{S}_\tau$ of strongly normalizing types: we will show that every type belongs in $\mathcal{S}_\tau$. We also define the set $\mathcal{S}_a$ as the largest set of terms such that:

- if $a \in \mathcal{S}_a$, a is strongly normalizing;
- if $a \in \mathcal{S}_a$ reduces to $\Lambda(\alpha : \mathsf{Typ}).\, u$, then for all types $\tau \in \mathcal{S}_\tau$, $u[\alpha \leftarrow \tau] \in \mathcal{S}_a$;
- if $a \in \mathcal{S}_a$ reduces to $d(\tau_j)^j(a_i)^i$, then for all i, a_i in $\mathcal{S}_a$.

These two sets are stable by reduction because the restrictions apply to all reductions of terms or types that are in the sets. Finally, let γ associate type variables to types and term variables to terms. We say that $\gamma \in \mathcal{S}_\gamma$ if for all $\alpha \in \mathsf{dom}(\gamma)$, $\gamma(\alpha) \in \mathcal{S}_\tau$ and for all $x \in \mathsf{dom}(\alpha)$, $\gamma(x) \in \mathcal{S}_a$.

We prove, by mututal induction on the structure of types and terms, the following results:

- for all types τ, if $\gamma \in \mathcal{S}_\gamma$, $\gamma(\tau) \in \mathcal{S}_\tau$;
- for all terms a, if $\gamma \in \mathcal{S}_\gamma$, $\gamma(a) \in \mathcal{S}_a$.

For types τ:

- If $\tau = \alpha$, if $\alpha \in \mathsf{dom}(\gamma)$, then $\gamma(\tau) = \gamma(\alpha) \in \mathcal{S}_\tau$, because $\gamma \in \mathcal{S}_\gamma$. Otherwise, τ does not reduce.
- If $\tau = \tau_1 \to \tau_2$ (similarly for $\tau = \forall(\alpha : \mathsf{Typ})\ \sigma$, and $\tau = \zeta\,(\tau_i)^i$), the reduction sequences from τ_1 and τ_2 are finite, thus the reductions from $\tau = \tau_1 \to \tau_2$ are finite (the arrow will never reduce).
- If $\tau = \mathsf{match}\ a\ \mathsf{with}\ (d_i(\tau_j)^j(x_{ik})^k \to \tau_i)^i$, then we have

  $$\gamma(\tau) = \mathsf{match}\ \gamma(a)\ \mathsf{with}\ (d_i(\tau_j)^j(x_{ik})^k \to \gamma(\tau_i))^i$$

  By induction hypothesis, the subterm $\gamma(a)$ and the subtypes $\gamma(\tau_i)$ do not have an infinite reduction sequence. Thus any infinite reduction sequence includes a head reduction. Suppose the reduction is from $\tau' = \mathsf{match}\ b\ \mathsf{with}\ (d_i(\tau_j)^j(x_{ik})^k \to \tau'_i)^i$. Then, $b = d_i(\tau'_j)^j(u_k)^k$. The head reduction then reduces τ' to $\tau'_i[x_{ik} \leftarrow u_k]^k$. Let $\gamma' = \gamma[x_{ik} \leftarrow u_k]^k$. We have $\gamma' \in \mathcal{S}_\gamma$, because for all k, $u_k \in \mathcal{S}_a$. Thus, by induction hypothesis, $\gamma'(\tau_i) \in \mathcal{S}_\tau$. But $\tau'_i[x_{ik} \leftarrow u_k]^k$ is a possible reduction of $\gamma'(\tau_i)$, thus $\tau'_i[x_{ik} \leftarrow u_k]^k \in \mathcal{S}_\tau$.

$$\textsc{EnvT-Empty}\quad \frac{}{\gamma \vdash \varnothing}$$

$$\textsc{EnvT-Var}\quad \frac{\gamma \vdash \Gamma \qquad x \# \Gamma \qquad \vdash \gamma(x) : \gamma(\tau) \qquad \gamma(x) \text{ is nonexpansive}}{\gamma \vdash \Gamma, x : \tau}$$

$$\textsc{EnvT-TVar}\quad \frac{\gamma \vdash \Gamma \qquad \alpha \# \Gamma \qquad \vdash \gamma(\alpha) : \gamma(\kappa)}{\gamma \vdash \Gamma, \alpha : \kappa}$$

$$\textsc{EnvT-Eq}\quad \frac{\gamma \vdash \Gamma \qquad \vdash \gamma(u_1) : \gamma(\tau) \simeq \gamma(u_2) : \gamma(\tau)}{\gamma \vdash \Gamma, (u_1 \simeq u_2) : \tau}$$

$$\textsc{Env-Path-Var}\quad \frac{\gamma \vdash \Gamma \qquad \gamma(\pi) \perp \Gamma}{\gamma \vdash \Gamma, \pi}$$

$$\textsc{Env-Result}\quad \frac{\gamma \vdash \Gamma \qquad (\vdash \gamma(u) : \mathsf{bool} \simeq \mathsf{True} : \mathsf{bool}) \implies (\vdash \gamma(p) : \gamma(\tau))}{\gamma \vdash \Gamma, (u){*}p : \tau}$$

Figure 6.12: Environment typing

For terms a:

- The cases of variables and pattern matching are similar to their type equivalents.
- Labels stay labels and are never substituted.
- For function definition and application, the subterms are strongly normalizing by induction hypothesis, and they never reduce to a type abstraction or a type constructor.
- For constructors, by induction hypothesis on the fields, they are all in $\mathcal{S}_a$. A constructor never reduces to a type abstraction.
- If $a = \Lambda(\alpha : \mathsf{Typ}).\, u$, then, by induction hypothesis, $\gamma(u)$ has no infinite reduction sequence. Moreover, for every type τ, if $\gamma' = \gamma[\alpha \leftarrow \tau]$, then $\gamma'(u)$ has no infinite reduction sequence. Thus, $u[\alpha \leftarrow \tau] \in \mathcal{S}_a$. Then $a \in \mathcal{S}_a$.
- If $a = a'\ \tau$, there are no infinite reduction sequences from a' and τ by induction hypothesis, thus an infinite reduction sequence from a must include a head reduction, and so must be a reduction sequence that yields a constructor or a type abstraction. Suppose the head reduction is from $b = (\Lambda(\alpha : \mathsf{Typ}).\, u)\ \tau'$. Then b reduces to $u[\alpha \leftarrow \tau']$. Let $\gamma' = \gamma[\alpha \leftarrow \tau']$. We have $\gamma' \in \mathcal{S}_\gamma$ because $\tau' \in \mathcal{S}_\tau$ by induction hypothesis. Thus by induction hypothesis, we have $\gamma'(\sigma) \in \mathcal{S}_\tau$ and $\gamma'(u) \in \mathcal{S}_a$. $\square$

For an environment Γ and a mapping from term variables to non-expansive terms, type variables to types, path variables to paths, and paths to non-expansive terms γ, we define a judgment $\gamma \vdash \Gamma$ that indicates that the types and terms in γ are closed and well-formed and define a valid instance of Γ. The rules defining this judgment are given in Figure 6.12.

We will prove that every well-typed closed non-expansive term reduces for $\longrightarrow_\iota$ to a value of the correct syntactic shape for its type. We will also prove

that equal non-expansive terms normalize to values, and that these values are semantically equal according to some relation. Although tempting, we cannot take this relation to only relate identical terms (up to renaming). Indeed, in function definitions $\lambda(x : \tau).a$, a is not closed anymore, and does not necessarily reduce to a value (for example, $a = \mathsf{match}\ x\ \mathsf{with}\ \mathsf{Unit} \to \mathsf{Unit}$ is equal to Unit, but reduction does not prove the equality). Since the terms inside function definitions are never taken out of the definition by the reduction $\longrightarrow_\iota$, we can simply ignore them (as long as they are syntactically equal). We do not actually define a (unary) set of terms with the correct syntactic shape: instead, we use reflexivity and say that for any closed well-typed term a of type τ, the pair (a, a) is in $\mathcal{E}[\tau]$. The set $\mathcal{E}[\tau]$ of comparable coerced terms is defined in terms of $\mathcal{V}[\tau]$, the set of comparable values at type τ. Two equal coerced terms may have different, but equal, types. This is not a problem, as we prove that, for two closed equal types τ_1 and τ_2, $\mathcal{V}[\tau_1] = \mathcal{V}[\tau_2]$.

The definition implementing these ideas is given in Figure 6.13. We limit our definitions, for $\mathcal{E}[\tau]$, to closed well-typed non-expansive terms equal in the empty environment, and for $\mathcal{V}[\tau]$ to values of types equal to τ. To keep the definitions readable, we implicitly limit the sets we define to only contain pairs of terms verifying these conditions, *i.e.* we allow ourselves to omit the typing conditions. The sets $\mathcal{E}[\tau]$ and $\mathcal{V}[\tau]$ are defined as the least fixed point of the equations given here (it exists, because we never use the interpretation in a contravariant position). For the needs of the proof, we also define a relation on environments γ_1, γ_2: we say $(\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]$ if the terms and types in the two environments are syntactically equal, related and of the appropriate type. In the same way, we require the environments to be well-typed.

To start with, we need the following technical lemma on the interpretations. Proving that an environment is related to itself is important to build environments to apply transitivity in.

Lemma 6.20 (Left slicing)**.**

- *If* $(u_1, u_2) \in \mathcal{E}[\tau]$, $(u_1, u_1) \in \mathcal{E}[\tau]$.
- *If* $(v_1, v_2) \in \mathcal{V}[\tau]$, $(v_1, v_1) \in \mathcal{V}[\tau]$.
- *If* $(\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]$, $(\gamma_1, \gamma_1) \in \mathcal{G}[\Gamma]$.

Proof. For the second item, by induction of the derivation of $(v_1, v_2) \in \mathcal{V}[\tau]$. The syntactic typing/equality conditions are immediately true (by reflexivity of equality, Lemma 6.4).

- For $(v_1, v_2) \in \mathcal{V}[\tau_1 \to \tau_2]$: v_1 has the right syntactic form, and there are no other conditions.
- For $(v_1, v_2) \in \mathcal{V}[\forall(\alpha : \mathsf{Typ})\ \sigma]$: we have $v_1 = \Lambda(\alpha : \mathsf{Typ}).\ w_1$ and $v_2 = \Lambda(\alpha : \mathsf{Typ}).\ w_2$. Consider $\vdash \tau : \mathsf{Typ}$. We have $(w_1[\alpha \leftarrow \tau], w_2[\alpha \leftarrow \tau]) \in \mathcal{V}[\sigma[\alpha \leftarrow \tau_1]]$: by induction hypothesis we conclude that $(v_1[\alpha \leftarrow \tau], v_1[\alpha \leftarrow \tau]) \in \mathcal{V}[\sigma[\alpha \leftarrow \tau_1]]$.
- For $(v_1, v_2) \in \mathcal{V}[\zeta\,(\tau_j)^j]$, we similarly apply the induction hypothesis on all fields.
- For a pattern matching: the selection of the branch is only decided by the type, then we can apply the induction hypothesis on the branch.

$$
\begin{array}{lcl}
\mathcal{E}[\tau] & \subseteq & \{(u_1, u_2) \mid\vdash u_1 : \tau \simeq u_2 : \tau\} \\
\mathcal{E}[\tau] & = & \{(u_1, u_2) \mid (\|u_1\|, \|u_2\|) \in \mathcal{V}[\tau])\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{V}[\tau] & \subseteq & \{(v_1, v_2) \mid\vdash v_1 : \tau \simeq v_2 : \tau\} \\
\mathcal{V}[\tau_1 \to \tau_2] & = & \{(\mathsf{fix}^\pi\, x\, (y : \tau_1') : \tau_2' \,.\, a', \mathsf{fix}^\pi\, x\, (y : \tau_1'') : \tau_2'' \,.\, a'')\} \\
\mathcal{V}[\forall(\alpha : \mathsf{Typ})\, \sigma] & = & \\
\quad \left\{ \begin{pmatrix} \Lambda(\alpha : \mathsf{Typ}).\, u_1 \\ \Lambda(\alpha : \mathsf{Typ}).\, u_2 \end{pmatrix} \middle| \begin{array}{l} \forall(\tau_1, \tau_2)\, (\vdash \tau_1 : \mathsf{Typ} \simeq \tau_2 : \mathsf{Typ} \wedge \mathcal{V}[\tau_1] = \mathcal{V}[\tau_2]) \\ \quad \Longrightarrow\ (u_1[\alpha \leftarrow \tau_1], u_2[\alpha \leftarrow \tau_2]) \in \mathcal{V}[\sigma[\alpha \leftarrow \tau_1]] \end{array} \right\} & & \\
\mathcal{V}[\zeta\, (\tau_j)^j] & = & \\
\quad \left\{ \begin{pmatrix} d(\tau_{1j})^j (v_{1i})^i \\ d(\tau_{2j})^j (v_{2i})^i \end{pmatrix} \middle| \begin{array}{l} (\vdash d : \forall(\alpha_j : \mathsf{Typ})^j\, (\tau_i)^i \to \zeta\, (\alpha_j)^j) \\ \wedge\ ((v_{1i}, v_{2i}) \in \mathcal{V}[\tau_i[\alpha_j \leftarrow \tau_j]^j])^i \end{array} \right\} & & \\
\mathcal{V}[\mathsf{match}\ a\ \mathsf{with}\ (d_i(\tau_k)^k (x_{ij})^j \to \sigma_i)^i] = & & \\
\quad \begin{cases} \mathcal{V}[\sigma_i[x_{ij} \leftarrow v_j]^j] & \text{if} \begin{cases} \wedge & \|a\| = d_i(\tau_k')^k (v_j)^j \\ & \vdash d_i : \forall(\alpha_k : \mathsf{Typ})^k\, (\tau_j)^i \to \zeta\, (\alpha_k)^k \end{cases} \\ \varnothing & \text{otherwise} \end{cases} & &
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{G}[\Gamma] & \subseteq & \{(\gamma_1, \gamma_2) \mid \gamma_1 \vdash \Gamma \wedge \gamma_2 \vdash \Gamma\} \\
\mathcal{G}[\varnothing] & = & \varnothing \\
\mathcal{G}[\Gamma, \alpha : \mathsf{Typ}] & = & \\
\quad \{(\gamma_1[\alpha \leftarrow \tau_1], \gamma_2[\alpha \leftarrow \tau_2]) \mid (\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma] \wedge \mathcal{V}[\tau_1] = \mathcal{V}[\tau_2] \wedge \vdash \tau_1 : \mathsf{Typ} \simeq \tau_2 : \mathsf{Typ}\} & & \\
\mathcal{G}[\Gamma, x : \tau] & = & \\
\quad \{(\gamma_1[\alpha \leftarrow u_1], \gamma_2[\alpha \leftarrow u_2]) \mid (\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma] \wedge (u_1, u_2) \in \mathcal{V}[\gamma_1(\tau)]\} & & \\
\mathcal{G}[\Gamma, (u_1 \simeq u_2) : \tau] & = & \{(\gamma_1, \gamma_2) \mid (\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma] \wedge (\gamma_1(u_1), \gamma_2(u_2)) \in \mathcal{E}[\gamma_1(\tau)]\}
\end{array}
$$

Figure 6.13: Interpretation of types

The proof of the result on environments similarly follows the structure of $\mathcal{G}[\Gamma]$.

□

We also need to prove that the interpretations are transitive:

Lemma 6.21 (Transitivity)**.**

- *If* $(v_1, v_2) \in \mathcal{V}[\tau]$ *and* $(v_2, v_3) \in \mathcal{V}[\tau]$*, then* $(v_1, v_3) \in \mathcal{V}[\tau]$.
- *If* $(u_1, u_2) \in \mathcal{E}[\tau]$ *and* $(u_2, u_3) \in \mathcal{E}[\tau]$*, then* $(u_1, u_3) \in \mathcal{E}[\tau]$.

Proof. Transitivity of $\mathcal{E}[\tau]$ is immediately implied by transitivity of $\mathcal{V}[\tau]$. The typing conditions are immediately satisfied by transitivity. The rest of the proof is done by mutual induction.

For transitivity of $\mathcal{V}[\tau]$, consider the different cases. Similarly to the previous lemma (6.20), it suffices to follow the structure of the derivation. □

Lemma 6.22 (Correcteness of the intepretation)**.**

- *If* $\Gamma \vdash u : \tau$*, then for all* $(\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]$*,* $(\gamma_1(u), \gamma_2(u)) \in \mathcal{E}[\gamma_1(\tau)]$.
- *If* $\Gamma \vdash u_1 : \tau_1 \simeq u_2 : \tau_2$ *then for all* $(\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]$*,* $(\gamma_1(u_1), \gamma_2(u_2)) \in \mathcal{E}[\gamma_1(\tau_1)]$.
- *If* $\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$*, then for all* $(\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]$*,* $\mathcal{V}[\gamma_1(\tau_1)] = \mathcal{V}[\gamma_2(\tau_2)]$. *Moreover, if* $\gamma \vdash \Gamma$*, then* $(\gamma, \gamma) \in \mathcal{G}[\Gamma]$.

Proof. We prove the three first results by mutual induction on the derivations. The first result is essentially a special case of the second: we could indeed obtain it by first applying reflexivity to deduce that u is equal to itself, although that changes the derivation so that we cannot proceed by induction. We will focus on the second result on term equality instead.

For term equality: suppose $\Gamma \vdash u_1 : \tau_1 \simeq u_2 : \tau_2$, and $(\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]$. Let us consider some of the different rules.

- For Eq-Trans, we have premises $\Gamma \vdash u_1 : \tau_1 \simeq u_3 : \tau_3$ and $\Gamma \vdash u_3 : \tau_3 \simeq u_2 : \tau_2$. By Lemma 6.20, we have $(\gamma_1, \gamma_1) \in \mathcal{G}[\Gamma]$. Thus, by induction hypothesis on the first premise, $(\gamma_1(u_1), \gamma_1(u_3)) \in \mathcal{E}[\gamma_1(\tau_1)]$. By induction hypothesis on the second premise, $(\gamma_1(u_3), \gamma_2(u_2)) \in \mathcal{E}[\gamma_1(\tau_3)]$. We also have $\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_3 : \kappa_3$ (Lemma 6.15), thus $\mathcal{E}[\gamma_1(\tau_1)] = \mathcal{E}[\gamma_1(\tau_3)]$. Then, by transitivity of $\mathcal{E}[\gamma_1(\tau_1)]$ (Lemma 6.21), $(\gamma_1(u_1), \gamma_2(u_3)) \in \mathcal{E}[\gamma_1(\tau_1)]$.
- For Eq-Split, consider the premise $\Gamma \vdash u : \zeta\,(\tau_k)^k$. By induction hypothesis, $(\gamma_1(u), \gamma_2(u)) \in \mathcal{E}[\gamma_1(\zeta\,(\tau_k)^k)]$. Thus, $(\|\gamma_1(u)\|, \|\gamma_2(u)\|) \in \mathcal{V}[\gamma_1(\zeta\,(\tau_k)^k)]$. Then there exists a constructor $\vdash d : \forall(\alpha_k : \mathsf{Typ})^k(\tau_i)^i \to \zeta\,(\alpha_k)^k$ such that $\|\gamma_1(u)\| = d(\tau_{1k})^k(v_{1i})^i$, $\|\gamma_2(u)\| = d(\tau_{2k})^k(v_{2i})^i$, and for all i, $(v_{1i}, v_{2i}) \in \mathcal{E}[\tau_i[\alpha_j \leftarrow \tau_{1j}]^j]$. Suppose the premise corresponding to this branch is

  $$\Gamma, (x_i : \tau_i[\alpha_k \leftarrow \tau_k]^k)^i, (d(\tau_k)^k(x_i)^i \simeq u) : \zeta\,(\tau_k)^k \vdash u_1 : \tau_1 \simeq u_2 : \tau_2$$

  Consider the environments $\gamma_1' = \gamma_1[x_i \leftarrow v_{1i}]^i$ and $\gamma_2' = \gamma_2[x_i \leftarrow v_{2i}]^i$. We have:

  $$(\gamma_1', \gamma_2') \in \mathcal{G}[\Gamma, (x_i : \tau_i[\alpha_k \leftarrow \tau_k]^k)^i, (d(\tau_k)^k(x_i)^i : u) \simeq (\zeta\,(\tau_k)^k :])$$

  Thus, $(\gamma_1'(u_1), \gamma_1'(u_2)) \in \mathcal{E}[\gamma_1'(\tau_1)]$. Since $\gamma_i'(u_i) = \gamma(u_i)$ for $i \in \{1, 2\}$, we have $(\gamma_1(u_1), \gamma_2(u_2)) \in \mathcal{E}[\gamma_1(\tau_1)]$.

- For Eq-Subst, this is true by the hypothesis $(\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]$.
- For Eq-Coerce, use (by induction hypothesis) the equality of the interpretation of equal types.
- For Eq-Var on x, by hypothesis $(\gamma_1(x), \gamma_2(x)) \in \mathcal{E}[\gamma_1(\tau)]$. Then we have $(\|\gamma_1(x)\|, \|\gamma_2(x)\|) \in \mathcal{V}[\gamma_1(\tau)]$.
- For congruence rules, let us consider for example Eq-Let. We have $u_1 = \mathsf{let}\ x = u_{01}\ \mathsf{in}\ u_{11}$ and $u_2 = \mathsf{let}\ x = u_{02}\ \mathsf{in}\ u_{12}$, with premises $\Gamma \vdash u_{01} : \sigma_1 \simeq u_{02} : \sigma_2$ and $\Gamma, x_1 : \tau_1, (x \simeq u_{01}) : \sigma_1 \vdash u_{11} : \tau_1 \simeq u_{12} : \tau_2$. Apply the induction hypothesis on the first premise: we have $(\|\gamma_1(u_{01})\|, \|\gamma_2(u_{02})\|) \in \mathcal{V}[\gamma_1(\tau_1)]$. Consider $\gamma_1' = \gamma_1[x \leftarrow \|\gamma_1(u_{01})\|]$ and $\gamma_2' = \gamma_2[x \leftarrow \|\gamma_2(u_{02})\|]$. We have both $\gamma_1' \vdash \Gamma, x : \sigma_1$ and $\gamma_2' \vdash \Gamma, x : \sigma_1$, since $\Gamma \vdash \gamma_2(\sigma_1) : \kappa \simeq \gamma_2(\sigma_1') : \kappa'$. Then, $(\gamma_1', \gamma_2') \in \mathcal{G}[\Gamma, x : \sigma_1, (x \simeq \sigma_1) : a_1]$, and by induction hypothesis, $(\gamma_1'(u_{11}), \gamma_2'(u_{12})) \in \mathcal{E}[\gamma_1'(\tau_1)]$. Finally, $\|\gamma_1'(u_{11})\| = \|\gamma_1(u_{11})[x \leftarrow \|u_{01}\|]\| = \|\gamma_1(\mathsf{let}\ x_1 = u_{01}\ \mathsf{in}\ u_{11})\|$ and similarly for u_2. Thus, $(\gamma_1(\|\mathsf{let}\ x_1 = u_{01}\ \mathsf{in}\ u_{11}\|), \gamma_2(\|\mathsf{let}\ x_2 = u_{02}\ \mathsf{in}\ u_{12}\|) \in \mathcal{E}[\gamma_1(\tau_1)]$.
- For reduction rules, the two terms normalize to the same thing and have the same type. We conclude by applying the first result on the typing derivation of one of the sides of the equality.

For equality on types, consider the last rule in the derivation:

- For TEq-Trans, proceed as for Eq-Trans: suppose we have $\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$ and $\Gamma \vdash \tau_2 : \kappa_2 \simeq \tau_3 : \kappa_3$ and consider $(\gamma_1, \gamma_3) \in \mathcal{G}[\Gamma]$. By Lemma 6.20, $(\gamma_1, \gamma_1) \in \mathcal{G}[\Gamma]$, thus (applying the induction hypothesis on the first premise), $\mathcal{V}[\gamma_1(\tau_1)] = \mathcal{V}[\gamma_1(\tau_2)]$. From the second premise, we obtain $\mathcal{V}[\gamma_1(\tau_2)] = \mathcal{V}[\gamma_3(\tau_3)]$. Thus, $\mathcal{V}[\gamma_1(\tau_1)] = \mathcal{V}[\gamma_3(\tau_3)]$.
- For TEq-Split, select a branch and build an environment as for Eq-Split, then use the induction hypothesis on this branch.
- For congruence rules, take for instance TEq-All. We have $\Gamma, \alpha : \mathsf{Typ} \vdash \sigma_1 : \mathsf{Sch} \simeq \sigma_2 : \mathsf{Sch}$. We want to prove that, for a term u and a closed type τ such that $\vdash \tau : \mathsf{Typ}$, $u[\alpha \leftarrow \tau] \in \mathcal{V}[\gamma_1(\sigma_1)[\alpha \leftarrow \tau]]$ if and only if $u[\alpha \leftarrow \tau] \in \mathcal{V}[\gamma_2(\sigma_2)[\alpha \leftarrow \tau]]$. Consider, for $i \in \{1, 2\}$, $\gamma_i' = \gamma_i[\alpha \leftarrow \tau]$. Since $\mathcal{V}[\tau] = \mathcal{V}[\tau]$, $(\gamma_1', \gamma_2') \in \mathcal{G}[\Gamma, \alpha : \mathsf{Typ}]$. By induction hypothesis, $\mathcal{V}[\gamma_1'(\sigma_1)] = \mathcal{V}[\gamma_2'(\sigma_2)]$, and $\gamma_i'(\sigma_i) = \gamma_i(\sigma_i)[\alpha \leftarrow \tau]$.
- For TEq-Match, consider the premise on the term that is matched on: $\Gamma \vdash a_1 : \tau_1 \simeq a_2 : \tau_2$. Then, $\|\gamma_1(a_1)\|$ and $\|\gamma_2(a_2)\|$ both normalize to the same constructor. Then the correct branch can be selected as in TEq-Split.
- For TEq-ReduceMatch: compute the intepretations on both side, they are actually the same.

□

From the fact that the $\mathcal{V}[\tau]$ are distinct for different shapes of τ, we can conclude that, in the empty environment, it is impossible to prove an equality between, *e.g.*, arrow types and universally quantified types. This suffices to prove progress.

Lemma 6.23 (Inversion)**.** *Consider a value v such that $(\alpha_k : \mathsf{Typ})^k \vdash v : \tau$.*

- *If $\tau = \tau_1 \to \tau_2$, there exists a, τ_1', τ_2' such that $v = \mathsf{fix}^\pi\, x\, (y : \tau_1') : \tau_2' \,.\, a$.*
- *If $\tau = \forall(\alpha : \mathsf{Typ})\, \tau'$, there exists w such that $v = \Lambda(\alpha : \mathsf{Typ}).\, w$.*
- *If $\tau = \zeta\,(\tau_i)^i$, there exists a constructor $\vdash d : \forall(\alpha_i : \mathsf{Typ})^i\, (\tau_j)^j \to \zeta\,(\alpha_i)^i$, types $(\tau_i')^i$ and terms $(a_j)^j$ such that $a = d(\tau_i')^i(a_j)^j$.*

Proof. We will only look at the last case, the other cases are similar. Consider the different cases for v:

- If $v = \mathsf{fix}^\pi\, x\, (y : \tau_1) : \tau_2 \,.\, a$, then we have $(\alpha_k : \mathsf{Typ})^k \vdash v : \tau_1 \to \tau_2$, and $(\alpha_k : \mathsf{Typ})^k \vdash \zeta\,(\tau_i)^i : \kappa \simeq \tau_1 \to \tau_2 : \kappa'$. We can instantiate the α_k with, *e.g.* unit: we get $\vdash \zeta\,(\tau_i[\alpha_k \leftarrow \mathsf{unit}]^k)^i : \kappa \simeq \tau_1[\alpha_k \leftarrow \mathsf{unit}]^k \to \tau_2[\alpha_k \leftarrow \mathsf{unit}]^k : \kappa'$ But this would imply $\mathcal{V}[\tau_1[\alpha_k \leftarrow \mathsf{unit}]^k \to \tau_2[\alpha_k \leftarrow \mathsf{unit}]^k] = \mathcal{V}[\zeta\,(\tau_i[\alpha_k \leftarrow \mathsf{unit}]^k)^i]$, which is false (function definitions are present in the first set but not the second).
- Similarly for $v = \Lambda(\alpha : \mathsf{Typ}).\, u$, and for $v = d(\tau_j)^j(v_k)^k$ where d is a constructor of another type.
- There remains only the case $v = d(\tau_i')^i(v_k)^k$ with d a constructor of ζ.

□

Lemma 6.24 (Progress)**.** *Suppose $\vdash a : \tau$. Then, either $a \longrightarrow_\beta b$ for some term b, or a is a value.*

Proof. We proceed by induction on the base term, as in the proof of Theorem 4.2 for ML, strengthening the induction to accept environments of the form $(\alpha_i : \mathsf{Typ})^i$. Instead of inversion for ML, we use Lemma 6.23.

For the (new) case of Reuse-Present and Reuse-Absurd: if $a = *p$, a is not typeable in an environment composed only of type variables: the only possibility is that $(\alpha_i)^i \vdash \mathsf{True} : \mathsf{bool} \simeq \mathsf{False} : \mathsf{bool}$, which is impossible as $(\mathsf{True}, \mathsf{False})$ is not included in the interpretation of bool. □

6.3.5 Full term equality and reduction

We'd like to prove that full term equality is stable by reduction. We have two cases to consider for an equality between a and b: the first one is performing a non-expansive reduction ($\longrightarrow_\iota$) on one of the terms (*e.g.* a reduces to a'). Then, a' stays equal to the same terms. The second is performing an expansive reduction on one term. Then, we can perform some number of non-expansive reductions then an expansive reduction on the other term and recover equal terms: essentially, we are performing the same reduction on the two sides up to non-expansive reduction.

Lemma 6.25 (Stability by non-expansive reduction)**.** *Suppose* $\mathsf{TermsEqual}(\Gamma \vdash^p a : \tau_a \simeq b : \tau_b)$*, and $a \longrightarrow_\iota a'$. Then,* $\mathsf{TermsEqual}(\Gamma \vdash^p a' : \tau_a \simeq b : \tau_b)$*.*

Proof. The atoms of a' are obtained by reducing one subterm of the decomposition of a, and removing some atoms. The removed atoms must have their condition equivalent to False by subject reduction for $\longrightarrow_\iota$ (subject reduction tells us the returned results from a' are *reduced outputs* of those of a, as defined in Definition 6.3). □

Lemma 6.26 (Stability by expansive reduction)**.** *Suppose* $\mathsf{TermsEqual}((\alpha_i : \mathsf{Typ})^i \vdash^a \tau_a : b \simeq \tau_b :)$ *, and* $a \longrightarrow_\beta a'$ *where the reduction reduces a function application. Then, there exists* b'*,* b'' *such that* $b \longrightarrow_\beta^* b'$ *without performing an expansive reduction, then* $b' \longrightarrow_\beta b''$*, performing an expansive reduction, and* $\mathsf{TermsEqual}(\Gamma \vdash^p a' : \tau_a \simeq b'' : \tau_b)$*.*

Proof. Reduce b until reaching either a value or an expansive reduction. If we reach a value b', we must have $\mathsf{TermsEqual}(\Gamma \vdash^p a : \tau_a \simeq b' : \tau_b)$. The decomposition of b' has no expansive atoms, thus the condition for all expansive atoms in a must be equivalent to False, thus there is no expansive reduction from a (expansive reductions in evaluation position have a constraint equal to True, by induction on evaluation context).

Otherwise, we reach an expansive reduction. It must be a reduction with equivalent function and argument. Then, by substitution, the bodies with argument substituted are equal terms. □

Chapter 7

Staging with mML

In this chapter, we introduce another component of our encoding of ornaments, as an extension of eML called mML. This extension has two purposes. First, it allows separating some meta-abstractions and meta-applications than can be reduced in a separate phase from the abstractions and applications already present in user code. The language is designed so that all mML code can be reduced without touching the eML code. Building on this phase distinction, we can introduce richer abstractions facilities to mML: since we guarantee that they can be eliminated, we can add things that are not available in eML, such as dependent types, abstraction on equalities, and type-level functions.

7.1 Overview of the design

We want the extensions in the language mML to be able to reduce in a separate phase from the eML reduction. This imposes several constraints.

We write mML application with a # to distinguish it from ML application. Similarly, all mML abstraction are marked with #.

First, we must ensure that ML constructs (functions, pattern matching, etc) do not return meta abstractions, because then reduction of some ML code could make new mML redexes appear. We ensure this by introducing a superkind Met of Sch that classifies the types of meta abstractions.

Since term meta-abstraction can reduce at any moment, even when the argument is not fully evaluated, it could substitute expansive terms for variables. But this would break stability of non-expansivity by evaluation. Thus we limit meta term application to non-expansive terms.

In an environment with an incoherent equality (such as (True $\simeq$ False) : bool), any two eML types are equal. We prefer to avoid this for mML: since mML code will be evaluated under arbitrary contexts, we need, *e.g.*, that type abstractions are not confused with term abstractions under any context. This is solved by disallowing mML types in type-level pattern matching, and limiting case splitting (TEq-Split) to only operate on types of kinds Typ and Sch.

Finally, we have to decide whether mML abstractions and applications are expansive. To allow reducing mML redexes in equalities, we consider most abstractions and applications non-expansive, and we require the body of abstractions to be non-expansive terms. However, to allow transporting expansive

computations in $m\mathsf{ML}$, for example for patches, we introduce *thunks*, noted $[\pi.a]$: thunks can contain an expansive term, whose labels start with π. A thunk is applied, noted $(u)^p$, by choosing a label p for the result of the evaluation of the body of the thunk. For example consider,

$$((\lambda^{\#}(x : \mathsf{bool}).\, [\pi.\, f^{\pi\cdot 1}\ x])\ \#\ \mathsf{True})^p$$

This reduces to:

$$([\pi.\, f^{\pi\cdot 1}\ \mathsf{True}])^p$$

This reduces to the following term, emitting a label $*p \leftarrow *p \cdot 1$:

$$f^{p\cdot 1}\ \mathsf{True}$$

Thus, thunks allows us to transport expansive computations inside $m\mathsf{ML}$ abstractions, as long as the place where they are used is ready to unthunk them. This is similar to labeled ML abstractions, but we can eliminate the thunking and unthunking at $m\mathsf{ML}$ reduction time, and we allow the thunk/unthunk redexes to be reduced in the full term equality for $m\mathsf{ML}$.

7.2 Definition of $m\mathsf{ML}$

7.2.1 Syntax and typing

The syntax of $m\mathsf{ML}$ builds on the syntax of $e\mathsf{ML}$. One thing we need to add is type-level functions, taking types, terms, or equalities as arguments. This requires a richer language of kinds to denote these abstractions. In the same way, we need a richer language of types to represent the term-level meta-abstractions on types, terms, and equalities.

The syntax of kinds and types is given in Figure 7.1. Compared to the syntax of kinds and types of $e\mathsf{ML}$ (Figure 6.1), the kinds are enriched with a kind of meta types Met that will serve to separate terms having a meta type from terms having an $e\mathsf{ML}$ type, and kinds for the type-level dependent meta-abstractions on types, terms and equalities. Equalities stay unnamed in meta-abstractions. The syntax of types is similarly enriched with types for term-level meta-abstractions on types, terms and equalities. These types have kind Met.

We also add type-level meta abstractions and applications on types, terms and equalities. The meta abstractions and applications are marked with # to distinguish them from normal abstractions and applications. As discussed previously, we limit meta term application to non-expansive terms. When applying an equality, we explicitly mark the equality, using the syntax $(u \simeq u) : \tau$ (this is the syntax used to mark equalities in $e\mathsf{ML}$ typing environments). This makes typing easier. Finally, the syntax of typing environments is modified to allow type variables at any kind instead of only the kind Typ. The syntax of terms (Figure 7.2) is similarly enriched with meta-abstractions and applications. We add meta-abstraction to non-expansive terms, but not meta-application as its reduction can yield an expansive term.

We now describe the typing rules of $m\mathsf{ML}$. Datatypes stay the same (in particular, they cannot contain values of a meta-type). The rules for the well-formedness of environments are staightforward and given on Figure 7.3. In EnvVar, the type of the variable can now be of kind Met. In EnvTVar, we

κ ::=		Kinds
	$\mid$ Typ	Monotypes
	$\mid$ Sch	Type schemes
	$\mid$ Met	Meta types
	$\mid$ $\Pi(\alpha : \kappa)\ \kappa$	Type function
	$\mid$ $\Pi(x : \tau)\ \kappa$	Term function
τ, σ ::=		Types
	$\mid$ α	Type variable
	$\mid$ $\tau \to \tau$	Function type
	$\mid$ $\zeta\ \overline{\tau}$	Datatype
	$\mid$ $\forall(\alpha : \mathsf{Typ})\ \tau$	Universal quantification
	$\mid$ $\mathsf{match}\ a\ \mathsf{with}\ \overline{P \to \tau}$	Pattern matching
	$\mid$ $\lambda^{\#}(\alpha : \kappa).\ \tau$	Type abstraction (meta)
	$\mid$ $\lambda^{\#}(x : \tau).\ \tau$	Term abstraction (meta)
	$\mid$ $\tau \# \tau$	Type application (meta)
	$\mid$ $\tau \# u$	Term application (meta)
	$\mid$ $\Pi(\alpha : \kappa)\ \tau$	Type function (meta)
	$\mid$ $\Pi(x : \tau)\ \tau$	Term function (meta)
	$\mid$ $\Pi((u \simeq u) : \tau)\ \tau$	Equality function (meta)
	$\mid$ $\Pi((u)*p : \tau)\ \tau$	Reused result function (meta)
	$\mid$ $[\tau]$	Thunk (meta)
Γ ::=		Typing environments
	$\mid$ $\varnothing$	Empty
	$\mid$ $\Gamma, x : \tau$	Variable
	$\mid$ $\Gamma, \alpha : \kappa$	Type variable
	$\mid$ $\Gamma, (a \simeq a) : \tau$	Equality
	$\mid$ $\Gamma, (u)*p : \tau$	Precomputed result
	$\mid$ Γ, π	Path variable
Δ ::=		Computed applications
	$\mid$ $\varnothing$	Empty
	$\mid$ $\Delta, (u)*p : \tau$	Result

Figure 7.1: Kinds and types of mML

a, b	$::=$	Terms
	$\mid x$	Variable
	$\mid \mathsf{let}\ x = a\ \mathsf{in}\ a$	Let binding
	$\mid \mathsf{fix}^{\pi}\ x\ (x : \tau) : \tau\ .\ a$	Function by fixed point
	$\mid a^{p}\ a$	Application
	$\mid a\ \tau$	Type application
	$\mid \Lambda(\alpha : \mathsf{Typ}).\ u$	Type abstraction
	$\mid d(\tau)^{i}(a)^{i}$	Construction
	$\mid \mathsf{match}\ a\ \mathsf{with}\ (P \to a \mid ..\ P \to a)$	Pattern matching
	$\mid *p$	Reused result
	$\mid \lambda^{\#}(\alpha : \kappa).\ u$	Type abstraction (meta)
	$\mid \lambda^{\#}(x : \tau).\ u$	Term abstraction (meta)
	$\mid \lambda^{\#}(u \simeq u) : \tau.\ u$	Equality abstraction (meta)
	$\mid u \# \tau$	Type application (meta)
	$\mid u \# u$	Term application (meta)
	$\mid u \# (u \simeq u) : \tau$	Equality application (meta)
	$\mid [\pi.\ u]$	Thunk creation (meta)
	$\mid (u)^{p}$	Thunk evaluation (meta)
	$\mid \lambda^{\#}((u)*p : \tau).\ u$	Result abstraction (meta)
	$\mid u \# u$	Result application (meta)
P	$::=$	Patterns
	$\mid d(\tau)^{i}(x)^{i}$	

$$
\begin{aligned}
u\ ::=\ & x \mid d(\tau)^{i}(u)^{i} \mid \mathsf{fix}^{\pi}\ x\ (x : \tau) : \tau\ .\ a \mid u\ \tau \mid \Lambda(\alpha : \kappa).\ u \mid \mathsf{let}\ x = u\ \mathsf{in}\ u \\
\mid\ & \mathsf{match}\ u\ \mathsf{with}\ (P \to u \mid ..\ P \to u) \mid *p \mid \lambda^{\#}(\alpha : \kappa).\ u \mid \lambda^{\#}(x : \tau).\ u \\
\mid\ & \lambda^{\#}(u \simeq u) : \tau.\ u \mid u \# \tau \mid u \# u \mid u \# (a \simeq a) : \tau \mid [\pi.\ a] \mid \lambda^{\#}((u)*p : \tau).\ u \\
\mid\ & u \#^{*} u
\end{aligned}
$$

Figure 7.2: Syntax of mML

$$\textsc{EnvEmpty}\quad \vdash \varnothing$$

$$\textsc{EnvPathVar}\quad \frac{\vdash \Gamma \qquad \pi \# \Gamma}{\vdash \Gamma, \pi}$$

$$\textsc{EnvVar}\quad \frac{\vdash \Gamma \qquad \Gamma \vdash \tau : \mathsf{Met} \qquad x \# \Gamma}{\vdash \Gamma, x : \tau}$$

$$\textsc{EnvTVar}\quad \frac{\vdash \Gamma \qquad \Gamma \vdash \kappa \text{ wf} \qquad \alpha \# \Gamma}{\vdash \Gamma, \alpha : \kappa}$$

$$\textsc{EnvEq}\quad \frac{\vdash \Gamma \qquad \Gamma \vdash u_1 : \tau \qquad \Gamma \vdash u_2 : \tau \qquad \Gamma \vdash \tau : \mathsf{Sch}}{\vdash \Gamma, (u_1 \simeq u_2) : \tau}$$

$$\textsc{EnvPtr}\quad \frac{\vdash \Gamma \qquad p \perp \Gamma \qquad (p = \pi \cdot q \implies \pi \in \Gamma) \qquad \Gamma \vdash u : \mathsf{bool} \qquad \Gamma \vdash \tau : \mathsf{Sch}}{\vdash \Gamma, (u)*p : \tau}$$

$$\textsc{Wf-Base}\quad \frac{\vdash \Gamma \qquad \kappa \in \{\mathsf{Typ}, \mathsf{Sch}, \mathsf{Met}\}}{\Gamma \vdash \kappa \text{ wf}}$$

$$\textsc{Wf-TyFun}\quad \frac{\alpha \# \Gamma \qquad \Gamma \vdash \kappa \text{ wf} \qquad \Gamma, \alpha : \kappa \vdash \kappa' \text{ wf}}{\Gamma \vdash \Pi(\alpha : \kappa)\, \kappa' \text{ wf}}$$

$$\textsc{Wf-TermFun}\quad \frac{x \# \Gamma \qquad \Gamma \vdash \tau : \mathsf{Met} \qquad \Gamma, x : \tau \vdash \kappa \text{ wf}}{\Gamma \vdash \Pi(x : \tau)\, \kappa \text{ wf}}$$

Figure 7.3: Well-formedness for environments and kind

allow quantification on type variables of any kind. We introduce a new judgment for well-formedness of kinds: we write $\Gamma \vdash \kappa$ wf if κ is well-formed in the environment Γ.

The well-formedness rules for types are given in Figure 7.4. We introduce new rules for the new constructions. The types of term-level abstractions are classified with kind Met. Type-level meta-abstractions are dependently typed using the new kinds. Type-level pattern matching and universal types are still limited to types of kind Typ or Sch: this enforces that the types of term-level meta-abstractions cannot appear inside a ML type. This is important for correctly reducing from mML to eML. We add a subtyping rule K-SubSch from Sch to Met: thus, the base kinds form a hierarchy, with Met containing Sch containaing Typ.

The typing rules are similar to the typing rules for eML, except that we add rules for the new constructs, similar to those for types. In rule Let, we check that the type of the term in the binding and of the returned type is of kind Sch and not Met. This was already done for Match.

7.2.2 The meta reduction

In mML, two reductions coexist: the normal eML reduction $\longrightarrow_\beta$ and a reduction $\longrightarrow_\#$ that only reduces meta applications. The reduction $\longrightarrow_\#$ is non-deterministic and can occur under any context, including applications. Its definition is given in Figure 7.12. As in $\longrightarrow_\beta$, meta-reduction emits results that can

$$\textsc{K-Var}\quad \frac{\vdash \Gamma \qquad \alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa}$$

$$\textsc{K-Datatype}\quad \frac{\vdash \Gamma \qquad \vdash \zeta : (\mathsf{Typ})^i \Rightarrow \mathsf{Typ} \qquad (\Gamma \vdash \tau_i : \mathsf{Typ})^i}{\Gamma \vdash \zeta\,(\tau_i)^i : \mathsf{Typ}}$$

$$\textsc{K-Arr}\quad \frac{\Gamma \vdash \tau_1 : \mathsf{Typ} \qquad \Gamma \vdash \tau_2 : \mathsf{Typ}}{\Gamma \vdash \tau_1 \to \tau_2 : \mathsf{Typ}}$$

$$\textsc{K-SubTyp}\quad \frac{\Gamma \vdash \tau : \mathsf{Typ}}{\Gamma \vdash \tau : \mathsf{Sch}}$$

$$\textsc{K-SubSch}\quad \frac{\Gamma \vdash \tau : \mathsf{Sch}}{\Gamma \vdash \tau : \mathsf{Met}}$$

$$\textsc{K-All}\quad \frac{\Gamma, \alpha : \kappa \vdash \tau : \mathsf{Sch}}{\Gamma \vdash \forall(\alpha : \mathsf{Typ})\,\tau : \mathsf{Sch}}$$

$$\textsc{K-Match}\quad \frac{\begin{array}{c}\kappa \in \{\mathsf{Typ}, \mathsf{Sch}\} \\ \vdash (d_i)^i : \zeta \text{ complete} \qquad (d_i : \forall(\alpha_k : \mathsf{Typ})^k\,(\tau_{ij})^j \to \zeta\,(\alpha_k)^k)^i \qquad \Gamma \vdash u : \zeta\,(\tau_k)^k \\ (\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, (d_i(\tau_k)^k(x_{ij})^j \simeq u) : \zeta\,(\tau_k)^k \vdash \sigma_i : \kappa)^i\end{array}}{\Gamma \vdash \mathsf{match}\ u\ \mathsf{with}\ (d_i(\tau_k)^k(x_{ij})^j \to \sigma_i)^i : \kappa}$$

$$\textsc{K-TyAbs}\quad \frac{\Gamma, \alpha : \kappa_0 \vdash \tau : \kappa}{\Gamma \vdash \lambda^{\#}(\alpha : \kappa_0).\,\tau : \Pi(\alpha : \kappa_0)\,\kappa}$$

$$\textsc{K-TyApp}\quad \frac{\Gamma \vdash \tau : \Pi(\alpha : \kappa_0)\,\kappa \qquad \Gamma \vdash \tau_0 : \kappa_0}{\Gamma \vdash \tau \,\#\, \tau_0 : \kappa[\alpha \leftarrow \tau_0]}$$

$$\textsc{K-TermAbs}\quad \frac{\Gamma, x : \tau_0 \vdash \tau : \kappa}{\Gamma \vdash \lambda^{\#}(x : \tau_0).\,\tau : \Pi(x : \tau_0)\,\kappa}$$

$$\textsc{K-TermApp}\quad \frac{\Gamma \vdash \tau : \Pi(x : \tau_0)\,\kappa \qquad \Gamma \vdash u : \tau_0}{\Gamma \vdash \tau \,\#\, u : \kappa[x \leftarrow u]}$$

$$\textsc{K-TyMetaArr}\quad \frac{\Gamma, \alpha : \kappa \vdash \tau : \mathsf{Met}}{\Gamma \vdash \Pi(\alpha : \kappa)\,\tau : \mathsf{Met}}$$

$$\textsc{K-TermMetaArr}\quad \frac{\Gamma, x : \tau_0 \vdash \tau : \mathsf{Met}}{\Gamma \vdash \Pi(x : \tau_0)\,\tau : \mathsf{Met}}$$

$$\textsc{K-EqMetaArr}\quad \frac{\Gamma, (u_1 \simeq u_2) : \sigma \vdash \tau : \mathsf{Met}}{\Gamma \vdash \Pi(u_1 \simeq u_2) : \sigma\,\tau : \mathsf{Met}}$$

$$\textsc{K-PtrMetaArr}\quad \frac{\begin{array}{c}p \perp \Gamma \\ \Gamma \vdash u : \mathsf{bool} \qquad \Gamma, (u \simeq \mathsf{True}) : \mathsf{bool} \vdash \tau : \mathsf{Sch} \qquad \Gamma, (u){*}p : \tau \vdash \sigma : \mathsf{Met}\end{array}}{\Gamma \vdash \Pi((u){*}p : \tau)\,\sigma : \mathsf{Met}}$$

$$\textsc{K-Thunk}\quad \frac{\Gamma \vdash \tau : \mathsf{Sch}}{\Gamma \vdash [\tau] : \mathsf{Met}}$$

Figure 7.4: Kinding rules for mML

$$\textsc{Var}\quad \frac{\vdash \Gamma \qquad x : \sigma \in \Gamma}{\Gamma \vdash^p x : \sigma \Rightarrow \varnothing}$$

$$\textsc{TAbs}\quad \frac{\Gamma, \alpha : \mathsf{Typ} \vdash^p u : \sigma \Rightarrow \varnothing}{\Gamma \vdash^p \Lambda(\alpha : \mathsf{Typ}).\, u : \forall(\alpha : \mathsf{Typ})\, \sigma \Rightarrow \varnothing}$$

$$\textsc{TApp}\quad \frac{\Gamma \vdash \tau : \mathsf{Typ} \qquad \Gamma \vdash^p a : \forall(\alpha : \mathsf{Typ})\, \sigma \Rightarrow \Delta}{\Gamma \vdash^p a\ \tau : \sigma[\alpha \leftarrow \tau] \Rightarrow \Delta}$$

$$\textsc{Fix}\quad \frac{\Gamma, \pi, x : \tau_1 \to \tau_2, y : \tau_1 \vdash^\pi a : \tau_2 \Rightarrow \Delta}{\Gamma \vdash^p \mathsf{fix}^\pi\ x\ (y : \tau_1) : \tau_2\ .\ a : \tau_1 \to \tau_2 \Rightarrow \varnothing}$$

$$\textsc{App}\quad \frac{\Gamma \vdash^p a : \tau_1 \to \tau_2 \Rightarrow \Delta_1 \qquad \Gamma, \Delta_1 \vdash^p b : \tau_1 \Rightarrow \Delta_2 \qquad p \leqslant q \qquad q \perp \Gamma, \Delta_1, \Delta_2}{\Gamma \vdash^p a^q\ b : \tau_2 \Rightarrow \Delta_1, \Delta_2, (\mathsf{True})*q : \tau_2}$$

$$\textsc{Let}\quad \frac{\begin{array}{c}\Gamma \vdash^p a : \tau' \Rightarrow \Delta_1 \\ \Gamma, \Delta_a, x : \tau', (x \simeq \mathsf{reuse}(a)) : \tau' \vdash^p b : \tau \Rightarrow \Delta_2 \qquad \Gamma \vdash \tau : \mathsf{Sch} \qquad \Gamma \vdash \tau' : \mathsf{Sch}\end{array}}{\Gamma \vdash^p \mathsf{let}\ x = a\ \mathsf{in}\ b : \tau \Rightarrow \Delta_1, \Delta_2[x \leftarrow \mathsf{reuse}(a)]}$$

$$\textsc{Con}\quad \frac{\begin{array}{c}\vdash \Gamma \qquad \vdash d : \forall(\alpha_j : \mathsf{Typ})^j\ (\tau_i)^i \to \zeta\ (\alpha_j)^j \\ (\Gamma \vdash \tau_j : \mathsf{Typ})^j \qquad (\Gamma, (\Delta_k)^{k>i} \vdash^p a_i : \tau_i[\alpha_j \leftarrow \tau_j]^j \Rightarrow \Delta_i)^i\end{array}}{\Gamma \vdash^p d(\tau_j)^j(a_i)^i : \zeta\ (\tau_j)^j \Rightarrow (\Delta_i)^i}$$

$$\textsc{Match}\quad \frac{\begin{array}{c}\vdash (d_i)^i : \zeta\ \mathsf{complete} \\ \Gamma \vdash \tau : \mathsf{Sch} \qquad (d_i : \forall(\alpha_k : \mathsf{Typ})^k\ (\tau_{ij})^j \to \zeta\ (\alpha_k)^k)^i \qquad \Gamma \vdash^p a : \zeta\ (\tau_k)^k \Rightarrow \Delta \\ (\Gamma, \Delta, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, (d_i(\tau_k)^k(x_{ij})^j \simeq \mathsf{reuse}(a)) : \zeta\ (\tau_k)^k \vdash^p b_i : \tau \Rightarrow \Delta_i)^i\end{array}}{\begin{array}{c}\Gamma \vdash^p \mathsf{match}\ a\ \mathsf{with}\ (d_i(\tau_k)^k(x_{ij})^j \to b_i)^i : \tau \\ \Rightarrow \Delta, (\mathsf{match}\ \mathsf{reuse}(a)\ \mathsf{with}\ d_i(\tau_k)^k(x_{ij})^j \to \Delta_i)^i\end{array}}$$

$$\textsc{Coerce}\quad \frac{\Gamma \vdash^p a : \tau' \Rightarrow \Delta \qquad \Gamma \vdash \tau' : \kappa' \simeq \tau : \kappa}{\Gamma \vdash^p a : \tau \Rightarrow \Delta}$$

$$\textsc{Reuse-Present}\quad \frac{(u)*q : \tau \in \Gamma \qquad \Gamma \vdash u : \mathsf{bool} \simeq \mathsf{True} : \mathsf{bool}}{\Gamma \vdash^p *q : \tau \Rightarrow \varnothing}$$

$$\textsc{Reuse-Absurd}\quad \frac{q \perp \Gamma \qquad \Gamma \vdash \mathsf{False} : \mathsf{bool} \simeq \mathsf{True} : \mathsf{bool} \qquad \Gamma \vdash \tau : \mathsf{Typ}}{\Gamma \vdash^p *q : \tau \Rightarrow \varnothing}$$

Figure 7.5: Typing rules for mML (eML rules)

$$\textsc{TyMetaAbs}\quad \frac{\Gamma, \alpha : \kappa \vdash^p u : \tau \Rightarrow \varnothing}{\Gamma \vdash^p \lambda^{\#}(\alpha : \kappa).\, u : \Pi(\alpha : \kappa)\, \tau \Rightarrow \varnothing}$$

$$\textsc{TyMetaApp}\quad \frac{\Gamma \vdash^p u : \Pi(\alpha : \kappa_0)\, \tau \Rightarrow \varnothing \qquad \Gamma \vdash \tau_0 : \kappa_0}{\Gamma \vdash^p u \# \tau_0 : \tau[\alpha \leftarrow \tau_0] \Rightarrow \varnothing}$$

$$\textsc{TermMetaAbs}\quad \frac{\Gamma, x : \tau_0 \vdash^p u : \tau \Rightarrow \varnothing}{\Gamma \vdash^p \lambda^{\#}(x : \tau_0).\, u : \Pi(x : \tau_0)\, \tau \Rightarrow \varnothing}$$

$$\textsc{TermMetaApp}\quad \frac{\Gamma \vdash^p u : \Pi(x : \tau_0)\, \tau \Rightarrow \varnothing \qquad \Gamma \vdash^p u_0 : \tau_0 \Rightarrow \varnothing}{\Gamma \vdash^p u \# u_0 : \tau[x \leftarrow u_0] \Rightarrow \varnothing}$$

$$\textsc{EqMetaAbs}\quad \frac{\Gamma, (u_1 \simeq u_2) : \sigma \vdash^p u : \tau \Rightarrow \varnothing}{\Gamma \vdash^p \lambda^{\#}((u_1 \simeq u_2) : \sigma).\, u : \Pi((u_1 \simeq u_2) : \sigma)\, \tau \Rightarrow \varnothing}$$

$$\textsc{EqMetaApp}\quad \frac{\Gamma \vdash^p u : \Pi(u_1 \simeq u_2) : \sigma\, \tau \Rightarrow \varnothing \qquad \Gamma \vdash u_1 : \sigma \simeq u_2 : \sigma}{\Gamma \vdash^p u \# (u_1 \simeq u_2) : \sigma : \tau \Rightarrow \varnothing}$$

$$\textsc{PtrMetaAbs}\quad \frac{\Gamma, (u_0 \simeq \mathsf{True}) : \mathsf{bool} \vdash \tau : \mathsf{Sch} \qquad \Gamma, (u_0)*q : \tau \vdash^p u : \sigma \Rightarrow \varnothing}{\Gamma \vdash^p \lambda^{\#}((u_0)*q : \tau).\, u : \Pi((u_0)*q : \tau)\, \sigma \Rightarrow \varnothing}$$

$$\textsc{PtrMetaApp}\quad \frac{\Gamma \vdash^p u : \Pi((u_0)*p : \tau)\, \sigma \Rightarrow \varnothing \qquad \Gamma, (u_0 \simeq \mathsf{True}) : \mathsf{bool} \vdash^p u' : \tau \Rightarrow \varnothing}{\Gamma \vdash^p u \#^* u' : \sigma[*p \leftarrow u'] \Rightarrow \varnothing}$$

$$\textsc{Thunk}\quad \frac{\Gamma, \pi \vdash^\pi a : \tau \Rightarrow \Delta}{\Gamma \vdash^p [\pi.\, a] : [\tau] \Rightarrow \varnothing} \qquad \textsc{Unthunk}\quad \frac{\Gamma \vdash^p u : [\tau] \Rightarrow \varnothing \qquad p \leqslant q \qquad q \perp \Gamma, \Delta}{\Gamma \vdash^p (u)^q : \tau \Rightarrow \Delta, (\mathsf{True})*q : \tau}$$

Figure 7.6: Typing rules for $m\mathsf{ML}$ (new rules)

$$\textsc{KEq-Trans}\quad \frac{\Gamma \vdash \kappa_1 \simeq \kappa_2 \qquad \Gamma \vdash \kappa_2 \simeq \kappa_3}{\Gamma \vdash \kappa_1 \simeq \kappa_3} \qquad \textsc{KEq-Base}\quad \frac{\vdash \Gamma \qquad \kappa \in \{\mathsf{Typ}, \mathsf{Sch}, \mathsf{Met}\}}{\Gamma \vdash \kappa \simeq \kappa}$$

$$\textsc{KEq-Type-Fun}\quad \frac{\Gamma \vdash \kappa_1 \simeq \kappa_2 \qquad \Gamma, \alpha : \kappa_1 \vdash \kappa_1' \simeq \kappa_2'}{\Gamma \vdash \Pi(\alpha : \kappa_1)\, \kappa_1' \simeq \Pi(\alpha : \kappa_2)\, \kappa_2'}$$

$$\textsc{KEq-Term-Fun}\quad \frac{\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2 \qquad \Gamma, x : \tau_1 \vdash \kappa_1' \simeq \kappa_2'}{\Gamma \vdash \Pi(x : \tau_1)\, \kappa_1' \simeq \Pi(x : \tau_2)\, \kappa_2'}$$

Figure 7.7: Kind equality for $m\mathsf{ML}$

be reused. This only happens when evaluating thunks. The label translation rules are extended for the new abstractions and applications (see Figure 7.13): thunk abtraction hides the labels emitted inside, all applications are transparent to labels. We do not provide a label translation for the other meta abstractions as they only expect non-expansive terms.

7.2.3 Equality

The equality of $m\mathsf{ML}$ is similar to the equality of $e\mathsf{ML}$. There are three major changes: we include congruence rules corresponding to the new constructs on types and terms; we include reduction rules corresponding to the head reduction of $\longrightarrow_{\#}$ (the non-head reduction are included because non-expansive term equality is a congruence). We limit case-splitting to proving equalities between terms whose type is of kind Sch, and between types whose kind is Sch: this ensures that the $m\mathsf{ML}$ part of type equality always holds, even in contexts that are absurd, and thus unreachable for $e\mathsf{ML}$ reduction: we need to guarantee progress for the full $m\mathsf{ML}$ reduction.

The new and changed rules are given on Figure 7.8 for the congruence rules for types, Figure 7.9 for the congruence rules for terms. As in $e\mathsf{ML}$, the congruence rules follow the structure of the typing rules. The reduction rules, the new split rule and the subkinding rules are given in Figure 7.10 and Figure 7.11.

Since types can appear in kinds, we also have a kind equality judgment, that only includes congruence rules (defined on Figure 7.7).

We also extend term equality to take thunk application into account. There is a new atom of the form $(u)^p$. We can either chose to keep these atoms as-is, in which case they are equal to atoms $(u')^p$ when u and u' are equal, or to expand them: we must provide a such that u is equal to $[\pi.\, a]$. Then, the atom is replaced by the decomposition of a before evaluating the equality. This makes such atoms transparent for term equality: we can reduce them as needed, thus meta-reduction preserves term equality.

Then, the definition of equality is in two steps: first we allow *expanding* as much as desired on each side, then we compare the terms for equality.

Definition 7.1 (Expansion)**.** *Consider* $\Gamma \vdash^p a : \tau \Rightarrow \Delta$. *Let* $S = \mathsf{labels}(a)$ *and suppose* $\mathsf{atoms}(a) = (*p_i \leftarrow (T_i, a_i))^{p_i \in \mathsf{labels}(a)}$, *with for all i,*

TEq-TyAbs

$$\frac{\Gamma \vdash \kappa_1' \simeq \kappa_2' \qquad \Gamma, \alpha : \kappa_1' \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2}{\Gamma \vdash \lambda^{\#}(\alpha : \kappa_1').\, \tau_1 : \Pi(\alpha : \kappa_1')\, \kappa_1 \simeq \lambda^{\#}(\alpha : \kappa_2').\, \tau_2 : \Pi(\alpha : \kappa_2')\, \kappa_2}$$

TEq-TermAbs

$$\frac{\Gamma \vdash \sigma_1 : \kappa_1' \simeq \sigma_2 : \kappa_2' \qquad \Gamma, x : \sigma_1 \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2}{\Gamma \vdash \lambda^{\#}(x : \sigma_1).\, \tau_1 : \Pi(x : \sigma_1)\, \kappa_1 \simeq \lambda^{\#}(x : \sigma_2).\, \tau_2 : \Pi(x : \sigma_2)\, \kappa_2}$$

TEq-TyApp

$$\frac{\Gamma \vdash \sigma_1 : \Pi(\alpha : \kappa_1')\, \kappa_1 \simeq \sigma_2 : \Pi(\alpha : \kappa_2')\, \kappa_2 \qquad \Gamma \vdash \tau_1 : \kappa_1' \simeq \tau_2 : \kappa_2'}{\Gamma \vdash \sigma_1 \# \tau_1 : \kappa_1[\alpha \leftarrow \tau_1]\sigma_2 \# \tau_2 \kappa_2[\alpha \leftarrow \tau_2]}$$

TEq-TermApp

$$\frac{\Gamma \vdash \sigma_1 : \Pi(x : \tau_1)\, \kappa_1 \simeq \sigma_2 : \Pi(x : \tau_2)\, \kappa_2 \qquad \Gamma \vdash u_1 : \tau_1 \simeq u_2 : \tau_2}{\Gamma \vdash \sigma_1 \# u_1 : \kappa_1[x \leftarrow u_1]\sigma_2 \# u_2 \kappa_2[x \leftarrow u_2]}$$

TEq-TyMetaArr

$$\frac{\Gamma \vdash \kappa_1 \simeq \kappa_2 \qquad \Gamma, \alpha : \kappa_1 \vdash \tau_1 : \mathsf{Met} \simeq \tau_2 : \mathsf{Met}}{\Gamma \vdash \Pi(\alpha : \kappa_1)\, \tau_1 : \mathsf{Met} \simeq \Pi(\alpha : \kappa_2)\, \tau_2 : \mathsf{Met}}$$

TEq-TermMetaArr

$$\frac{\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2 \qquad \Gamma, x : \tau_1 \vdash \sigma_1 : \mathsf{Met} \simeq \sigma_2 : \mathsf{Met}}{\Gamma \vdash \Pi(x : \tau_1)\, \sigma_1 : \mathsf{Met} \simeq \Pi(x : \tau_2)\, \sigma_2 : \mathsf{Met}}$$

TEq-EqMetaArr

$$\frac{\begin{array}{c}\Gamma \vdash u_{11} : \sigma_1 \simeq u_{21} : \sigma_2 \\ \Gamma \vdash u_{12} : \sigma_1 \simeq u_{22} : \sigma_2 \qquad \Gamma, (u_{11} \simeq u_{12}) : \sigma_1 \vdash \tau_1 : \mathsf{Met} \simeq \tau_2 : \mathsf{Met}\end{array}}{\Gamma \vdash \Pi(u_{11} \simeq u_{12}) : \sigma_1\, \tau_1 : \mathsf{Met} \simeq \Pi(u_{21} \simeq u_{22}) : \sigma_2\, \tau_2 : \mathsf{Met}}$$

TEq-PtrMetaArr

$$\frac{\begin{array}{c}p \perp \Gamma \\ \Gamma \vdash u_1 : \mathsf{bool} \simeq u_2 : \mathsf{bool} \qquad \Gamma, (u_1 \simeq \mathsf{True}) : \mathsf{bool} \vdash \tau_1 : \mathsf{Sch} \simeq \tau_2 : \mathsf{Sch} \\ \Gamma, (u_1)*p : \tau_1 \vdash \sigma_1 : \mathsf{Met} \simeq \sigma_2 : \mathsf{Met}\end{array}}{\Gamma \vdash \Pi((u_1)*p : \tau_1)\, \sigma_1 : \mathsf{Met} \simeq \Pi((u_2)*p : \tau_2)\, \sigma_2 : \mathsf{Met}}$$

TEq-Thunk

$$\frac{\Gamma \vdash \tau_1 : \mathsf{Sch} \simeq \tau_2 : \mathsf{Sch}}{\Gamma \vdash [\tau_1] : \mathsf{Met} \simeq [\tau_2] : \mathsf{Met}}$$

Figure 7.8: Type equality for mML: new congruence rules

Eq-TyMetaAbs

$$\frac{\Gamma \vdash \kappa_1 \simeq \kappa_2 \qquad \Gamma, \alpha : \kappa_1 \vdash u_1 : \tau \simeq u_2 : \tau_2}{\Gamma \vdash \lambda^{\#}(\alpha : \kappa_1).\, u_1 : \Pi(\alpha : \kappa_1)\, \tau_1 \simeq \lambda^{\#}(\alpha : \kappa_2).\, u_2 : \Pi(\alpha : \kappa_2)\, \tau_2}$$

Eq-TermMetaAbs

$$\frac{\Gamma \vdash \tau_1 : \mathsf{Met} \simeq \tau_2 : \mathsf{Met} \qquad \Gamma, x : \tau_1 \vdash a_1 : \sigma_1 \simeq a_2 : \sigma_2}{\Gamma \vdash \lambda^{\#}(x : \tau_1).\, a_1 : \Pi(x : \tau_1)\, \sigma_1 \simeq \lambda^{\#}(x : \tau_2).\, a_2 : \Pi(x : \tau_2)\, \sigma_2}$$

Eq-EqMetaAbs

$$\frac{\begin{array}{c}\Gamma \vdash u_{11} : \sigma_1 \simeq u_{21} : \sigma_2 \\ \Gamma \vdash u_{12} : \sigma_2 \simeq u_{22} : \sigma_2 \qquad \Gamma, (u_{11} \simeq u_{12}) : \sigma_1 \vdash u_1 : \tau_1 \simeq u_2 : \tau_2\end{array}}{\begin{array}{c}\Gamma \vdash \lambda^{\#}((u_{11} \simeq u_{12}) : \sigma_1).\, u_1 : \Pi((u_{11} \simeq u_{12}) : \sigma_1)\, \tau_1 \\ \simeq \lambda^{\#}((u_{21} \simeq u_{22}) : \sigma_2).\, u_2 : \Pi((u_{21} \simeq u_{22}) : \sigma_2)\, \tau_2\end{array}}$$

Eq-PtrMetaAbs

$$\frac{\begin{array}{c}p \perp \Gamma \qquad \Gamma \vdash u_1' : \mathsf{bool} \simeq u_2' : \mathsf{bool} \\ \Gamma, (u_1' \simeq \mathsf{True}) : \mathsf{bool} \vdash \tau_1 : \mathsf{Sch} \simeq \tau_2 : \mathsf{Sch} \qquad \Gamma, (u_1')*p : \tau_1 \vdash u_1 : \sigma_1 \simeq u_2 : \sigma_2\end{array}}{\Gamma \vdash \lambda^{\#}((u_1')*p : \tau_1).\, u_1 : \Pi((u_1')*p : \tau_1)\, \sigma_1}$$

Eq-Thunk

$$\frac{\mathsf{TermsEqual}(\Gamma \vdash^{\pi} a_1 : \tau_1 \simeq a_2 : \tau_2)}{\Gamma \vdash [\pi.\, a_1] : [\tau_1] \simeq [\pi.\, a_2] : [\tau_2]}$$

Eq-TyMetaApp

$$\frac{\Gamma \vdash u_1 : \Pi(\alpha : \kappa_1)\, \sigma_1 \simeq u_2 : \Pi(\alpha : \kappa_2)\, \sigma_2 \qquad \Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2}{\Gamma \vdash u_1 \# \tau_1 : \sigma_1[\alpha \leftarrow \tau_1] \simeq u_2 \# \tau_2 : \sigma_2[\alpha \leftarrow \tau_2]}$$

Eq-TermMetaApp

$$\frac{\Gamma \vdash u_1 : \Pi(x : \tau_1')\, \tau_1 \simeq u_2 : \Pi(x : \tau_2')\, \tau_2 \qquad \Gamma \vdash u_1' : \tau_1' \simeq u_2' : \tau_2'}{\Gamma \vdash u_1 \# u_1' : \tau_1[x \leftarrow u_1'] \simeq u_2 \# u_2' : \tau_2[x \leftarrow u_2']}$$

Eq-EqMetaApp

$$\frac{\begin{array}{c}\Gamma \vdash u_1 : \Pi(u_{11} \simeq u_{12}) : \sigma_1\, \tau_1 \simeq u_2 : \Pi(u_{21} \simeq u_{22}) : \sigma_2\, \tau_2 \\ \Gamma \vdash u_{11} : \sigma_1 \simeq u_{12} : \sigma_1 \qquad \Gamma \vdash u_{21} : \sigma_2 \simeq u_{22} : \sigma_2\end{array}}{\Gamma \vdash u_1 \# (u_{11} \simeq u_{12}) : \sigma_1 : \tau_1 \simeq u_2 \# (u_{21} \simeq u_{22}) : \sigma_2 : \tau_2}$$

Eq-PtrMetaApp

$$\frac{\begin{array}{c}\Gamma \vdash u_1 : \Pi((u_1'')*p : \tau_1)\, \sigma_1 \simeq u_2 : \Pi((u_2'')*p : \tau_1)\, \sigma_2 \\ \Gamma, (u_1'' \simeq \mathsf{True}) : \mathsf{bool} \vdash u_1' : \tau_1 \simeq u_2' : \tau_2\end{array}}{\Gamma \vdash u_1 \#^* u_1' : \sigma_1[*p \leftarrow u_1'] \simeq u_2 \#^* u_2' : \sigma_2[*p \leftarrow u_2']}$$

Figure 7.9: Term equality for mML: new congruence rules

TEq-SubSch

$$\frac{\Gamma \vdash \tau : \mathsf{Sch} \simeq \tau' : \kappa}{\Gamma \vdash \tau : \mathsf{Met} \simeq \tau' : \kappa}$$

TEq-Split

$$\frac{\begin{array}{c}(\vdash d_i : \forall(\alpha_k : \mathsf{Typ})^k\ (\tau_{ij})^j \to \zeta\,(\alpha_k)^k)^i \\ \Gamma \vdash u : \zeta\,(\tau_k)^k \qquad \Gamma \vdash \sigma_1 : \kappa_1 \qquad \Gamma \vdash \sigma_2 : \kappa_2 \qquad \kappa_1, \kappa_2 \in \{\mathsf{Typ}, \mathsf{Sch}\} \\ (\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, (d_i(\tau_k)^k(x_{ij})^j \simeq u) : \zeta\,(\tau_k)^k \vdash \sigma_1 : \kappa_1 \simeq \sigma_2 : \kappa_2)^i\end{array}}{\Gamma \vdash \sigma_1 : \kappa_1 \simeq \sigma_2 : \kappa_2}$$

TEq-Reduce-MetaTApp

$$\frac{\Gamma, \alpha : \kappa_0 \vdash \sigma : \kappa \qquad \Gamma \vdash \tau : \kappa_0}{\Gamma \vdash (\lambda^{\#}(\alpha : \kappa_0).\,\sigma) \# \tau : \kappa[\alpha \leftarrow \tau] \simeq \sigma[\alpha \leftarrow \tau] : \kappa[\alpha \leftarrow \tau]}$$

TEq-Reduce-MetaApp

$$\frac{\Gamma, x : \tau \vdash \sigma : \kappa \qquad \Gamma \vdash u : \tau}{\Gamma \vdash (\lambda^{\#}(x : \tau).\,\sigma) \# u : \kappa[x \leftarrow u] \simeq \sigma[x \leftarrow u] : \kappa[x \leftarrow u]}$$

TEq-Reduce-MetaEqApp

$$\frac{\Gamma, (u_1 \simeq u_2) : \tau \vdash \sigma : \kappa \qquad \Gamma \vdash u_1 : \tau \simeq u_2 : \tau}{\Gamma \vdash (\lambda^{\#}((u_1 \simeq u_2) : \tau).\,\sigma) \# ((u_1 \simeq u_2) : \tau) : \kappa \simeq \sigma : \kappa}$$

Figure 7.10: Type equality for mML: subkinding, reduction and split

Eq-Reduce-MetaTApp

$$\frac{\Gamma, \alpha : \kappa \vdash u : \sigma \qquad \Gamma \vdash \tau : \kappa}{\Gamma \vdash (\lambda^{\#}(\alpha : \kappa).\,u) \# \tau : \sigma[\alpha \leftarrow \tau] \simeq u[\alpha \leftarrow \tau] : \sigma[\alpha \leftarrow \tau]}$$

Eq-Reduce-MetaApp

$$\frac{\Gamma, x : \tau_0 \vdash u : \tau \qquad \Gamma \vdash u_0 : \tau_0}{\Gamma \vdash (\lambda^{\#}(x : \tau).\,u) \# u_0 : \tau[x \leftarrow u_0] \simeq u[x \leftarrow u_0] : \tau[x \leftarrow u_0]}$$

Eq-Reduce-MetaEqApp

$$\frac{\Gamma, (u_1 \simeq u_2) : \tau \vdash u : \sigma \qquad \Gamma \vdash u_1 : \tau \simeq u_2 : \tau}{\Gamma \vdash (\lambda^{\#}((u_1 \simeq u_2) : \tau).\,u) \# ((u_1 \simeq u_2) : \tau) : \sigma \simeq u : \sigma}$$

Eq-Reduce-MetaPtrApp

$$\frac{p \perp \Gamma \qquad \Gamma, (u_0)*p : \tau \vdash u : \sigma \qquad \Gamma, (u_0 \simeq \mathsf{True}) : \mathsf{bool} \vdash u' : \tau}{\Gamma \vdash (\lambda^{\#}((u_0)*p : \tau).\,u) \#^{*} u' : \sigma[*p \leftarrow u'] \simeq u[*p \leftarrow u'] : \sigma[*p \leftarrow u']}$$

Figure 7.11: Term equality for mML: new reduction rules

$$
\begin{array}{rcl}
(\lambda^{\#}(\alpha : \kappa).\, \sigma) \# \tau & \xrightarrow{0}_{\#} & \sigma[\alpha \leftarrow \tau] \\
(\lambda^{\#}(x : \tau).\, \sigma) \# u & \xrightarrow{0}_{\#} & \sigma[x \leftarrow u] \\
(\lambda^{\#}(u_1 \simeq u_2) : \tau.\, \sigma) \# (u_1 \simeq u_2) : \tau' & \xrightarrow{0}_{\#} & \sigma
\end{array}
$$

$$
\begin{array}{rcl}
(\lambda^{\#}(\alpha : \kappa).\, u) \# \tau & \xrightarrow{0}_{\#} & u[\alpha \leftarrow \tau] \\
(\lambda^{\#}(x : \tau).\, u) \# u_0 & \xrightarrow{0}_{\#} & u[x \leftarrow u_0] \\
(\lambda^{\#}(u_1 \simeq u_2) : \tau.\, u) \# (u_1' \simeq u_2') : \tau' & \xrightarrow{0}_{\#} & u \\
(\lambda^{\#}((u')*p : \tau).\, u) \#^{*} u_0 & \xrightarrow{0}_{\#} & u[*p \leftarrow u_0] \\
([\pi.\, a])^{p} & \xrightarrow{*p \leftarrow \mathsf{reuse}(a[\pi \leftarrow p])}_{\#} & a[\pi \leftarrow p]
\end{array}
$$

$$
\textsc{Red-NoLabel}\ \frac{a \xrightarrow{0}_{\#} b}{\mathcal{C}[a] \xrightarrow{0}_{\#} \mathcal{C}[b]}
\qquad
\textsc{Red-Label}\ \frac{a \xrightarrow{*p \leftarrow u}_{\#} b}{\mathcal{C}[a] \xrightarrow{(*p \leftarrow u) \uparrow \mathcal{C}}_{\#} (\mathcal{C}[*p \leftarrow u])[b]}
$$

Figure 7.12: Definition of $\longrightarrow_{\#}$

$$
\begin{array}{lcl}
(*p \leftarrow u) \uparrow [\pi.\, []] & = & 0 \\
(*p \leftarrow u) \uparrow [] \# \tau & = & *p \leftarrow u \\
(*p \leftarrow u) \uparrow [] \# u' & = & *p \leftarrow u \\
(*p \leftarrow u) \uparrow [] \# (u_1' \simeq u_2') : \tau & = & *p \leftarrow u \\
(*p \leftarrow u) \uparrow [] \#^{*} u' & = & *p \leftarrow u \\
(*p \leftarrow u) \uparrow ([])^{q} & = & *p \leftarrow u
\end{array}
$$

Figure 7.13: Label translation for $m\mathsf{ML}$

- *either* $a_i = u_i\ u'_i$*, and then* $\Gamma, \Delta \cup T_i \vdash u_i : \tau_i \to \sigma_i$ *and* $\Gamma, \Delta \cup T_i \vdash u'_i : \tau_i$*;*

- *or* $a_i = (u_i)^{p_i}$ *and then* $\Gamma, \Delta \cup T_i \vdash u_i : [\sigma_i]$

. Then, consider a set of non-expansive term expand_i *for all* i *for which* a_i *is a thunk application, and a set of terms* $\mathsf{expanded}_i$*, such that:*

- *For all* i*,* $\Gamma, \Delta \cup T_i \vdash \mathsf{expand}_i : \mathsf{bool}$*,* i.e. *the* expand_i *are boolean non-expansive expressions.*

- *For all* i*,* $\Gamma, \Delta \cup T_i, (\mathsf{expand}_i \simeq \mathsf{True}) : \mathsf{bool} \vdash u_i : \sigma_i \simeq [p.\ \mathsf{expanded}_i] : \sigma_i$*,* i.e. *the expression inside the thunk application is equivalent to* $[p.\mathsf{expanded}_i]$ *whenever* expand_i *is* True*.*

Then, consider a' *the term obtained by replacing all thunk application atoms in* a *with* $\mathsf{match}\ \mathsf{expand}_i\ \mathsf{with}\ \mathsf{False} \to (u_i)^{p_i} \mid \mathsf{True} \to \mathsf{expanded}_i[\pi \leftarrow p_i]$*. We say that* a' *is a one-step expansion of* a*.*

We way that a' *is an expansion of* a *if it can be obtained from* a *by a finite number of expansions.* ♢

Definition 7.2 (Term equality for mML)**.** *Consider an environment* Γ*, two terms* a_u*,* b_u*, and two types* τ_a*,* τ_b*. Consider expansions* a *and* b *of these terms, and suppose there exists* p*,* Δ_a*,* Δ_b *such that* $\Gamma \vdash^p a : \tau_a \Rightarrow \Delta_a$ *and* $\Gamma \vdash^p b : \tau_b \Rightarrow \Delta_b$*. Let* $S_a = \mathsf{labels}(a)$*,* $S_b = \mathsf{labels}(b)$

Suppose $\mathsf{atoms}(a) = (*p_i \leftarrow (T_{a,i}, a_i))^{p_i \in \mathsf{labels}(a)}$ *and* $\mathsf{atoms}(b) = (*q_j \leftarrow (T_{b,j}, b_j))^{q_j \in \mathsf{labels}(b)}$*, with for all* i*,*

- *either* $a_i = u_i\ u'_i$*, and then* $\Gamma, \Delta_a \cup T_{a,i} \vdash u_{a,i} : \tau_{a,i} \to \sigma_{a,i}$ *and* $\Gamma, \Delta_a \cup T_{a,i} \vdash u'_{a,i} : \tau_{a,i}$*;*

- *or* $a_i = (u_i)^{p_i}$ *and then* $\Gamma, \Delta_a \cup T_{a,i} \vdash u_i : [\sigma_{a,i}]$

and similarly for Moreover, suppose that for all i*, and similarly for all* j*,* $\Gamma, \Delta_b \cup T_{b,j} \vdash u_{b,j} : \tau_{b,j} \to \sigma_{b,j}$ *and* $\Gamma, \Delta_b \cup T_{b,j} \vdash u'_{b,j} : \tau_{b,j}$*. Finally, suppose that* $\Delta_a = ((u^c_{a,i}) * p_i : \sigma_{a,i})^i$ *and* $\Delta_b = ((u^c_{b,j}) * q_j : \sigma_{b,j})^j$*.*

We define a type $\mathsf{unthunk}\ \tau$ *with one constructor* $\mathsf{Unthunk} : \tau \to \mathsf{unthunk}\ \tau$ *to represent thunk-valued atoms. Then, for each* i*, we define* $\mathsf{atom}_{a,i} = \mathsf{App}(u_i, u'_i)$ *and* $\mathsf{atomty}_{a,i} = \mathsf{app}\ (\tau_i, \sigma_i)$ *if the expression defining* p_i *is of the form* $u_i\ u'_i$*, and* $\mathsf{atom}_{a,i} = \mathsf{Unthunk}\ u$*,* $\mathsf{atomty}_{a,i} = \mathsf{unthunk}\ \sigma_{a,i}$ *if the expression defining* p_i *is of the form* $(p_i)_-$*, and similarly for each* j

Then, we say that that a *and* b *are equal, noted* $\mathsf{TermsEqual}(\Gamma \vdash^p a_u : \tau_a \simeq b_u : \tau_b)$ *if and only if:*

- *For all* k*,*

$$\begin{aligned}
&\Gamma, \Delta_a, \Delta_b, ((\textsc{Index}_\ell(u^c_{a,i} \to p_i)^i \simeq \textsc{Index}_\ell(u^c_{b,j} \to q_j)^j) : \textsc{Index}_\ell(u^c_{a,i} \to \tau_i)^i)^{\ell < k} \\
&\quad \vdash \textsc{Index}_k(u^c_{a,i} \to \mathsf{atom}_{a,i})^i : \textsc{Index}_k(u^c_{a,i} \to \mathsf{atomty}_{a,i})^i \\
&\quad \simeq \textsc{Index}_k(u^c_{b,i} \to \mathsf{atom}_{b,i})^i : \textsc{Index}_k(u^c_{b,i} \to \mathsf{atomty}_{b,i})^i
\end{aligned}$$

  i.e. *the* k*-th atom on the left-hand side is equivalent to the* k*-th atom on the right-hand side.*

- *The final expressions are equal:*

$$\Gamma, \Delta_a, \Delta_b,$$
$$((\textsc{Index}_\ell(u^c_{a,i} \to p_i)^i \simeq \textsc{Index}_\ell(u^c_{b,j} \to q_j)^j) : \textsc{Index}_\ell(u^c_{a,i} \to \tau_i)^i)^{\ell<(n,m)}$$
$$\vdash \mathsf{reuse}(a) : \tau_a \simeq \mathsf{reuse}(b) : \tau_b$$

where n and m are the number of labels appearing in a and b respectively.

$\diamondsuit$

This equality has the same properties as the eML equality. In particular, for transitivity, we can perform expansion on all terms and then apply transitivity as for eML.

Then, reduction preserves equality:

Lemma 7.1 (Meta reduction preserves equality)**.** *Suppose* $\Gamma \vdash^p a : \tau \Rightarrow \Delta$ *and* $a \longrightarrow_\# a'$*. Then,* $\mathsf{TermsEqual}(\Gamma \vdash^p a : \tau \simeq a' : \tau)$*.*

Proof. A meta-reduction is either a non-expansive reduction, that preserves equality as in Lemma 6.25, or a reduction of a thunk, in which case we need to mark the expanded thunk as to-be-expanded and then recover identical terms. $\square$

7.3 Metatheory of mML

7.3.1 Confluence

First, we prove that the mML reduction $\longrightarrow_\#$ is confluent. This is not immediate because of labels: we need to show that the same labels are substituted whatever the order thunks are reduced in, and independently of where mML reduction moves them. As explained when presenting labeled ML, this is only true for well-typed terms and well-formed types and kinds.

Well-labeling and subject reduction

We have not yet proved subject reduction for $\longrightarrow_\#$, and we intend to use confluence in the proof. Fortunately, the reduction $\longrightarrow_\#$ does not drop expansive expressions and labels are preserved: the obstacles to proving confluence using only the well-labeling of §5 are absent here.

Our first step is thus to extend the well-labeling defined in Figure 5.8 to mML. The new rules, given in Figure 7.14 for terms, and Figure 7.16 for types, are essentially a simplification of the typing rules for mML, where only the label information is kept: the judgment $\Sigma; \Gamma \vdash^p_l a \Rightarrow \Delta$ reads as *with path variables* Σ *and labels* Γ *in context, and with current label* p*, the term* a *is well-labeled and emits labels in* Δ. We also give a well-labeling judgment for kinds and terms. Since they do not emit labels, the judgments are simpler: $\Sigma; \Gamma \vdash_l \tau$ (*resp.* $\Sigma; \Gamma \vdash_l \kappa$) states that the type τ (*resp.* the kind κ) is well-labeled in the environment Σ, Γ.

The well-labeling judgment for mML is more lenient than the typing judgment:

$$\textsc{L-Var}\quad \frac{\text{orthogonal}(\Gamma)}{\Sigma;\Gamma \vdash_l^p x \Rightarrow \varnothing} \qquad \textsc{L-TAbs}\quad \frac{\Sigma;\Gamma \vdash_l^p u \Rightarrow \varnothing}{\Sigma;\Gamma \vdash_l^p \Lambda(\alpha : \mathsf{Typ}).\, u \Rightarrow \varnothing}$$

$$\textsc{L-TApp}\quad \frac{\Sigma;\Gamma \vdash_l^p a \Rightarrow \Delta \qquad \Sigma;\Gamma \vdash_l \tau}{\Sigma;\Gamma \vdash_l^p a\ \tau \Rightarrow \Delta}$$

$$\textsc{L-Fix}\quad \frac{\pi \# \Sigma \qquad \Sigma,\pi;\Gamma \vdash_l^\pi a \Rightarrow \Delta \qquad \Sigma;\Gamma \vdash_l \tau_1 \qquad \Sigma;\Gamma \vdash_l \tau_2}{\Sigma;\Gamma \vdash_l^p \mathsf{fix}^\pi\ x\ (y : \tau_1) : \tau_2\ .\ a \Rightarrow \varnothing}$$

$$\textsc{L-App}\quad \frac{\Sigma;\Gamma \vdash_l^p a \Rightarrow \Delta_1 \qquad \Sigma;\Gamma,\Delta_1 \vdash_l^p b \Rightarrow \Delta_2 \qquad p \leqslant q \qquad q \perp \Gamma,\Delta_1,\Delta_2}{\Sigma;\Gamma \vdash_l^q a^q\ b \Rightarrow \Delta_1,\Delta_2,q}$$

$$\textsc{L-Let}\quad \frac{\Sigma;\Gamma \vdash_l^p a \Rightarrow \Delta_1 \qquad \Sigma;\Gamma,\Delta_1 \vdash_l^p b \Rightarrow \Delta_2}{\Sigma;\Gamma \vdash_l^p \mathsf{let}\ x = a\ \mathsf{in}\ b \Rightarrow \Delta_1,\Delta_2}$$

$$\textsc{L-Con}\quad \frac{\text{orthogonal}(\Gamma) \qquad (\Sigma;\Gamma \vdash_l \tau_i)^i \qquad (\Sigma;\Gamma,(\Delta_k)^{k>j} \vdash_l^p a_j \Rightarrow \Delta_j)^j}{\Sigma;\Gamma \vdash_l^p d(\tau_i)^i(a_j)^j \Rightarrow (\Delta_j)^j}$$

$$\textsc{L-Match}\quad \frac{\begin{array}{c}\Sigma;\Gamma \vdash_l^p a \Rightarrow \Delta \\ (\Sigma;\Gamma \vdash_l \tau_k)^k \qquad (\Sigma;\Gamma,\Delta \vdash_l^p a_i \Rightarrow \Delta_i)^i \qquad \text{orthogonal}((\Delta_i)^i)\end{array}}{\Sigma;\Gamma \vdash_l^p \mathsf{match}\ a\ \mathsf{with}\ (d_i(\tau_k)^k(x_{ij})^j \to b_i)^i \Rightarrow \Delta,(\Delta_i)^i}$$

$$\textsc{L-Reuse}\quad \frac{\text{orthogonal}(\Gamma) \qquad p \in \Gamma \vee p \perp \Gamma}{\Sigma;\Gamma \vdash_l^p *p \Rightarrow \varnothing} \qquad \textsc{L-TyMetaAbs}\quad \frac{\Sigma;\Gamma \vdash_l \kappa \qquad \Sigma;\Gamma \vdash_l^p u \Rightarrow \varnothing}{\Sigma;\Gamma \vdash_l^p \lambda^\#(\alpha : \kappa).\, u \Rightarrow \varnothing}$$

$$\textsc{L-TyMetaApp}\quad \frac{\Sigma;\Gamma \vdash_l^p a \Rightarrow \Delta \qquad \Sigma;\Gamma \vdash_l \tau_0}{\Sigma;\Gamma \vdash_l^p a \# \tau_0 \Rightarrow \Delta} \qquad \textsc{L-TermMetaAbs}\quad \frac{\Sigma;\Gamma \vdash_l \tau_0 \qquad \Sigma;\Gamma \vdash_l^p u \Rightarrow \varnothing}{\Sigma;\Gamma \vdash_l^p \lambda^\#(x : \tau_0).\, u \Rightarrow \varnothing}$$

$$\textsc{L-TermMetaApp}\quad \frac{\Sigma;\Gamma \vdash_l^p a \Rightarrow \Delta \qquad \Sigma;\Gamma \vdash_l^p u \Rightarrow \varnothing}{\Sigma;\Gamma \vdash_l^p a \# u \Rightarrow \Delta}$$

$$\textsc{L-EqMetaAbs}\quad \frac{\Sigma;\Gamma \vdash_l \sigma \qquad \Sigma;\Gamma \vdash_l^p u_1 \Rightarrow \varnothing \qquad \Sigma;\Gamma \vdash_l^p u_2 \Rightarrow \varnothing \qquad \Sigma;\Gamma \vdash_l^p u \Rightarrow \varnothing}{\Sigma;\Gamma \vdash_l^p \lambda^\#((u_1 \simeq u_2) : \sigma).\, u \Rightarrow \varnothing}$$

$$\textsc{L-EqMetaApp}\quad \frac{\Sigma;\Gamma \vdash_l^p a \Rightarrow \Delta \qquad \Sigma;\Gamma \vdash_l \sigma \qquad \Sigma;\Gamma \vdash_l^p u_1 \Rightarrow \varnothing \qquad \Sigma;\Gamma \vdash_l^p u_2 \Rightarrow \varnothing}{\Sigma;\Gamma \vdash_l^p a \# (u_1 \simeq u_2) : \sigma \Rightarrow \Delta}$$

Figure 7.14: Well-labeling for types

$$\textsc{L-PtrMetaAbs}\quad \frac{q \perp \Gamma \qquad \Sigma;\Gamma \vdash_l^p u_0 \Rightarrow \varnothing \qquad \Sigma;\Gamma \vdash_l \tau \qquad \Sigma;\Gamma, q \vdash_l^p u \Rightarrow \varnothing}{\Sigma;\Gamma \vdash_l^p \lambda^{\#}((u_0)*q : \tau).\, u \Rightarrow \varnothing}$$

$$\textsc{L-PtrMetaApp}\quad \frac{\Sigma;\Gamma \vdash_l^p a \Rightarrow \Delta \qquad \Sigma;\Gamma \vdash_l^p u \Rightarrow \varnothing}{\Sigma;\Gamma \vdash_l^p a \,\#^*\, u \Rightarrow \Delta} \qquad \textsc{L-Thunk}\quad \frac{\pi \# \Sigma \qquad \Sigma, \pi;\Gamma \vdash_l^\pi a \Rightarrow \Delta}{\Sigma;\Gamma \vdash_l^p [\pi.\, a] \Rightarrow \varnothing}$$

$$\textsc{L-Unthunk}\quad \frac{p \leqslant q \qquad q \perp \Gamma, \Delta \qquad \Sigma;\Gamma \vdash_l^p a \Rightarrow \Delta}{\Sigma;\Gamma \vdash_l^p (a)^q \Rightarrow \Delta, q}$$

Figure 7.15: Well-labeling for types (cont.)

$$\textsc{LT-Var}\quad \frac{\text{orthogonal}(\Gamma)}{\Sigma;\Gamma \vdash_l \alpha} \qquad \textsc{LT-Datatype}\quad \frac{\text{orthogonal}(\Gamma) \qquad (\Sigma;\Gamma \vdash_l \tau_i)^i}{\Sigma;\Gamma \vdash_l \zeta\,(\tau_i)_i} \qquad \textsc{LT-Arr}\quad \frac{\Sigma;\Gamma \vdash_l \tau_1 \qquad \Sigma;\Gamma \vdash_l \tau_2}{\Sigma;\Gamma \vdash_l \tau_1 \to \tau_2}$$

$$\textsc{LT-All}\quad \frac{\Sigma;\Gamma \vdash_l \tau}{\Sigma;\Gamma \vdash_l \forall(\alpha : \mathsf{Typ})\, \tau} \qquad \textsc{LT-Match}\quad \frac{\Sigma;\Gamma \vdash_l^p u \Rightarrow \varnothing \qquad (\Sigma;\Gamma \vdash_l \tau_k)^k \qquad (\Sigma;\Gamma \vdash_l \sigma_i)^i}{\Sigma;\Gamma \vdash_l \mathsf{match}\ u\ \mathsf{with}\ (d_i(\tau_k)^k(x_{ij})^j \to \sigma_i)^i}$$

$$\textsc{LT-TyAbs}\quad \frac{\Sigma;\Gamma \vdash_l \kappa \qquad \Sigma;\Gamma \vdash_l \tau}{\Sigma;\Gamma \vdash_l \lambda^{\#}(\alpha : \kappa).\, \tau} \qquad \textsc{LT-TyApp}\quad \frac{\Sigma;\Gamma \vdash_l \sigma \qquad \Sigma;\Gamma \vdash_l \tau}{\Sigma;\Gamma \vdash_l \sigma \# \tau}$$

$$\textsc{LT-TermAbs}\quad \frac{\Sigma;\Gamma \vdash_l \tau \qquad \Sigma;\Gamma \vdash_l \sigma}{\Sigma;\Gamma \vdash_l \lambda^{\#}(x : \tau).\, \sigma} \qquad \textsc{LT-TermApp}\quad \frac{\Sigma;\Gamma \vdash_l \tau \qquad \Sigma;\Gamma \vdash_l^p u \Rightarrow \varnothing}{\Sigma;\Gamma \vdash_l \tau \# u}$$

$$\textsc{LT-TyMetaArr}\quad \frac{\Sigma;\Gamma \vdash_l \kappa \qquad \Sigma;\Gamma \vdash_l \tau}{\Sigma;\Gamma \vdash_l \Pi(\alpha : \kappa)\, \tau} \qquad \textsc{LT-TermMetaArr}\quad \frac{\Sigma;\Gamma \vdash_l \tau \qquad \Sigma;\Gamma \vdash_l \sigma}{\Sigma;\Gamma \vdash_l \Pi(x : \tau)\, \sigma}$$

$$\textsc{LT-EqMetaArr}\quad \frac{\Sigma;\Gamma \vdash_l^p u_1 \Rightarrow \varnothing \qquad \Sigma;\Gamma \vdash_l^p u_2 \Rightarrow \varnothing \qquad \Sigma;\Gamma \vdash_l \tau \qquad \Sigma;\Gamma \vdash_l \sigma}{\Sigma;\Gamma \vdash_l \Pi(u_1 \simeq u_2) : \tau\, \sigma}$$

$$\textsc{LT-PtrMetaArr}\quad \frac{p \perp \Gamma \qquad \Sigma;\Gamma \vdash_l^p u \Rightarrow \varnothing \qquad \Sigma;\Gamma \vdash_l \tau \qquad \Sigma;\Gamma, p \vdash_l \sigma}{\Sigma;\Gamma \vdash_l \Pi((u)*p : \tau)\, \sigma} \qquad \textsc{LT-Thunk}\quad \frac{\Sigma;\Gamma \vdash_l \tau}{\Sigma;\Gamma \vdash_l [\tau]}$$

Figure 7.16: Well-labeling for types

$$\frac{\text{orthogonal}(\Gamma) \qquad \kappa \in \{\mathsf{Typ}, \mathsf{Sch}, \mathsf{Met}\}}{\Sigma; \Gamma \vdash_l \kappa}\ \textsc{LK-Base} \qquad \frac{\Sigma; \Gamma \vdash_l \kappa \qquad \Sigma; \Gamma \vdash_l \kappa'}{\Sigma; \Gamma \vdash_l \Pi(\alpha : \kappa)\, \kappa'}\ \textsc{LK-TyFun}$$

$$\frac{\Sigma; \Gamma \vdash_l \tau \qquad \Sigma; \Gamma \vdash_l \kappa}{\Sigma; \Gamma \vdash_l \Pi(x : \tau)\, \kappa}\ \textsc{LK-TermFun}$$

Figure 7.17: Well-labeling for kinds

Lemma 7.2 (Well-typed expressions are well-labeled)**.** *Consider a typing environment Γ. We can obtain a labeling environment $\Sigma; \Gamma'$ from it by removing everything except path variables and the names of labels. Similarly, from a list of computed results for typing Δ with their types and conditions, we can obtain a list of computed results for labeling Δ'. Then,*

- *if $\Gamma \vdash \kappa$ wf, then $\Sigma; \Gamma' \vdash_l \kappa$;*
- *if $\Gamma \vdash \tau : \kappa$, then $\Sigma; \Gamma' \vdash_l \tau$;*
- *if $\Gamma \vdash p : a\tau\Delta$, then $\Sigma; \Gamma' \vdash_l^p a \Rightarrow \Delta'$.*

Proof. By induction on the derivation. Some hypotheses requires in the labeling derivation but absent from the $m\mathsf{ML}$ derivation can be extracted from the already-present typing hypotheses (Lemma 6.12). □

We need weakening and substitution for this judgment:

Lemma 7.3 (Weakening)**.** *Suppose $\Sigma; \Gamma \vdash_l^p a \Rightarrow \Delta$. Then:*

- *Suppose $q \leqslant p$, and q independent of Γ, Δ. Then, $\Sigma; \Gamma \vdash_l^q a \Rightarrow \Delta$.*
- *Suppose $\pi \notin \Sigma$. Then, $\Sigma, \pi; \Gamma \vdash_l^p a \Rightarrow \Delta$.*
- *Suppose $q \perp p, \Gamma, \Delta$. Then, $\Sigma; \Gamma, q \vdash_l^p a \Rightarrow \Delta$.*

We have the same results for types and kinds.

Proof. By induction on the derivation: these changes translate to weakening in the premises. The independence side-conditions stay true in the derivations. □

Lemma 7.4 (Substitution)**.** *Suppose $\Sigma; \Gamma \vdash_l^p a \Rightarrow \Delta$.*

- *Consider $\pi \in \Sigma$, and $q \perp \Gamma, \Delta$. Then, $\Sigma - \pi; \Gamma[\pi \leftarrow q] \vdash_l^{p[\pi \leftarrow q]} a[\pi \leftarrow q] \Rightarrow \Delta[\pi \leftarrow q]$.*
- *Consider x and a non-expansive term u such that $\Sigma; \Gamma \vdash_l^p u \Rightarrow \varnothing$. Then, $\Sigma; \Gamma \vdash_l^p a[x \leftarrow u] \Rightarrow \Delta$.*

We have the same results for types and kinds.

Proof. By induction on the derivation. □

We then prove subject reduction for the well-labeling. This lemma is similar to Lemma 5.11, except that we do not need a case for labels that disappear. It is also extended to handle types and kinds.

Lemma 7.5 (Subject reduction for well-labeling and $\longrightarrow_{\#}$)**.**

- *Suppose* $\Sigma; \Gamma \vdash_l \kappa$. *If* $\kappa \longrightarrow_{\#} \kappa'$, *then* $\Sigma; \Gamma \vdash_l \kappa'$.
- *Suppose* $\Sigma; \Gamma \vdash_l \tau$. *If* $\tau \longrightarrow_{\#} \tau'$, *then* $\Sigma; \Gamma \vdash_l \tau'$.
- *Suppose* $\Sigma; \Gamma \vdash_l^p a \Rightarrow \Delta$. *Moreover, suppose* $a \xrightarrow{l}_{\#} b$. *Then:*
  - *If* $l = 0$, *we have* $\Sigma; \Gamma \vdash_l^p b \Rightarrow \Delta$.
  - *If* $l = *q \leftarrow u$, *we have* $\Sigma; \Gamma \vdash_l^p b \Rightarrow \Delta'$ *where* $\Delta' = \Delta - \{q\} \uplus S$ *where* S *is an orthogonal set of labels prefixed by* q, *and* $q \in \Delta$.

Proof. The proof is similar to Lemma 5.11, except that we do not have to handle the case of reduction of a pattern matching, thus labels never disappear. □

The parallel reduction

In order to prove confluence, we follow Takahashi [1989]. We define a parallel reduction for $\longrightarrow_{\#}$. For types, it is noted $a \rhd^{\sigma} a'$, where σ is a substitution of labels, corresponding to the labels emitted by the one-step reductions represented by this parallel reduction. When σ is empty, we will also write $a \rhd a'$. It also applies to types and kinds. Since all terms appearing in types and kinds are non-expansive, σ will always be empty and we write $\tau \rhd \tau'$ and $\kappa \rhd \kappa'$. The reduction has two parts: it has congruence rules, presented in Figure 7.18 for terms and Figure 7.19 for types and kinds, and reduction rules, presented in Figure 7.20. The congruence rules (for example P-App) reduce each subexpression. If labels are generated, the labels are substituted in all the subexpressions, as well as in the substitutions that were generated while reducing these expressions (until there is nothing to substitute). The rules for let binding (P-Let) and pattern matching (P-Match) also ensure that the substitutions stay well-scoped, as is already done by label translation.

We will need a few substitution and reflexivity results on $\rhd$:

Lemma 7.6 (Reflexivity of $\rhd$)**.** *The parallel reduction is reflexive:*

- *For all* a, $a \rhd^{\varnothing} a$.
- *For all* τ, $\tau \rhd \tau$.
- *For all* κ, $\kappa \rhd \kappa$.

Proof. By mutual induction on terms, types, and kinds: apply the lemma for each subexpression, and always choose the congruence rules. □

Lemma 7.7 (Substitution for $\rhd$)**.** *Let* σ *be a substitution (of term, type or label variables, or of labels). Consider* $\sigma^{\rhd}$ *a reduced version of this substitution (*i.e. *where some terms and types have been replaced by their reductions by* $\rhd$*). Then,*

- *If* $a \rhd^l a'$, *then* $a[\sigma] \rhd^{l[\sigma^{\rhd}]} a'[\sigma^{\rhd}]$.
- *If* $\tau \rhd \tau'$, *then* $\tau[\sigma] \rhd \tau'[\sigma^{\rhd}]$.
- *If* $\kappa \rhd \kappa'$, *then* $\kappa[\sigma] \rhd \kappa'[\sigma^{\rhd}]$.

$$\textsc{P-Var}\quad \frac{}{x \rhd^{\varnothing} x}$$

$$\textsc{P-Let}\quad \frac{a \rhd^{l_a} a' \qquad b \rhd^{l_b} b'}{\mathsf{let}\ x = a\ \mathsf{in}\ b \rhd^{l_a, l_b[l_a, x \leftarrow \mathsf{reuse}(a')]} \mathsf{let}\ x = a'\ \mathsf{in}\ b'[l_a]}$$

$$\textsc{P-Fix}\quad \frac{a \rhd^{l} a' \qquad \tau_1 \rhd \tau_1' \qquad \tau_2 \rhd \tau_2'}{\mathsf{fix}^{\pi}\ x\ (y : \tau_2') : \tau_1\ .\ a \rhd^{\varnothing} \mathsf{fix}^{\pi}\ x\ (y : \tau_2') : \tau_1'\ .\ a'}$$

$$\textsc{P-App}\quad \frac{a \rhd^{l_a} a' \qquad b \rhd^{l_b} b'}{a^p\ b \rhd^{l_a, l_b[l_a]} a'^p\ b'[l_a]}$$

$$\textsc{P-TAbs}\quad \frac{u \rhd^{\varnothing} u'}{\Lambda(\alpha : \mathsf{Typ}).\ u \rhd^{\varnothing} \Lambda(\alpha : \mathsf{Typ}).\ u'}$$

$$\textsc{P-TApp}\quad \frac{a \rhd^{l} a'}{a\ \tau \rhd^{l} a'\ \tau}$$

$$\textsc{P-Match}\quad \frac{a \rhd^{l} a' \qquad (\tau_k \rhd \tau_k')^k \qquad (b_i \rhd^{l_i} b_i')^i}{\begin{array}{c}\mathsf{match}\ a\ \mathsf{with}\ (d_i(\tau_k)^k(x_{ij})^j \to b_i)^i \rhd^{l, (\mathsf{matchreuse}(a')\ \mathsf{with}\ d_i(\tau_k)^k(x_{ij})^j \to l_i[l])^i} \\ \mathsf{match}\ a'\ \mathsf{with}\ (d_i(\tau_k')^k(x_{ij})^j \to b_i'[l])^i\end{array}}$$

$$\textsc{P-Reuse}\quad \frac{}{*p \rhd^{\varnothing} *p}$$

$$\textsc{P-TyMetaAbs}\quad \frac{\kappa \rhd \kappa' \qquad u \rhd^{\varnothing} u'}{\lambda^{\#}(\alpha : \kappa).\ u \rhd^{\varnothing} \lambda^{\#}(\alpha : \kappa').\ u'}$$

$$\textsc{P-TyMetaApp}\quad \frac{u \rhd^{\varnothing} u' \qquad \tau \rhd \tau'}{u \# \tau \rhd^{\varnothing} u' \# \tau'}$$

$$\textsc{P-TermMetaAbs}\quad \frac{\tau \rhd \tau' \qquad u \rhd^{\varnothing} u'}{\lambda^{\#}(x : \tau).\ u \rhd^{\varnothing} \lambda^{\#}(x : \tau').\ u'}$$

$$\textsc{P-TermMetaApp}\quad \frac{u_1 \rhd^{\varnothing} u_1' \qquad u_2 \rhd^{\varnothing} u_2'}{u_1 \# u_2 \rhd^{\varnothing} u_1' \# u_2'}$$

$$\textsc{P-EqMetaAbs}\quad \frac{u_0 \rhd^{\varnothing} u_0' \qquad u_1 \rhd^{\varnothing} u_1' \qquad u_2 \rhd^{\varnothing} u_2' \qquad \tau \rhd \tau'}{\lambda^{\#}(u_1 \simeq u_2) : \tau.\ u_0 \rhd^{\varnothing} \lambda^{\#}(u_1' \simeq u_2') : \tau'.\ u_0'}$$

$$\textsc{P-EqMetaApp}\quad \frac{u_0 \rhd^{\varnothing} u_0' \qquad u_1 \rhd^{\varnothing} u_1' \qquad u_2 \rhd^{\varnothing} u_2' \qquad \tau \rhd \tau'}{u_0 \# (u_1 \simeq u_2) : \tau \rhd^{\varnothing} u_0' \# (u_1' \simeq u_2') : \tau'}$$

$$\textsc{P-PtrMetaAbs}\quad \frac{u_0 \rhd^{\varnothing} u_0' \qquad u \rhd^{\varnothing} u' \qquad \tau \rhd \tau'}{\lambda^{\#}((u_0)*p : \tau).\ u \rhd \lambda^{\#}((u_0')*p : \tau').\ u'}$$

$$\textsc{P-PtrMetaApp}\quad \frac{u_1 \rhd^{\varnothing} u_1' \qquad u_2 \rhd^{\varnothing} u_2'}{u_1 \# u_2 \rhd^{\varnothing} u_1' \# u_2'}$$

$$\textsc{P-Thunk}\quad \frac{a \rhd^{l} a'}{[\pi.\ a] \rhd^{\varnothing} [\pi.\ a']}$$

$$\textsc{P-Unthunk}\quad \frac{u \rhd^{\varnothing} u'}{(u)^p \rhd^{\varnothing} (u')^p}$$

Figure 7.18: Parallel reduction: congruence for terms

$$\textsc{P-KBase}\quad \frac{\kappa \in \{\mathsf{Typ}, \mathsf{Sch}, \mathsf{Met}\}}{\kappa \rhd \kappa}$$

$$\textsc{P-KTyFun}\quad \frac{\kappa_1 \rhd \kappa_1' \qquad \kappa_2 \rhd \kappa_2'}{\Pi(\alpha : \kappa_1)\, \kappa_2 \rhd \Pi(\alpha : \kappa_1')\, \kappa_2'}$$

$$\textsc{P-KTermFun}\quad \frac{\tau \rhd \tau' \kappa \rhd \kappa'}{\Pi(x : \tau)\, \kappa \rhd \Pi(x : \tau')\, \kappa'}$$

$$\textsc{P-TyVar}\quad \frac{}{\alpha \rhd \alpha}$$

$$\textsc{P-TyFun}\quad \frac{\tau_1 \rhd \tau_1' \qquad \tau_2 \rhd \tau_2'}{\tau_1 \to \tau_2 \rhd \tau_1' \to \tau_2'}$$

$$\textsc{P-TyDatatype}\quad \frac{(\tau_i \rhd \tau_i')^i}{\zeta\, (\tau_i)^i \rhd \zeta\, (\tau_i')^i}$$

$$\textsc{P-TyAll}\quad \frac{\tau \rhd \tau'}{\forall(\alpha : \mathsf{Typ})\, \tau \rhd \forall(\alpha : \mathsf{Typ})\, \tau'}$$

$$\textsc{P-TyMatch}\quad \frac{a \rhd^l a' \qquad (\tau_k \rhd \tau_k')^k \qquad (\sigma_i \rhd \sigma_i')^i}{\mathsf{match}\ u\ \mathsf{with}\ (d_i(\tau_k)^k(x_{ij})^j \to \sigma_i)^i \rhd \mathsf{match}\ u'\ \mathsf{with}\ (d_i(\tau_k')^k(x_{ij})^j \to \sigma_i')^i}$$

$$\textsc{P-TyTyAbs}\quad \frac{\kappa \rhd \kappa' \qquad \tau \rhd \tau'}{\lambda^{\#}(\alpha : \kappa).\, \tau \rhd \lambda^{\#}(\alpha : \kappa').\, \tau'}$$

$$\textsc{P-TyTyApp}\quad \frac{\sigma \rhd \sigma' \qquad \tau \rhd \tau'}{\sigma \# \tau \rhd \sigma' \# \tau'}$$

$$\textsc{P-TyTermAbs}\quad \frac{\tau \rhd \tau' \qquad \sigma \rhd \sigma'}{\lambda^{\#}(x : \tau).\, \sigma \rhd \lambda^{\#}(x : \tau').\, \sigma'}$$

$$\textsc{P-TyTermApp}\quad \frac{\tau \rhd \tau' \qquad u \rhd u'}{\tau \# u \rhd \tau' \# u'}$$

$$\textsc{P-TyMetaTyFun}\quad \frac{\kappa \rhd \kappa' \qquad \tau \rhd \tau'}{\Pi(\alpha : \kappa)\, \tau \rhd \Pi(\alpha : \kappa')\, \tau'}$$

$$\textsc{P-TyMetaTermFun}\quad \frac{\tau \rhd \tau' \qquad \sigma \rhd \sigma'}{\Pi(x : \tau)\, \sigma \rhd \Pi(x : \tau')\, \sigma'}$$

$$\textsc{P-TyMetaEqFun}\quad \frac{u_1 \rhd u_1' \qquad u_2 \rhd u_2' \qquad \tau \rhd \tau' \qquad \sigma \rhd \sigma'}{\Pi(u_1 \simeq u_2) : \tau\, \sigma \rhd \Pi(u_1' \simeq u_2') : \tau'\, \sigma'}$$

$$\textsc{P-TyMetaPtrFun}\quad \frac{u \rhd u' \qquad \tau \rhd \tau' \qquad \sigma \rhd \sigma'}{\Pi((u)*p : \tau)\, \sigma \rhd \Pi((u')*p : \tau')\, \sigma'}$$

$$\textsc{P-TyThunk}\quad \frac{\tau \rhd \tau'}{[\tau] \rhd [\tau']}$$

Figure 7.19: Parallel reductions: congruence for types and kinds

$$\textsc{P-TyTermRed}\quad \frac{\sigma \rhd \sigma' \qquad u \rhd u'}{(\lambda^{\#}(x:\tau).\,\sigma) \# u \rhd \sigma'[x \leftarrow u']}$$

$$\textsc{P-TyTyRed}\quad \frac{\sigma \rhd \sigma' \qquad \tau \rhd \tau'}{(\lambda^{\#}(\alpha:\kappa).\,\sigma) \# \tau \rhd \sigma'[\alpha \leftarrow \tau']}$$

$$\textsc{P-TermTermRed}\quad \frac{u_1 \rhd u_1' \qquad u_2 \rhd u_2'}{(\lambda^{\#}(x:\tau).\,u_1) \# u_2 \rhd u_1'[x \leftarrow u_2']}$$

$$\textsc{P-TermTyRed}\quad \frac{u \rhd u' \qquad \tau \rhd \tau'}{(\lambda^{\#}(\alpha:\kappa).\,u) \# \tau \rhd u'[\alpha \leftarrow \tau']}$$

$$\textsc{P-TermEqRed}\quad \frac{u \rhd u'}{(\lambda^{\#}(u_1 \simeq u_2):\tau.\,u) \# (u_1 \simeq u_2):\tau' \rhd u'}$$

$$\textsc{P-TermPtrRed}\quad \frac{u_2 \rhd u_2' \qquad u_3 \rhd u_3'}{(\lambda^{\#}((u_1)*p:\tau).\,u_2) \# u_3 \rhd u_2'[*p \leftarrow u_3']}$$

$$\textsc{P-TermThunkRed}\quad \frac{a \rhd^{l} a'}{([\pi.\,a])^{p} \rhd^{*p \leftarrow \mathsf{reuse}(a')[\pi \leftarrow p]} a'[\pi \leftarrow p]}$$

Figure 7.20: Parallel reductions: reduction rules

Proof. By induction on the derivation of the reduction. When reaching a variable substituted by σ, use the reduction from its value for σ to its value for $\sigma^{\rhd}$. □

We also need to prove that the reduction is compatible with making terms reusable:

Lemma 7.8 (Reusability for $\rhd$)**.** *Suppose $a \rhd^{l} a'$. Then, there exists u such that $\mathsf{reuse}(a) \rhd^{l} u$ and $\mathsf{reuse}(a_2) = u[l]$.*

Proof. By induction on the derivation. □

Embedding of one reduction into the other

We then need to prove that $\longrightarrow_{\#}$ is a subset of $\rhd$, and that $\rhd$ is a subset of $\longrightarrow_{\#}^{*}$. From this, we will deduce subject reduction for $\rhd$, and that $\longrightarrow_{\#}$ is confluent if $\rhd$ is confluent.

We need a few lemmas analogous to lemmas from §5. The following lemma is the equivalent for $m\mathsf{ML}$ of Lemma 5.7:

Lemma 7.9 (Disjoint labels)**.** *Suppose $\Sigma;\Gamma \vdash_{l}^{p} a \Rightarrow \Delta$. Then, Γ and Δ are orthogonal and all labels in Δ are prefixed by p.*

Proof. Similar to Lemma 5.7. □

The two next lemmas state that the typing describes what labels may be emitted by reducing a term. We need one for the usual reduction and one for the parallel reduction:

Lemma 7.10 (Emitted labels)**.** *Suppose $\Sigma;\Gamma \vdash_{l}^{q} a \Rightarrow \Delta$.*

- *If* $a \xrightarrow{*p \leftarrow u}_{\#} a'$*, then* $p \in \Delta$*.*
- *If* $a \rhd^{l} a'$*, for all* $*p \leftarrow u \in l$*,* $p \in \Delta$*.*

Proof. By induction on the derivation of the reduction. □

The usual reduction and the parallel reductions do not perform the same substitutions of labels: in $a\ b$, if the reduction of b emits a substitution, it will be substituted in a by the usual reduction but not by the parallel reduction. We need to prove that this has no consequences on the result: this is the case because all labels that could be emitted by b are out of scope for a.

Lemma 7.11.

- *If* $\Sigma; \Gamma \vdash_{l}^{q} a \Rightarrow \Delta$ *and* $p \notin \Gamma, \Delta$*, then* $a[*p \leftarrow u] = a$*.*
- *If* $\Sigma; \Gamma \vdash_{l} \tau$ *and* $p \perp \Gamma$*, then* $\tau[*p \leftarrow u] = \tau$*.*
- *If* $\Sigma; \Gamma \vdash_{l} \kappa$ *and* $p \perp \Gamma$*, then* $\kappa[*p \leftarrow u] = \kappa$*.*

Proof. By induction on the derivation. □

We also need the same result for the emitted substitutions. Here, we need a stronger hypothesis because emitted substitutions may reference labels that are suffixes of already existing labels.

Lemma 7.12. *Suppose* $\Sigma; \Gamma \vdash_{l}^{q} a \Rightarrow \Delta$*. Let* $*p \leftarrow u$ *be a substitution with* $q \leqslant p$ *and* $p \perp \Gamma, \Delta$*. Suppose* $a \rhd^{l} b$*. Then,* $l[*p \leftarrow u] = l$*.*

Proof. By induction on the reduction, we prove that all labels appearing in the terms in l are suffixes of the labels in Γ, Δ. □

We prove that the parallel reduction embeds the usual reduction. This relies on the well-labeling to be true: when propagating reduction labels, the parallel reduction only substitutes in the part of the context that are *after* the evaluation of the label, while $\longrightarrow_{\#}$ substitutes in the whole context. The well-labeling guarantees that such labels cannot appear in the parts that are before the evaluation.

Lemma 7.13 ($\longrightarrow_{\#}$ is included in $\rhd$)**.**

- *Suppose* $\Sigma; \Gamma \vdash_{l}^{p} a \Rightarrow \Delta$*. Then if* $a \xrightarrow{l}_{\#} a'$*,* $a \rhd^{l} a'$*.*
- *Suppose* $\Sigma; \Gamma \vdash_{l} \tau$*. Then, if* $\tau \longrightarrow_{\#} \tau'$*,* $\tau \rhd \tau'$*.*
- *Suppose* $\Sigma; \Gamma \vdash_{l} \kappa$*. Then, if* $\kappa \longrightarrow_{\#} \kappa'$*,* $\kappa \rhd \kappa'$

Proof. By induction on the reduction, inverting the typing derivation along the way.

Head reduction map directly to reduction rules of the parallel reduction. The subexpressions of the reduction rules are not reduced: we map them to themselves using reflexivity for parallel reduction (Lemma 7.6).

For context rules, we will take the reduction in context $\mathcal{C} = \mathsf{let}\ x = a\ \mathsf{in}\ []$ as an example, as it has the most complicated features, and we will suppose that the term in the context reduces with an attached substitution.

Suppose $b \xrightarrow{*p \leftarrow u}_{\#} a'$. Then, $\mathcal{C}[b] \xrightarrow{*p \leftarrow u[x \leftarrow \mathsf{reuse}(a)]}_{\#} \mathsf{let}\ x = a[*p \leftarrow u]\ \mathsf{in}\ b[*p \leftarrow u[x \leftarrow \mathsf{reuse}(a)]]$. Consider the labeling derivation of $\mathsf{let}\ x = a\ \mathsf{in}\ b$. The last rule must be L-Let. Inverting the rule Let, we obtain $\Sigma; \Gamma \vdash_l^q a \Rightarrow \Delta$ and $\Sigma; \Gamma, \Delta \vdash_l^q b \Rightarrow \Delta'$. By Lemma 7.10, we know that $p \in \Delta'$. Thus, $p \perp \Gamma, \Delta$. Then, we can apply Lemma 7.11: we have $a[*p \leftarrow u] = a$.

We conclude by applying P-Let: we have $a \rhd^{\varnothing} a$ (by reflexivity, Lemma 7.6) and $b \rhd^{*p \leftarrow u} b'$ by induction hypothesis, thus $\mathsf{let}\ x = a\ \mathsf{in}\ b \rhd^{*p \leftarrow u[x \leftarrow \mathsf{reuse}(a)]} \mathsf{let}\ x = a\ \mathsf{in}\ b'$. □

We also prove that the parallel reduction is a subset of the (iterated) usual reduction:

Lemma 7.14 ($\rhd$ included in $\longrightarrow_{\#}$)**.**

- *Suppose* $\Sigma; \Gamma \vdash_l \kappa$ *and* $\kappa \rhd \kappa'$*. Then,* $\kappa \longrightarrow_{\#}^{*} \kappa'$*.*
- *Suppose* $\Sigma; \Gamma \vdash_l \tau$ *and* $\tau \rhd \tau'$*. Then,* $\tau \longrightarrow_{\#}^{*} \tau'$*.*
- *Suppose* $\Sigma; \Gamma \vdash_l^a \Delta \Rightarrow$ *, and* $a \rhd^l a'$*. Then,* $a \xrightarrow{l}{}_{\#}^{*} a'$*.*

Proof. By mutual induction on the parallel reduction derivations. We describe the reasoning for two representative rules:

- If the last rule is a congruence rule such as P-Let:

$$\text{P-Let}\quad \frac{a \rhd^{l_a} a' \qquad b \rhd^{l_b} b'}{\mathsf{let}\ x = a\ \mathsf{in}\ b \rhd^{l_a, l_b[l_a, x \leftarrow \mathsf{reuse}(a')]} \mathsf{let}\ x = a'\ \mathsf{in}\ b'[l_a]}$$

  The last rule of the well-labeling derivation must be L-Let:

$$\text{L-Let}\quad \frac{\Sigma; \Gamma \vdash_l^p a \Rightarrow \Delta_1 \qquad \Sigma; \Gamma, \Delta_1 \vdash_l^p b \Rightarrow \Delta_2}{\Sigma; \Gamma \vdash_l^p \mathsf{let}\ x = a\ \mathsf{in}\ b \Rightarrow \Delta_1, \Delta_2}$$

  By induction hypothesis on the well-labeled term b, we have $b \xrightarrow{l_b}_{\#} b'$. All the labels $q \in l_b$ are in Δ_2 by Lemma 7.10. Thus, $l_2 \perp \Gamma, \Delta_1$, and for any subset l_2' of l_2, $a[l_2'] = a$. Thus, (by applying the context rule for $\mathcal{C} = \mathsf{let}\ x = a\ \mathsf{in}\ []$, we have $\mathsf{let}\ x = a\ \mathsf{in}\ b \xrightarrow{l_b[x \leftarrow \mathsf{reuse}(a)]}{}_{\#}^{*} \mathsf{let}\ x = a\ \mathsf{in}\ b'$.

  By induction hypothesis on the well-labeled term a, we have $a \xrightarrow{l_a}_{\#} a'$. Thus, by applying the context rule for $\mathcal{C} = \mathsf{let}\ x = []\ \mathsf{in}\ b'$, we obtain $\mathsf{let}\ x = a\ \mathsf{in}\ b' \xrightarrow{l_a}{}_{\#}^{*} \mathsf{let}\ x = a\ \mathsf{in}\ b'[l_a]$. By composing these two reductions we obtain: $\mathsf{let}\ x = a\ \mathsf{in}\ b \xrightarrow{l_a, l_b[l_a, x \leftarrow \mathsf{reuse}(a')]}{}_{\#}^{*} \mathsf{let}\ x = a'\ \mathsf{in}\ b'[l_a]$.

- If the last rule is a reduction rule such as P-TermThunkRed:

$$\text{P-TermThunkRed}\quad \frac{a \rhd^l a'}{([\pi.\, a])^p \rhd^{*p \leftarrow \mathsf{reuse}(a')[\pi \leftarrow p]} a'[\pi \leftarrow p]}$$

  By inverting the well-labeling derivation (L-Unthunk then L-Thunk), we obtain that a is well-typed. Thus, by induction hypothesis, $a \xrightarrow{l}{}_{\#}^{*} a'$.

Then, $([\pi.a])^p \longrightarrow^*_\# ([\pi.a'])^p$ (the $[\pi.[]]$ removes the substitution emitted by the reduction of a). By head-reduction, we have

$$([\pi.\, a'])^p \xrightarrow{*p \leftarrow \mathsf{reuse}(a')[\pi \leftarrow p]}_\# a'[\pi \leftarrow p]$$

We conclude by composition. □

Confluence for $\rhd$

We now define the maximal reduction for $\rhd$: $a \rhd^l a'$ is a maximal reduction of a if we always use the reduction rules whenever possible. Then, we write this $a \rhd\rhd^l a'$. We can similarly define a maximal reduction for types and kinds.

The core of the confluence proof is the following lemmas, expressing that if we start by doing a step of non-maximal reduction, we can always obtain the maximal reduction by performing another step of reduction. The term version is more complex to express because it also needs to state how the substitutions labelling these reductions are related: intuitively, performing a first reduction might make some redexes that were hidden under a lambda exposed. Reducing these lambdas in the second reduction step will emit labels that are not visible when doing the maximal reduction in one step. The labels of these now-revealed redexes are suffixes of labels emitted in the first reduction.

Lemma 7.15.

- *Suppose $\Sigma; \Gamma \vdash_l \kappa_0$. Consider a maximal reduction $\kappa_0 \rhd\rhd \kappa_2$. Suppose we also have $\kappa_0 \rhd \kappa_1$. Then, $\kappa_1 \rhd \kappa_2$.*
- *Suppose $\Sigma; \Gamma \vdash_l \tau_0$. Consider a maximal reduction $\tau_0 \rhd\rhd \tau_2$. Suppose we also have $\tau_0 \rhd \tau_1$. Then, $\tau_1 \rhd \tau_2$.*
- *Suppose $\Sigma; \Gamma \vdash^p_l a_0 \Rightarrow \Delta$. Consider a maximal reduction $a_0 \rhd\rhd^{l_{12}} a_2$. Suppose we also have $a_0 \rhd^{l_1} a_1$. Then, there exists l_2, l'_2 and $l^\rhd_1$ such that:*
  - $a_1 \rhd^{l_2, l'_2} a_2$.
  - $l_1[l_2, l'_2] \rhd l^\rhd_1$, i.e. $l^\rhd_1$ *associates labels to reduced versions of their corresponding terms in l_1.*
  - *The labels in l'_2 are suffixes of the labels in l_1,*
  - $l_{12} = l^\rhd_1[l_2, l'_2], l_2$.

Proof. By mutual induction on terms, types and kinds. We consider a few representative cases:

- If the last rule of the maximal reduction is P-Thunk, the last rule of the non-maximal reduction is also P-Thunk.

$$\frac{a_0 \rhd\rhd^{l_{a,12}} a_2}{[\pi.\, a_0] \rhd\rhd^{\varnothing} [\pi.\, a_2]} \qquad \frac{a_0 \rhd^{l_{a,1}} a_1}{[\pi.\, a_0] \rhd^{\varnothing} [\pi.\, a_1]}$$

  Inverting the well-labeling derivation, we obtain that a is well-labeled. Then by induction hypothesis, there exists $l_{a,2}, l'_{a,2}$ such that $a_1 \rhd^{l_{a,2}, l'_{a,2}} a_2$. Then, applying P-Thunk, $[\pi.\, a_1] \rhd^{\varnothing} [\pi.\, a_2]$. We conclude with $l_1 = l^\rhd_1 = l_2 = l'_2 = l_{12} = \varnothing$.

- If the last rule of the maximal reduction is P-Unthunk, then the last rule of the non-maximal reduction is also P-Unthunk (the last rule cannot be P-TermThunkRed because if it was possible to apply it it would be applied in the maximal reduction).

$$\frac{u_0 \rhd\rhd^{\varnothing} u_2}{(u_0)^p \rhd\rhd^{\varnothing} (u_2)^p} \qquad \frac{u_0 \rhd^{\varnothing} u_1}{(u_0)^p \rhd^{\varnothing} (u_1)^p}$$

  Inverting the well-labeling derivation, we obtain that u is well-labeled. Then by induction hypothesis, we have $u_1 \rhd^{\varnothing} u_2$. Thus, applying the rule P-Unthunk, we obtain $(u_1)^p \rhd^{\varnothing} (u_2)^p$.

- If the last rule of the maximal reduction is P-TermThunkRed and the last rule of the non-maximal reduction is P-Unthunk: then, the second-to-last rule of the non-maximal reduction is P-Thunk.

$$\frac{a_0 \rhd\rhd^{l_{a,12}} a_2}{([\pi.\, a_0])^p \rhd\rhd^{*p \leftarrow \mathsf{reuse}(a_2)[\pi \leftarrow p]} a_2[\pi \leftarrow p]} \qquad \frac{\dfrac{a_0 \rhd^{l_{a,1}} a_1}{[\pi.\, a_0] \rhd^{\varnothing} [\pi.\, a_1]}}{([\pi.\, a_0])^p \rhd^{\varnothing} ([\pi.\, a_1])^p}$$

  By inverting the well-labeling derivation, we obtain that a_0 is well-labeled. By induction hypothesis, there exists $l_{a,2}, l'_{a,2}$ such that $a_1 \rhd^{l_{a,2}, l'_{a,2}} a_2$ Then, by P-TermThunkRed, we obtain $([\pi.a_1])^p \rhd^{*p \leftarrow \mathsf{reuse}(a_2)[\pi \leftarrow p]} a_2[\pi \leftarrow p]$. We conclude, with $l_{12} = l_2 = *p \leftarrow \mathsf{reuse}(a_2)[\pi \leftarrow p]$, and $l_1 = l_1^{\rhd} = l'_2 = \varnothing$.

- If the last rule of the maximal reduction is P-TermThunkRed and the last rule of the non-maximal reduction is not P-Unthunk, it must be P-TermThunkRed too.

$$\frac{a_0 \rhd\rhd^{l_{a,12}} a_2}{([\pi.\, a_0])^p \rhd\rhd^{*p \leftarrow \mathsf{reuse}(a_2)[\pi \leftarrow p]} a_2[\pi \leftarrow p]}$$

$$\frac{a_0 \rhd^{l_{a,1}} a_1}{([\pi.\, a_0])^p \rhd^{*p \leftarrow \mathsf{reuse}(a_1)[\pi \leftarrow p]} a_1[\pi \leftarrow p]}$$

  Similarly to the previous case, we have $a_0 \rhd\rhd^{l_{a,12}} a_2$, $a_0 \rhd^{l_{a,1}} a_1$, $a_1 \rhd^{l_{a,2}, l'_{a,2}} a_2$. By substitution, we have $a_1[\pi \leftarrow p] \rhd^{(l_{a,2}, l'_{a,2})[\pi \leftarrow p]} a_2[\pi \leftarrow p]$.

  We have $l_1 = *p \leftarrow \mathsf{reuse}(a_1)[\pi \leftarrow p]$. By substitution (Lemma 7.7), we have $a_1[\pi \leftarrow p] \rhd^{(l_2, l'_2)[\pi \leftarrow p]} a_2[\pi \leftarrow p]$. Then, by Lemma 7.8, there exists u such that $\mathsf{reuse}(a_1) \rhd^{\varnothing} u$ and $*a_2[\pi \leftarrow p] = u[(l_2, l'_2)[\pi \leftarrow p]]$.

  Let $l_2 = \varnothing$, $l'_2 = (l_{a,2}, l'_{a,2})[\pi \leftarrow p]$, Then, let $l_1^{\rhd} = *p \leftarrow u$. We have $u[l_2, l'_2] = *a_2[\pi \leftarrow p]$. Thus $l_1^{\rhd}[l_2, l'_2] = *p \leftarrow \mathsf{reuse}(a_2)[\pi \leftarrow p]$

  By subject reduction for well-labeling (Lemma 7.5), the labels emitted by the reduction $a_1[\pi \leftarrow p] \rhd^{(l_{a,2}, l'_{a,2})[\pi \leftarrow p]} a_2[\pi \leftarrow p]$ must be prefixed by p (the output label set for $([\pi.\, a_0])^p$ only contains p, thus the new labels appearing may only start with p and parallel reduction only emits exposed labels by Lemma 7.10).

  Then, $l_{12} = l_1^{\rhd}, l_2$ and we conclude.

- If the last rule of the maximal reduction is P-Let, the last rule of the other reduction is also P-Let:

$$\frac{a_0 \rhd\rhd^{l_{a,12}} a_2 \qquad b_0 \rhd\rhd^{l_{b,12}} b_2}{\mathsf{let}\ x = a_0\ \mathsf{in}\ b_0 \rhd\rhd^{l_{a,12}, l_{b,12}[l_{a,12}, x \leftarrow \mathsf{reuse}(a_2)]} \mathsf{let}\ x = a_2\ \mathsf{in}\ b_2[l_{a,12}]}$$

$$\frac{a_0 \rhd\rhd^{l_{a,1}} a_1 \qquad b_0 \rhd\rhd^{l_{b,1}} b_1}{\mathsf{let}\ x = a_0\ \mathsf{in}\ b_0 \rhd\rhd^{l_{a,1}, l_{b,1}[l_{a,1}, x \leftarrow \mathsf{reuse}(a_1)]} \mathsf{let}\ x = a_1\ \mathsf{in}\ b_1[l_{a,1}]}$$

By inverting the well-labeling derivation, we obtain that a_0 and b_0 are well-labeled. Thus, by induction hypothesis, there exists $l^{\rhd}_{a,1}, l_{a,2}, l'_{a,2}$ such that $a_1 \rhd^{l_{a,2}, l'_{a,2}} a_2$ and $l_{a,12} = l^{\rhd}_{a,1}[l_{a,2}, l'_{a,2}], l_{a,2}$, and $l^{\rhd}_{b,1}, l_{b,2}, l'_{b,2}$ such that $b_1 \rhd^{l_{b,2}, l'_{b,2}} b_2$ and $l_{a,12} = l^{\rhd}_{b,1}[l_{b,2}, l'_{b,2}], l_{b,2}$.

By substitution (Lemma 7.7), we have $b_1[l_{a,1}] \rhd^{l_{b,2}[l^{\rhd}_{a,1}], l'_{b,2}[l^{\rhd}_{a,1}]} b_2[l^{\rhd}_{a,1}]$.

Thus, applying P-Let, we get:

$$\begin{aligned}&\mathsf{let}\ x = a_1\ \mathsf{in}\ b_1[l_{a,1}]\\ &\quad \rhd^{l_{a,2}, l'_{a,2}, (l_{b,2}, l_{b,2})[l^{\rhd}_{a,1}][x \leftarrow a_2, l_{a,2}, l'_{a,2}]}\ \mathsf{let}\ x = a_2\ \mathsf{in}\ b_2[l^{\rhd}_{a,1}][l_{a,2}, l'_{a,2}]\end{aligned}$$

Let us first show that the reduced expression is the one we want: we have $b_2[l^{\rhd}_{a,1}][l_{a,2}, l'_{a,2}] = b_2[l^{\rhd}_{a,1}[l_{a,2}, l'_{a,2}], l_{a,2}, l'_{a,2}] = b_2[l^{\rhd}_{a,1}[l_{a,2}, l'_{a,2}], l_{a,2}] = b_2[l_{a,12}]$. The second-to-last equality is true because the labels in $l'_{a,2}$ do not appear in b_2 because they are strict suffixes of labels that may be emitted by the reduction of a_0 (by induction hypothesis).

By a similar argument, we have:

$$\begin{aligned}&(l_{b,2}, l'_{b,2})[l^{\rhd}_{a,1}][x \leftarrow \mathsf{reuse}(a_2), l_{a,2}, l'_{a,2}]\\ &\quad = (l_{b,2}, l'_{b,2})[x \leftarrow \mathsf{reuse}(a_2), l^{\rhd}_{a,1}[l_{a,2}, l'_{a,2}], l_{a,2}, l'_{a,2}]\\ &\quad = (l_{b,2}, l'_{b,2})[x \leftarrow \mathsf{reuse}(a_2), l^{\rhd}_{a,1}[l_{a,2}, l'_{a,2}], l_{a,2}]\\ &\quad = (l_{b,2}, l'_{b,2})[x \leftarrow \mathsf{reuse}(a_2), l_{a,12}]\end{aligned}$$

We take $l_1 = l_{a,1}, l_{b,1}[x \leftarrow \mathsf{reuse}(a_1), l_{a,1}]$, $l^{\rhd}_1 = l^{\rhd}_{a,1}, l^{\rhd}_{b,1}[x \leftarrow \mathsf{reuse}(a_e), l^{\rhd}_{a,2}]$, $l_2 = l_{a,2}, l_{b,2}[x \leftarrow \mathsf{reuse}(a_2), l_{a,12}]$, $l'_2 = l'_{a,2}, l'_{b,2}[x \leftarrow \mathsf{reuse}(a_2), l_{a,12}]$.

Then, we have $l_{12} = l^{\rhd}_1[l_2, l'_2], l_2$. □

From this, we deduce the diamond property for the parallel reduction. We'll allow ourselves to write $\rhd$ without label for $\rhd^l$ for some substitution l.

Lemma 7.16 (Diamond property for $\rhd$)**.** *Consider a well-labeled term a_0 Suppose $a_0 \rhd a_1$ and $a_0 \rhd a_2$. Then, there exists a_{12} such that: $a_1 \rhd a_{12}$ and $a_2 \rhd a_{12}$.*

We have the same results on types and kinds.

Proof. Take a_{12} to be the maximal reduction of a_0. □

Lemma 7.17 (Confluence for $\rhd$)**.** *Suppose $a_0 \rhd^* a_1$ and $a_0 \rhd^* a_2$. Then, there exists a_{12} such that $a_1 \rhd^* a_{12}$ and $a_2 \rhd^* a_{12}$.*

We have the same result for types and kinds.

Proof. We will write the proof for types. Suppose $\tau_{0,0} \rhd \tau_{1,0} \rhd \ldots \rhd \tau_{n,0}$ and $\tau_{0,0} \rhd \tau_{0,1} \rhd \ldots \rhd \tau_{0,m}$.

By subject reduction, the $\tau_{i,0}$ and $\tau_{0,j}$ are well-labeled.

For each $0 < i \leqslant n$, $0 < j \leqslant m$, suppose $\tau_{i-1,j-1}$ is well-labeled and let $\tau_{i,j}$ be the type such that $\tau_{i,j-1} \rhd \tau_{i,j}$ and $\tau_{i-1,j} \rhd \tau_{i,j}$. It exists by the diamond property (Lemma 7.16). Moreover, it is well-labeled by subject reduction.

Then $\tau_{n,m}$ is such that $\tau_{n,0} \rhd \tau_{n,1} \rhd \ldots \rhd \tau_{n,m}$ and $\tau_{0,m} \rhd \tau_{1,m} \rhd \ldots \rhd \tau_{n,m}$, ie. $\tau_{n,0} \rhd^* \tau_{n,m}$ and $\tau_{0,m} \rhd^* \tau_{n,m}$. □

As a consequence, we have confluence for $\longrightarrow_{\#}$:

Theorem 7.1 (Confluence for $\longrightarrow_{\#}$)**.** *The reduction $\longrightarrow_{\#}$ is confluent: consider a well-typed term a_0, and suppose $a_0 \xrightarrow{l_1}_{\#} a_1$, $a_0 \xrightarrow{l_2}_{\#} a_2$. Then, there exists a_3 and l'_1, l'_2 such that $a_1 \xrightarrow{l'_1}_{\#} a_3$ and $a_2 \xrightarrow{l'_2}_{\#} a_3$.*

The same result is true for types and kinds.

Proof. Well-typed terms, types, and kinds are well-labeled (Lemma 7.2) $\rhd$ is confluent on well-typed terms, types and kinds, and $\rhd^*$ and $\longrightarrow^*_{\#}$ coincide. □

7.3.2 Basic properties of the typing derivation

We need weakening, reflexivity and substitution lemmas as for eML (see §6.3.1), as well as symmetry of the equality. Neither the proofs nor the statements change significantly, and we will not repeat them here.

7.3.3 Strong normalization

Our goal in this section is to prove that meta-reduction and type reduction are strongly normalizing. The notations used in this proof are only used here, and will be re-used for other purposes later in this article.

Theorem 7.2 (Normalization for meta-reduction)**.** *The meta-reduction $\longrightarrow_{\#}$ is strongly normalizing.*

As usual, the proof uses *reducibility sets* [Girard et al., 1989]. We need some unusual properties to deal with labels: terms must stay strongly normalizing, even when labels are substituted by arbitrary terms of the right types. Since the labels are always substituted by expressions whose types are eML types, and thus substitution of labels does not interact with the mML reduction, we actually allow substitution of labels by *safe* term:

Definition 7.3 (Safe terms)**.** *The set of safe terms $\mathcal{B}_{\mathsf{a}}$ is defined as the maximal set of terms such that*

- *All terms in $\mathcal{B}_{\mathsf{a}}$ are strongly normalizing.*
- *If $a \in \mathcal{B}_{\mathsf{a}}$, then a does not begin with a meta-abstraction or a meta thunk.*
- *Suppose $a \in \mathcal{B}_{\mathsf{a}}$. If $a \xrightarrow{0}_{\#} a'$, $a' \in \mathcal{B}_{\mathsf{a}}$. If $a \xrightarrow{*p \leftarrow u}_{\#} a'$, then both $a' \in \mathcal{B}_{\mathsf{a}}$ and $u \in \mathcal{B}_{\mathsf{a}}$.*

◇

$$\begin{array}{lcl} \langle\!\langle \mathsf{Typ} \rangle\!\rangle & = & \{\mathcal{B}_{\mathsf{a}}\} \\ \langle\!\langle \mathsf{Sch} \rangle\!\rangle & = & \{\mathcal{B}_{\mathsf{a}}\} \\ \langle\!\langle \mathsf{Met} \rangle\!\rangle & = & \mathcal{C}_{\mathsf{a}} \end{array} \qquad \begin{array}{lcl} \langle\!\langle \Pi(\alpha : \kappa_1)\, \kappa_2 \rangle\!\rangle & = & \langle\!\langle \kappa_1 \rangle\!\rangle \to \langle\!\langle \kappa_2 \rangle\!\rangle \\ \langle\!\langle \Pi(x : \tau)\, \kappa \rangle\!\rangle & = & \mathbf{1} \to \langle\!\langle \kappa \rangle\!\rangle \end{array}$$

Figure 7.21: Interpretation of kinds as sets of interpretations

The set of safe terms is stable by $\mathsf{reuse}(_)$: reuse distributes over reduction, and does not create meta abstractions or thunks.

Lemma 7.18 (Stability by label substitution)**.** *Suppose $a \in \mathcal{B}_{\mathsf{a}}$, and consider a label p and $u \in \mathcal{B}_{\mathsf{a}}$. Then, $a[*p \leftarrow u] \in \mathcal{B}_{\mathsf{a}}$, and we have the same property for partial substitution (where only some labels are substituted).*

Proof. All reductions of $a[*p \leftarrow u]$ are either reductions of a or of u: because u can never begin with a constructor for $\longrightarrow_{\#}$, there can be no redex spanning the substitution boundary. □

We now define reducibility sets. The unusual property here is CR4, which asserts that labels can be arbitrarily substituted. CR2 needs to check that generated labels are still safe.

Definition 7.4 (Reducibility set)**.** *A set $\mathcal{S}$ of terms is called a* reducibility set *if it respects the properties CR1-4 below. We write $\mathcal{C}_{\mathsf{a}}$ the set of reducibility sets of terms.*

CR1 Every term $a \in \mathcal{S}$ is strongly normalizing.

CR2 If $a \in \mathcal{S}$ and $a \xrightarrow{0}_{\#} a'$ then $a' \in \mathcal{S}$; if $a \xrightarrow{*p \leftarrow u}_{\#} a'$ then $a' \in \mathcal{S}$ and $u \in \mathcal{B}_{\mathsf{a}}$.

*CR3 Suppose a is not a meta-abstraction (*i.e. *it is a neutral term for meta-reduction), and for all a' such that $a \xrightarrow{l}_{\#} a'$ we have $a' \in \mathcal{S}$ and moreover if $l = *p \leftarrow u$ then $u \in \mathcal{B}_{\mathsf{a}}$. Then $a \in \mathcal{S}$.*

CR4 If $a \in \mathcal{S}$ and $u \in \mathcal{B}_{\mathsf{a}}$, then $a[*p \leftarrow u] \in \mathcal{S}$

Similarly, replacing terms with types and kinds, we obtain a version of the properties CR1-4 for sets of types and sets of kinds. A set of types or kinds is called a reducibility set if it respects those properties, and we write $\mathcal{C}_{\mathsf{t}}$ the set of reducibility sets of types, and $\mathcal{C}_{\mathsf{k}}$ the set of reducibility sets of kinds. ♢

Lemma 7.19. *$\mathcal{B}_{\mathsf{a}}$ is a reducibility set.*

Proof. The properties CR1-3 are immediate from the definition. CR4 is exactly Lemma 7.18. □

Let us all $\mathcal{N}_{\mathsf{k}}$, $\mathcal{N}_{\mathsf{t}}$, and $\mathcal{N}_{\mathsf{a}}$ the maximal reducibility sets of kinds, types and terms.

Definition 7.5 (Interpretation of types and kinds)**.** *We define an interpretation $\langle\!\langle \kappa \rangle\!\rangle$ of kinds as sets of possible interpretations of types, with $\mathbf{1}$ the set with one element $\bullet$. Type-level abstractions and applications are interpreted as functions in set theory. The interpretation is given on Figure 7.21.*

On Figure 7.22, we also define an interpretation $[\![\kappa]\!]_\rho$ of a kind κ as a set of types and an interpretation $[\![\tau]\!]_\rho$ of a type τ as a set of terms, both under

$$
\begin{array}{rcl}
[\![\mathsf{Typ}]\!]_\rho = [\![\mathsf{Sch}]\!]_\rho = [\![\mathsf{Met}]\!]_\rho & = & \mathcal{N}_\mathsf{t} \\
[\![\Pi(\alpha : \kappa_1)\ \kappa_2]\!]_\rho & = & \left\{ \tau \in \mathcal{N}_\mathsf{t} \;\middle|\; \begin{array}{l} \forall \tau' \in [\![\kappa_1]\!]_\rho, [\![\tau']\!]_\rho \text{ defined} \\ \qquad \Longrightarrow \tau \# \tau' \in [\![\kappa_2]\!]_{\rho[\alpha \leftarrow [\![\tau']\!]_\rho]} \end{array} \right\} \\
[\![\Pi(x : \tau)\ \kappa]\!]_\rho & = & \{ \tau \in \mathcal{N}_\mathsf{t} \mid \forall u \in [\![\tau]\!]_\rho, \tau \# u \in [\![\kappa]\!]_\rho \}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\alpha]\!]_\rho & = & \rho(\alpha) \\
[\![\tau_1 \to \tau_2]\!]_\rho = [\![\zeta\ \overline{\tau}]\!]_\rho = [\![\mathsf{match}\ a\ \mathsf{with}\ \ldots]\!]_\rho = [\![\forall(\alpha : \mathsf{Typ})\ \ldots]\!]_\rho & = & \mathcal{B}_\mathsf{a} \\
[\![\Pi(x : \tau_1)\ \tau_2]\!]_\rho & = & \{ a \in \mathcal{N}_\mathsf{a} \mid \forall u \in [\![\tau_1]\!]_\rho, a \# u \in [\![\tau_2]\!]_\rho \} \\
[\![\Pi(\alpha : \kappa)\ \tau]\!]_\rho & = & \left\{ a \in \mathcal{N}_\mathsf{a} \;\middle|\; \begin{array}{l} \forall \tau' \in [\![\kappa]\!]_\rho, [\![\tau']\!]_\rho \text{ defined} \\ \qquad \Longrightarrow a \# \tau' \in [\![\tau]\!]_{\rho[\alpha \leftarrow [\![\tau']\!]_\rho]} \end{array} \right\} \\
[\![\Pi(u_1 \simeq u_2) : \tau\ \tau']\!]_\rho & = & \{ a \in \mathcal{N}_\mathsf{a} \mid a \# (u_1 \simeq u_2) : \tau \in [\![\tau']\!]_\rho \} \\
[\![\Pi((u){*}p : \tau)\ \sigma]\!]_\rho & = & \{ a \in \mathcal{N}_\mathsf{a} \mid \forall u \in [\![\tau]\!]_\rho, a \#^* u \in [\![\sigma]\!]_\rho \} \\
[\![[\tau]]\!]_\rho & = & \{ a \in \mathcal{N}_\mathsf{a} \mid \forall p.(a)^p \in [\![\tau]\!]_\rho \} \\
[\![\lambda^\#(x : \tau').\ \tau]\!]_\rho & = & \lambda \bullet .\ [\![\tau]\!]_\rho \\
[\![\lambda^\#(\alpha : \kappa).\ \tau]\!]_\rho & = & \lambda(S_\alpha \in \langle\!\langle \kappa \rangle\!\rangle).\ [\![\tau]\!]_{\rho[\alpha \leftarrow S_\alpha]} \\
[\![\tau_1 \# \tau_2]\!]_\rho & = & [\![\tau_1]\!]_\rho\ [\![\tau_2]\!]_\rho \\
[\![\tau \# u]\!]_\rho & = & [\![\tau]\!]_\rho\ \bullet
\end{array}
$$

Figure 7.22: Interpretation of kinds and types as sets of types and terms

an assignment ρ of reducibility sets to type variables. The definition is done by mutual induction on types and kinds.

$\diamondsuit$

Definition 7.6 (Type environment)**.** *We will write $\rho \models \Gamma$ if for all $(\alpha : \kappa) \in \Gamma$, $\rho(\alpha) \in \langle\!\langle \kappa \rangle\!\rangle$.* $\diamondsuit$

Lemma 7.20 (Equal kinds have the same interpretation)**.** *If $\Gamma \vdash \kappa_1 \simeq \kappa_2$, then $\langle\!\langle \kappa_1 \rangle\!\rangle = \langle\!\langle \kappa_2 \rangle\!\rangle$.*

Proof. By induction on the derivation of the judgment $\Gamma \vdash \kappa_1 \simeq \kappa_2$.

- This is immediately true for KEq-Base.
- For KEq-Trans, apply the induction hypothesis and use transitivity of equality.
- For KEq-Type-Fun and KEq-Term-Fun: the subkinds have equal interpretations by induction hypothesis, so the interpretations are equal. $\square$

Lemma 7.21 (Interpretation of kinds and types)**.** *Assume $\rho \models \Gamma$. Then:*

- *If $\Gamma \vdash \kappa$ wf, then $\langle\!\langle \kappa \rangle\!\rangle$ and $[\![\kappa]\!]_\rho$ are well-defined, and $[\![\kappa]\!]_\rho \in \mathcal{C}_\mathsf{t}$.*
- *If $\Gamma \vdash \tau : \kappa$, then $[\![\tau]\!]_\rho$ is well-defined, and we have $[\![\tau]\!]_\rho \in \langle\!\langle \kappa \rangle\!\rangle$.*

Proof. By simultaneaous induction on the well-formedness derivations for types and kinds.

- For K-Var: if $(\alpha : \kappa) \in \Gamma$, since $\rho \models \Gamma$, then $\rho(\alpha) \in \langle\!\langle \kappa \rangle\!\rangle$.

- The types in the conclusion of rules K-Datatype, K-Arr, K-All, K-Match, K-SubTyp and K-SubSch are all interpreted as $\mathcal{N}_{\mathsf{a}}$, which is in the intepretation of the kinds (Typ, Sch and Met) that appear as conclusions of these rules.

- For rule Wf-Base, the interpretations of Typ, Sch and Met is $\mathcal{N}_{\mathsf{t}}$, which is in $\mathcal{C}_{\mathsf{t}}$.

- For the type-level meta abstractions (K-TyAbs and K-TermAbs), by induction. The extended type environment ρ' matches the extended type environment Γ': we still have $\rho' \models \Gamma'$. The interpretation is a function, which is in the interpretation of function kinds as sets of interpretations.

- Similarly for the type-level applications (K-TyApp and K-TermApp): we provide an argument of the right type, and the interpretation of the kind of the function guarantees that the application will be in the interpretation of the result kind.

- The rules Wf-TyFun, Wf-TermFun, K-TyMetaArr, K-TermMetaArr, K-EqMetaArr, K-PtrMetaArr and K-Thunk are similar. We only give the proof for K-TermMetaArr: take $\tau = \Pi(x : \tau_1)\, \tau_2$. Define $S_1 = [\![\tau_1]\!]_\rho$ and $S_2 = [\![\tau_2]\!]_\rho$. By induction hypothesis, S_1 and S_2 are reducibility sets. We will prove CR1-4 for $S = [\![\tau]\!]_\rho = \{a \in \mathcal{N}_{\mathsf{a}} \mid \forall u \in S_1, a \# u \in S_2\}$.

  CR1 S is a subset of $\mathcal{N}_{\mathsf{a}}$.

  CR2 Consider $a \in S$ and a' such that $a \xrightarrow{\sigma}_{\#} a'$. For a given $u \in S_1$, $a \# u \in S_2 \xrightarrow{\sigma}_{\#} a' \# u[\sigma]$. Thus, $a' \# u[\sigma] \in S_2$ by CR2 for S_2. Then, $a' \in S$. Moreover, we know that σ only substitutes a safe term.

  CR3 Consider a, not an abstraction, such that if $a \xrightarrow{\sigma}_{\#} a'$, $a' \in S$. For $u \in S_1$, we'll prove $a \# u \in S_2$. Since a is not an abstraction, $a \# u$ reduces either to $a' \# u[\sigma]$ with $a \xrightarrow{\sigma}_{\#} a'$, or $a \# u'$ with $u \xrightarrow{0}_{\#} u'$. In the first case, $a' \in S$ by hypothesis and $u[\sigma] \in S_1$, so $a' \# u[\sigma] \in S_2$. In the second case, $u' \in S_1$ by CR2, so $a \# u' \in S_2$. By CR3 for S_2, because $a \# u$ is not an abstraction, $a \# u \in S_2$.

  CR4 Consider $a \in S$, and a substitution $*p \leftarrow u$ with $u \in \mathcal{B}_{\mathsf{a}}$. For $u' \in S_1$, we have to prove $a[*p \leftarrow u] \# u' \in S_2$. This is a partial substitution of $a \# u' \in S_2$ thus is in S_2 too, □

We need the following substitution lemma:

Lemma 7.22 (Substitution)**.** *For all τ, κ, τ', α, a, u, x, p and ρ, we have:*

- $[\![\tau]\!]_{\rho[\alpha \leftarrow [\![\tau']\!]_\rho]} = [\![\tau[\alpha \leftarrow \tau']]\!]_\rho$
- $[\![\kappa]\!]_{\rho[\alpha \leftarrow [\![\tau']\!]_\rho]} = [\![\kappa[\alpha \leftarrow \tau']]\!]_\rho$
- $[\![\tau]\!]_\rho = [\![\tau[x \leftarrow a]]\!]_\rho = [\![\tau[*p \leftarrow u]]\!]_\rho$
- $[\![\kappa]\!]_\rho = [\![\kappa[x \leftarrow a]]\!]_\rho = [\![\kappa[*p \leftarrow u]]\!]_\rho$

Proof. By induction on types and kinds. This comes from the fact that the interpretation of type variables is obtained by looking them up in the context, and that the interpretation never considers the terms inside types and kinds. □

We then need to prove that conversion is sound with respect to the relation. We start by proving soundness of reduction:

Lemma 7.23 (Soundness of meta-reduction)**.** *Assume that $\rho \models \Gamma$, and τ, τ', κ, κ' are well-kinded (or well-formed) in Γ. Then $[\![\tau]\!]_\rho = [\![\tau']\!]_\rho$ whenever $\tau \longrightarrow^h_\# \tau'$ and $[\![\kappa]\!]_\rho = [\![\kappa']\!]_\rho$ whenever $\kappa \longrightarrow_\# \kappa'$.*

Proof. By structural induction on the context in which head reduction occurs. The only interesting context is the hole []. Consider the different kinds of head-reduction on types (the induction hypothesis is not concerned with terms, and there is no head-reduction on kinds). The cases of all meta-reductions are similar. Consider for example $(\lambda^\#(\alpha : \kappa).\, \tau) \# \tau' \longrightarrow^h_\# \tau[\alpha \leftarrow \tau']$. The interpretation of the left-hand side is $(\lambda S_\alpha \in \langle\!\langle\kappa\rangle\!\rangle.[\![\tau]\!]_{\rho[\alpha\leftarrow S_\alpha]})\ [\![\tau']\!]_\rho = [\![\tau]\!]_{\rho[\alpha\leftarrow[\![\tau']\!]_\rho]}$ and the interpretation of the right-hand side is $[\![\tau[\alpha \leftarrow \tau']]\!]_\rho = [\![\tau]\!]_{\rho[\alpha\leftarrow[\![\tau']\!]_\rho]}$ by substitution (Lemma 7.22). □

Lemma 7.24 (Soundness of conversion)**.** *If $\rho \models \Gamma$ and $\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$, then $[\![\tau_1]\!]_\rho = [\![\tau_2]\!]_\rho$. If $\Gamma \vdash \kappa_1 \simeq \kappa_2$, then $\langle\!\langle\kappa_1\rangle\!\rangle = \langle\!\langle\kappa_2\rangle\!\rangle$.*

Proof. By induction on the equality judgment.

- This is obviously true for TEq-Trans and KEq-Trans.
- By induction hypothesis, this is true for subtyping rules (TEq-SubTyp, TEq-SubSch).
- For all *e*ML congruence rules (TEq-Datatype, TEq-Arr, TEq-All, TEq-Match), the result holds because the interpretation of $[\![\tau_1]\!]_\rho = [\![\tau_2]\!]_\rho = \mathcal{B}_\mathsf{a}$.
- For all *m*ML congruence rules, the result holds: by induction hypothesis, the interpretation of the subtypes/kinds appearing as premises is the same, so the interpretation of the types/kinds is the same.
- For the *m*ML reduction rules, by soundness of meta-reduction (Lemma 7.23).
- For TEq-Split and the *e*ML reduction rules (TEq-ReduceMatch), the types τ_1 and τ_2 have kind Typ or Sch. Their interpretations are in $[\![\mathsf{Typ}]\!]_\rho = [\![\mathsf{Sch}]\!]_\rho = \{\mathcal{N}_\mathsf{t}\}$, thus are both equal to $\mathcal{N}_\mathsf{t}$. □

Now we can prove the fundamental lemma. We consider environments γ associating term variables to terms and type variables to types.

Definition 7.7. *We say $\rho, \gamma \models \Gamma$ if $\rho \models \Gamma$, for all $(x : \tau) \in \Gamma$, $\gamma(x) \in [\![\tau]\!]_\rho$, for all $*p \in \Gamma$, $\gamma(*p) \in \mathcal{B}_\mathsf{a}$, and for all $(\alpha : \kappa) \in \Gamma$, $[\![\gamma(\alpha)]\!]_\rho = \rho(\alpha)$.* ◇

Lemma 7.25 (Fundamental lemma)**.** *Suppose $\rho, \gamma \models \Gamma$. Then:*

- *If $\Gamma \vdash \kappa$ wf, then $\gamma(\kappa) \in \mathcal{N}_\mathsf{k}$.*
- *If $\Gamma \vdash \tau : \kappa$, then $\gamma(\tau) \in [\![\kappa]\!]_\rho$.*
- *If $\Gamma \vdash a : \tau$, then $\gamma(a) \in [\![\tau]\!]_\rho$.*

Proof. By mutual induction on typing, kinding, and well-formedness derivations. We will examine a few representative rules:

- If the last rule is a conversion, use soundness of conversion.
- If the last rule is a reuse: $\gamma(*p) \in \mathcal{B}_{\mathsf{a}}$, and the type in conclusion of the typing derivation has kind Sch so its interpretation is $\mathcal{B}_{\mathsf{a}}$.
- If the last rule is App: suppose $a = a_1 a_2$, with $\Gamma \vdash a_1 : \tau_1 \to \tau_2$ and $\Gamma \vdash a_2 : \tau_1$. Assume $\rho, \gamma \vDash \Gamma$. We have $\gamma(a_1) \in [\![\tau_1]\!]_\rho$ and $\gamma(a_2) \in [\![\tau_2]\!]_\rho$. We need to show: $\gamma(a_1\ a_2) = \gamma(a_1)\ \gamma(a_2) \in [\![\tau_2]\!]_\rho = \mathcal{B}_{\mathsf{a}}$ (because τ_2 is necessarily of kind Typ). Meta-reduction does not reduce ML application: then, any reduction is either a reduction of $\gamma(a_1)$ or $\gamma(a_2)$. By hypothesis, these two terms are strongly normalizing, thus $\gamma(a_1\ a_2) \in \mathcal{B}_{\mathsf{a}}$.
- If the last rule is a meta-application TermMetaApp

$$\frac{\Gamma \vdash u : \Pi(x : \tau_1)\ \tau_2 \qquad \Gamma \vdash u' : \tau_1}{\Gamma \vdash u \# u' : \tau_2[x \leftarrow u']}$$

  Consider $\rho, \gamma \vDash \Gamma$. Then, by induction hypothesis, we have: $\gamma(u) \in [\![\Pi(x : \tau_1)\ \tau_2]\!]_\rho$, and $\gamma(u') \in [\![\tau_1]\!]_\rho$. We thus have: $\gamma(u \# u') = \gamma(u) \# \gamma(u') \in [\![\tau_2]\!]_\rho$. By Lemma 7.22 $[\![\tau_2]\!]_\rho = [\![\tau_2[x \leftarrow u']]\!]_\rho$. It follows that $\gamma(u \# u') \in [\![\tau_2[x \leftarrow u']]\!]_\rho$.
- If the last rule is a meta-abstraction TermMetaAbs:

$$\frac{\Gamma, x : \tau_1 \vdash u : \tau_2}{\Gamma \vdash \lambda^{\#}(x : \tau_1).\ u : \Pi(x : \tau_1)\ \tau_2}$$

  Consider $\rho, \gamma \vDash \Gamma$. Consider $u' \in [\![\tau_1]\!]_\rho$. We need to prove that $(\lambda^{\#}(x : \gamma(\tau_1)).\ \gamma(u)) \# u' \in [\![\tau_2]\!]_\rho$. Let us write $u'' = \gamma(u)$, $\tau_1' = \gamma(\tau_1)$. Then we will proceed by induction on the sum of the longest derivation starting τ_1', u', and u'. Let us apply CR3 to $(\lambda^{\#}(x : \tau_1').\ u'') \# u'$, and consider all possible reductions:
  - If the reduction happens in τ_1', u'', or u', we obtain a term of the same shape with the number of possible reductions strictly decreased. Then, we apply the induction hypothesis.
  - Otherwise, we reduce the head redex: by hypothesis, $u''[x \leftarrow u'] \in [\![\tau_2]\!]_\rho$.
- If the last rule is K-TyAbs:

$$\frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Gamma \vdash \lambda^{\#}(\alpha : \kappa_1).\ \tau : \forall(\alpha : \kappa_1)\ \kappa_2}$$

  As previously, we need to prove that for all $\tau' \in [\![\kappa_1]\!]_\rho$, $(\lambda^{\#}(\alpha : \gamma(\kappa_1)).\gamma(\tau)) \# \tau' \in [\![\kappa_2]\!]_\rho$. We will only consider the case of the head redex:
  - if $[\![\tau']\!]_\rho$ is not defined, there is nothing to prove;
  - otherwise, we need to show that $\gamma(\tau)[\alpha \leftarrow \tau'] \in [\![\kappa_2]\!]_{\rho[\alpha \leftarrow [\![\tau']\!]_\rho]}$. Define $\gamma' = \gamma[\alpha \leftarrow \tau']$ and $\rho' = \rho[\alpha \leftarrow [\![\tau']\!]_\rho]$. Then, we have $\rho', \gamma' \vDash \Gamma, \alpha : \kappa_1$. We conclude by the induction hypothesis.

– If the last rule is Unthunk:

$$\textsc{Unthunk}\quad \frac{p \leqslant q \qquad q \perp \Gamma, \Delta \qquad \Gamma \vdash^p u : [\tau] \Rightarrow \Delta}{\Gamma \vdash^p (u)^q : \tau \Rightarrow \Delta, (\mathsf{True}) * q : \tau}$$

We examine the possible reductions as for TermMetaApp. When reducing the thunk, we emit $*\gamma(a)$ where a is the term currently inside the thunk. We have $\gamma(a) \in \mathcal{B}_{\mathsf{a}}$, thus $*\gamma(a) \in \mathcal{B}_{\mathsf{a}}$. □

We can now prove the main result of this section:

Proof. [Proof of Theorem 7.2] Consider a kind, type, or term X that is well-typed in a context Γ. We can take the identity substitution $\gamma(x) = x$ for all $x \in \Gamma$ and apply the fundamental lemma. All interpretations are subsets of $\mathcal{N}_{\mathsf{a}}$, thus $X \in \mathcal{N}_{\mathsf{a}}$. □

7.3.4 Subject reduction and soundness

In order to prove subject reduction and soundness, we need to understand better the equalities on $m\mathsf{ML}$ types. We prove a decomposition result: if we have an equality between two $m\mathsf{ML}$ types that start with a type constructor, then they must start with the safe constructor, and the arguments to the constructor are also equal. This is a stronger result than the same for $e\mathsf{ML}$: we assert that, even in absurd contexts, we cannot derive an absurd equality between meta types. This makes the strong reduction sound.

The proof is by *normalizing* equality derivations, *i.e.* maximally reducing (for $\longrightarrow_{\#}$) the terms appearing in the derivation. Then such derivations do not have any type-level $m\mathsf{ML}$ reduction. Once we have such a derivation between two terms with a given $m\mathsf{ML}$ head constructor, we can show by case analysis that the head constructor is preserved throughout the equality derivation. The definition of normalized equality derivations, noted $\Gamma \vdash_{norm} \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$ is given in Figure 7.23 for non-congruence rule and Figure 7.24 for congruence rules. The rules are a subset of the rules for $\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$. For $m\mathsf{ML}$, the congruence rules (equivalent to the rules presented in Figure 7.8 for the full equality) and the transitivity rule (TEqN-Trans) are preserved, while the meta-reduction rules (that were presented in Figure 7.10) are removed. Once the derivation reaches an $e\mathsf{ML}$ type or a term, it proceeds using the usual equality rules (TEqN-eML). We also need to preserve the sub-kinding rules (TEqN-SubSch-R).

Normal equalities are a subset of the usual equality, and are compatible with substitution.

Lemma 7.26 (Normal equalities are equalities)**.** *Suppose* $\Gamma \vdash_{norm} \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$*. Then,* $\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$

Proof. By induction on the derivation: each rule of the normal equality can be translated to the corresponding rule on the usual equality, except TEqN-eML which directly embeds an usual equality. □

Lemma 7.27 (Substitutions for normal equalities)**.** *Suppose* $\Gamma, x : \sigma, \Gamma' \vdash_{norm} \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$*. Then, if* $\Gamma \vdash u_1 : \sigma \simeq u_2 : \sigma$*, we have* $\Gamma, \Gamma'[x \leftarrow u_1] \vdash_{norm} \tau_1[x \leftarrow u_1] : \kappa_1[x \leftarrow u_1] \simeq \tau_2[x \leftarrow u_2] : \kappa_2[x \leftarrow u_2]$*.*

$$\textsc{TEqN-eML}\quad \frac{\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2 \qquad \kappa_1, \kappa_2 \in \{\mathsf{Typ}, \mathsf{Sch}\}}{\Gamma \vdash_{norm} \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2}$$

$$\textsc{TEqN-Trans}\quad \frac{\Gamma \vdash_{norm} \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2 \qquad \Gamma \vdash_{norm} \tau_2 : \kappa_2 \simeq \tau_3 : \kappa_3}{\Gamma \vdash_{norm} \tau_1 : \kappa_1 \simeq \tau_3 : \kappa_3}$$

$$\textsc{TEqN-SubSch-L}\quad \frac{\Gamma \vdash_{norm} \tau : \mathsf{Sch} \simeq \tau' : \kappa}{\Gamma \vdash_{norm} \tau : \mathsf{Met} \simeq \tau' : \kappa} \qquad \textsc{TEqN-SubSch-R}\quad \frac{\Gamma \vdash_{norm} \tau : \kappa \simeq \tau' : \mathsf{Sch}}{\Gamma \vdash_{norm} \tau : \kappa \simeq \tau' : \mathsf{Met}}$$

$$\textsc{TEqN-SubTyp-L}\quad \frac{\Gamma \vdash_{norm} \tau : \mathsf{Typ} \simeq \tau' : \kappa}{\Gamma \vdash_{norm} \tau : \mathsf{Sch} \simeq \tau' : \kappa} \qquad \textsc{TEqN-SubTyp-R}\quad \frac{\Gamma \vdash_{norm} \tau : \kappa \simeq \tau' : \mathsf{Typ}}{\Gamma \vdash_{norm} \tau : \kappa \simeq \tau' : \mathsf{Sch}}$$

Figure 7.23: Normal equalities (except congruence rules)

Similarly, if $\Gamma, \alpha : \kappa, \Gamma' \vdash_{norm} \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$*. Then, if* $\Gamma \vdash_{norm} \sigma_1 : \kappa \simeq \sigma_2 : \kappa$*, we have* $\Gamma, \Gamma'[\alpha \leftarrow \sigma_1] \vdash_{norm} \tau_1[\alpha \leftarrow \sigma_1] : \kappa_1[\alpha \leftarrow \sigma_1] \simeq \tau_2[\alpha \leftarrow \sigma_2] : \kappa_2[\alpha \leftarrow \sigma_2]$.

Proof. By induction on equalities, similarly to substitution for usual equalities. □

We prove that every equality can be turned into a normal equality. This requires some care around function kinds: we not only want to prove that the equality between the two type-level functions can be normalized, but also that for any pair of arguments with a normal equality between them, there is a normal equality between the application of the the left function to the left argument and the right function to the right argument. What we want is a logical relation: we aim to prove that if $\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$, then $(\tau_1, \tau_2) \in [\![\Gamma \vdash \kappa_1]\!]$, and if $(\tau_1, \tau_2) \in [\![\Gamma \vdash \kappa]\!]$, then $\Gamma \vdash_{norm} \|\tau_1\| : \kappa \simeq \|\tau_2\| : \kappa$. The relation $[\![\Gamma \vdash \kappa]\!]$ is defined by induction on the kind-depth of Γ. Its definition is given on Figure 7.26. For the base kinds Typ, Sch and Met, we simply require that the equality derivation between them be normalizable. For arrow kinds, we require that the equality be normalizable, and that, for any pair of equal arguments (according to the relation), the applications of the (equal) types to equal arguments have a normal equality.

It is not immediate that this relation is well-defined: the substitutions in the arrow rules may grow the size of the kind. To make the definition precise, we define the *kind-depth* of a kind $\mathsf{KindDepth}(\kappa)$ inductively as the depth of the kind expression, stopping at types and terms (Figure 7.25 Kind-depth ignores the types and terms present in a kind: it is thus invariant by substitution of type and term variables. This is enough to prove that the logical relation is well-defined.

Lemma 7.28 (Substitution preserves kind-depth)**.** *Consider a kind* κ*, type and term variables* x *and* α*, a non-expansive term* u *and a type* τ*. Then,* $\mathsf{KindDepth}(\kappa) = \mathsf{KindDepth}(\kappa[x \leftarrow u]) = \mathsf{KindDepth}(\kappa[\alpha \leftarrow \tau])$.

TEqN-TVar

$$\frac{(\alpha : \kappa) \in \Gamma}{\Gamma \vdash_{norm} \alpha : \kappa \simeq \alpha : \kappa}$$

TEqN-TyAbs

$$\frac{\Gamma \vdash \kappa_1' \simeq \kappa_2' \qquad \Gamma, \alpha : \kappa_1' \vdash_{norm} \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2}{\Gamma \vdash_{norm} \lambda^{\#}(\alpha : \kappa_1').\, \tau_1 : \Pi(\alpha : \kappa_1')\, \kappa_1 \simeq \lambda^{\#}(\alpha : \kappa_2').\, \tau_2 : \Pi(\alpha : \kappa_2')\, \kappa_2}$$

TEqN-TermAbs

$$\frac{\Gamma \vdash_{norm} \sigma_1 : \kappa_1' \simeq \sigma_2 : \kappa_2' \qquad \Gamma, x : \sigma_1 \vdash_{norm} \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2}{\Gamma \vdash_{norm} \lambda^{\#}(x : \sigma_1).\, \tau_1 : \Pi(x : \sigma_1)\, \kappa_1 \simeq \lambda^{\#}(x : \sigma_2).\, \tau_2 : \Pi(x : \sigma_2)\, \kappa_2}$$

TEqN-TyApp

$$\frac{\Gamma \vdash_{norm} \sigma_1 : \Pi(\alpha : \kappa_1')\, \kappa_1 \simeq \sigma_2 : \Pi(\alpha : \kappa_2')\, \kappa_2 \qquad \Gamma \vdash_{norm} \tau_1 : \kappa_1' \simeq \tau_2 : \kappa_2'}{\Gamma \vdash \sigma_1 \# \tau_1 : \kappa_1[\alpha \leftarrow \tau_1]\sigma_2 \# \tau_2\kappa_2[\alpha \leftarrow \tau_2]}$$

TEqN-TermApp

$$\frac{\Gamma \vdash_{norm} \sigma_1 : \Pi(x : \tau_1)\, \kappa_1 \simeq \sigma_2 : \Pi(x : \tau_2)\, \kappa_2 \qquad \Gamma \vdash u_1 : \tau_1 \simeq u_2 : \tau_2}{\Gamma \vdash \sigma_1 \# u_1 : \kappa_1[x \leftarrow u_1]\sigma_2 \# u_2\kappa_2[x \leftarrow u_2]}$$

TEqN-TyMetaArr

$$\frac{\Gamma \vdash \kappa_1 \simeq \kappa_2 \qquad \Gamma, \alpha : \kappa_1 \vdash_{norm} \tau_1 : \mathsf{Met} \simeq \tau_2 : \mathsf{Met}}{\Gamma \vdash_{norm} \Pi(\alpha : \kappa_1)\, \tau_1 : \mathsf{Met} \simeq \Pi(\alpha : \kappa_2)\, \tau_2 : \mathsf{Met}}$$

TEqN-TermMetaArr

$$\frac{\Gamma \vdash_{norm} \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2 \qquad \Gamma, x : \tau_1 \vdash_{norm} \sigma_1 : \mathsf{Met} \simeq \sigma_2 : \mathsf{Met}}{\Gamma \vdash_{norm} \Pi(x : \tau_1)\, \sigma_1 : \mathsf{Met} \simeq \Pi(x : \tau_2)\, \sigma_2 : \mathsf{Met}}$$

TEqN-EqMetaArr

$$\frac{\begin{array}{c}\Gamma \vdash u_{11} : \sigma_1 \simeq u_{21} : \sigma_2 \\ \Gamma \vdash u_{12} : \sigma_1 \simeq u_{22} : \sigma_2 \qquad \Gamma, (u_{11} \simeq u_{12}) : \sigma_1 \vdash_{norm} \tau_1 : \mathsf{Met} \simeq \tau_2 : \mathsf{Met}\end{array}}{\Gamma \vdash_{norm} \Pi(u_{11} \simeq u_{12}) : \sigma_1\, \tau_1 : \mathsf{Met} \simeq \Pi(u_{21} \simeq u_{22}) : \sigma_2\, \tau_2 : \mathsf{Met}}$$

TEqN-PtrMetaArr

$$\frac{\begin{array}{c}p \perp \Gamma \\ \Gamma \vdash u_1 : \mathsf{bool} \simeq u_2 : \mathsf{bool} \qquad \Gamma, (u_1 \simeq \mathsf{True}) : \mathsf{bool} \vdash_{norm} \tau_1 : \mathsf{Sch} \simeq \tau_2 : \mathsf{Sch} \\ \Gamma, (u_1)*p : \tau_1 \vdash_{norm} \sigma_1 : \mathsf{Met} \simeq \sigma_2 : \mathsf{Met}\end{array}}{\Gamma \vdash_{norm} \Pi((u_1)*p : \tau_1)\, \sigma_1 : \mathsf{Met} \simeq \Pi((u_2)*p : \tau_2)\, \sigma_2 : \mathsf{Met}}$$

TEqN-Thunk

$$\frac{\Gamma \vdash_{norm} \tau_1 : \mathsf{Sch} \simeq \tau_2 : \mathsf{Sch}}{\Gamma \vdash_{norm} [\tau_1] : \mathsf{Met} \simeq [\tau_2] : \mathsf{Met}}$$

Figure 7.24: Normal equalities: congruence rules

$$\mathsf{KindDepth}(\mathsf{Typ}) = \mathsf{KindDepth}(\mathsf{Met}) = \mathsf{KindDepth}(\mathsf{Sch}) = 0$$

$$\mathsf{KindDepth}(\Pi(\alpha : \kappa_1)\ \kappa_2) = 1 + \max(\mathsf{KindDepth}(\kappa_1), \mathsf{KindDepth}(\kappa_2))$$

$$\mathsf{KindDepth}(\Pi(x : \tau)\ \kappa) = 1 + \mathsf{KindDepth}(\kappa)$$

Figure 7.25: Kind depth

$$[\![\Gamma \vdash \kappa \in \{\mathsf{Typ}, \mathsf{Sch}, \mathsf{Met}\}]\!] = \{(\sigma_1, \sigma_2) \mid \Gamma \vdash_{norm} \|\sigma_1\| : \kappa \simeq \|\sigma_2\| : \kappa\}$$

$$[\![\Gamma \vdash \Pi(x : \tau)\ \kappa]\!] = \left\{ (\sigma_1, \sigma_2) \;\middle|\; \begin{array}{ll} & \Gamma \vdash_{norm} \|\sigma_1\| : \Pi(x : \tau)\ \kappa \simeq \|\sigma_2\| : \Pi(x : \tau)\ \kappa \\ \wedge & \forall (u_1, u_2), \Gamma \vdash u_1 : \tau \simeq u_2 : \tau \implies \\ & \qquad (\sigma_1 \# u_1, \sigma_2 \# u_2) \in [\![\Gamma \vdash \kappa[x \leftarrow u_1]]\!] \end{array} \right\}$$

$$[\![\Gamma \vdash \Pi(\alpha : \kappa')\ \kappa]\!] = \left\{ (\sigma_1, \sigma_2) \;\middle|\; \begin{array}{ll} & \Gamma \vdash_{norm} \|\sigma_1\| : \Pi(\alpha : \kappa')\ \kappa \simeq \|\sigma_2\| : \Pi(\alpha : \kappa')\ \kappa \\ \wedge & \forall (\tau_1, \tau_2) \in [\![\Gamma \vdash \kappa']\!], \\ & \qquad (\sigma_1 \# \tau_1, \sigma_2 \# \tau_2) \in [\![\Gamma \vdash \kappa[\alpha \leftarrow \tau_1]]\!] \end{array} \right\}$$

Figure 7.26: Logically normalizable equalities

Proof. Immediate by structural induction on the kind: kind-depth does not take into account any part where substitution can occur. □

This definition is invariant by substitution of equal things:

Lemma 7.29 (Substitution of terms and types)**.**

- *Consider* $\Gamma, x : \tau \vdash \kappa$ *wf. Suppose* $\Gamma \vdash u_1 : \tau \simeq u_2 : \tau$*. Then,* $[\![\Gamma \vdash \kappa[x \leftarrow u_1]]\!] = [\![\Gamma \vdash \kappa[x \leftarrow u_2]]\!]$*.*
- *Consider* $\Gamma, \alpha : \kappa \vdash \kappa'$ *wf. Suppose* $\Gamma \vdash_{norm} \sigma_1 : \kappa \simeq \sigma_2 : \kappa$*. Then,* $[\![\Gamma \vdash \kappa'[\alpha \leftarrow \sigma_1]]\!] = [\![\Gamma \vdash \kappa'[\alpha \leftarrow \sigma_2]]\!]$*.*

Proof. Immediate by induction on the definition of the relation, and using the same results on the normal equality derivations. □

We need symmetry and transitivity for the relation:

Lemma 7.30 (Symmetry for logical normalizability)**.** *Suppose* $(\tau_1, \tau_2) \in [\![\Gamma \vdash \kappa]\!]$*. Then* $(\tau_2, \tau_1) \in [\![\Gamma \vdash \kappa]\!]$*.*

Proof. By induction on kind-depth.

- For base kinds, we need to prove $\Gamma \vdash_{norm} \|\tau_2\| : \kappa \simeq \|\tau_1\| : \kappa$. By hypothesis, $\Gamma \vdash_{norm} \|\tau_1\| : \kappa \simeq \|\tau_2\| : \kappa$. By symmetry of normal equality, we have the result.
- For arrow kinds: we'll examine the case $\kappa = \Pi(\alpha : \kappa_1)\ \kappa_2$. For the base normalized equality, proceed as for base kinds. Consider $(\sigma_1, \sigma_2) \in [\![\Gamma \vdash \kappa_1]\!]$. We need to prove $(\tau_2 \# \sigma_1, \tau_1 \# \sigma_2) \in [\![\Gamma \vdash \kappa_2[\alpha \leftarrow \sigma_1]]\!]$. We have $(\sigma_2, \sigma_1) \in [\![\Gamma \vdash \kappa_1]\!]$ because $[\![\Gamma \vdash \kappa_1]\!]$ is symmetric by induction hypothesis. Thus, $(\tau_1 \# \sigma_2, \tau_2 \# \sigma_1) \in [\![\Gamma \vdash \kappa_2[\alpha \leftarrow \sigma_2]]\!]$. By invariance

by substitution of equal things (Lemma 7.29), $[\![\Gamma \vdash \kappa_2[\alpha \leftarrow \sigma_2]]\!] = [\![\Gamma \vdash \kappa_2[\alpha \leftarrow \sigma_1]]\!]$ Moreover, by induction hypothesis, $[\![\Gamma \vdash \kappa_2[\alpha \leftarrow \sigma_1]]\!]$ is symmetric (it has the same kind-depth as κ_2 which is lower than the kind-depth of κ). Thus, $(\tau_2 \# \sigma_1, \tau_1 \# \sigma_2) \in [\![\Gamma \vdash \kappa_2[\alpha \leftarrow \sigma_1]]\!]$

□

Lemma 7.31 (Logical normalizability is transitive)**.** *Suppose* $(\tau_1, \tau_2) \in [\![\Gamma \vdash \kappa]\!]$ *and* $(\tau_2, \tau_3) \in [\![\Gamma \vdash \kappa]\!]$*, then* $(\tau_1, \tau_3) \in [\![\Gamma \vdash \kappa]\!]$

Proof. By induction on kind-depth:

- For base kinds, we have $\Gamma \vdash_{norm} \|\tau_1\| : \kappa \simeq \|\tau_2\| : \kappa$ and $\Gamma \vdash_{norm} \|\tau_1\| : \kappa \simeq \|\tau_2\| : \kappa$. By TEqN-Trans, $\Gamma \vdash_{norm} \|\tau_1\| : \kappa \simeq \|\tau_3\| : \kappa$.

- For arrow kinds: the base normalized equality is obtained as for base kinds. We'll examine the case $\kappa = \Pi(\alpha : \kappa_1)\ \kappa_2$. Consider $(\sigma_1, \sigma_3) \in [\![\Gamma \vdash \kappa_1]\!]$. We need to prove $(\tau_1 \# \sigma_1, \tau_3 \# \sigma_3) \in [\![\Gamma \vdash \kappa_2[\alpha \leftarrow \sigma_1]]\!]$. We have $(\sigma_1, \sigma_1) \in [\![\Gamma \vdash \kappa_1]\!]$: from $(\sigma_1, \sigma_3) \in [\![\Gamma \vdash \kappa_1]\!]$, by symmetry (Lemma 7.31, we have $(\sigma_3, \sigma_1) \in [\![\Gamma \vdash \kappa_1]\!]$, and we conclude by transitivity (since κ_1 has lower kind-depth than κ). From $(\tau_1, \tau_2) \in [\![\Gamma \vdash \kappa]\!]$, we obtain $(\tau_1 \# \sigma_1, \tau_2 \# \sigma_1) \in [\![\Gamma \vdash \kappa_2[\alpha \leftarrow \sigma_1]]\!]$. We also have $(\tau_2 \# \sigma_1, \tau_3 \# \sigma_3) \in [\![\Gamma \vdash \kappa_2[\alpha \leftarrow \sigma_1]]\!]$. By transitivity of $[\![\Gamma \vdash \kappa_2[\alpha \leftarrow \sigma_1]]\!]$ (induction hypothesis, since $\kappa_2[\alpha \leftarrow \sigma_1]$ has lower kind-depth than κ), $(\tau_1 \# \sigma_1, \tau_3 \# \sigma_3) \in [\![\Gamma \vdash \kappa_2[\alpha \leftarrow \sigma_1]]\!]$.

□

We know want to prove, by induction on equality derivations, that pairs of equal types are included in the relation. We'll need to strengthen our inductive hypothesis to allow some substitutions without changing the derivation we are doing the induction on. We need to introduce a notion of environments mapping one typing environment to another.

Definition 7.8 (Environment mapping)**.** *We say that a pair of environments* (γ_1, γ_2) *map* Γ *to* Γ'*, noted* $\Gamma' \vdash (\gamma_1, \gamma_2) : \Gamma$*, if it has the same domain as* Γ *and:*

- *For all* $(x : \tau) \in \Gamma$*,* $\Gamma' \vdash \gamma_1(x) : \gamma_1(\tau) \simeq \gamma_2(x) : \gamma_2(\tau)$*.*

- *For all* $(\alpha : \kappa) \in \Gamma$*,* $(\gamma_1(\alpha), \gamma_2(\alpha)) \in [\![\Gamma' \vdash \gamma_1(\kappa)]\!]$*.*

- *For all* $\pi \in \Gamma$*,* $\pi \in \Gamma'$*.*

- *For all* $(u)*p : \tau \in \Gamma$*,* $(\gamma_1(u))*p : \gamma_1(\tau) \in \Gamma'$

◇

Then a pair of identity environments, mapping variables to themselves, satisfies this definition. Then we can state a lemma that can be proved by induction on the derivations:

Lemma 7.32 (Equalities are logically normalizable)**.** *Suppose* $\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$*. Moreover, suppose* $\Gamma' \vdash (\gamma_1, \gamma_2) : \Gamma$*. Then,* $(\gamma_1(\tau_1), \gamma_2(\tau_2)) \in [\![\Gamma' \vdash \gamma_1(\kappa_1)]\!]$*.*

Proof. We must (simultaneously) prove a reflexivity result, although it's the same process because any well-formedness derivation for a type can be expanded into an equality derivation between this type and itself. We need a separate result for the induction to be structural on derivations, but the arguments are identical and we omit it. The result is the following: suppose $\Gamma \vdash \tau : \kappa$, and suppose $\Gamma' \vdash (\gamma_1, \gamma_2) : \Gamma$. Then, $(\gamma_1(\tau), \gamma_2(\tau)) \in [\![\Gamma' \vdash \gamma_1(\kappa)]\!]$.

Let us consider the proof of the main result. By induction on the equality derivation:

- Suppose the last rule is an eML congruence or reduction rule, or TEq-Split. Then, the conclusion of the rule has kind Typ or Sch and we can directly apply TEqN-eML.

- Suppose the last rule is TEq-Thunk. Then, the subderivation inside TEq-Thunk has kind Sch, and is immediately normal after applying the rule TEqN-eML.

- Suppose the last rule is TEq-Trans.

$$\text{TEq-Trans}\quad \frac{\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2 \qquad \Gamma \vdash \tau_2 : \kappa_2 \simeq \tau_3 : \kappa_3}{\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_3 : \kappa_3}$$

  Consider the two sub-derivations $\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$ and $\Gamma \vdash \tau_2 : \kappa_2 \simeq \tau_3 : \kappa_3$. By induction hypothesis, $(\gamma_1(\tau_1), \gamma_1(\tau_2)) \in [\![\Gamma \vdash \gamma_1(\kappa_1)]\!]$ and $(\gamma_1(\tau_2), \gamma_2(\tau_3)) \in [\![\Gamma \vdash \gamma_1(\kappa_2)]\!]$. Then $[\![\Gamma \vdash \gamma_1(\kappa_1)]\!] = [\![\Gamma \vdash \gamma_1(\kappa_2)]\!]$. We conclude by Lemma 7.31.

- Suppose the last rule is TEq-Var:

$$\text{TEq-Var}\quad \frac{\vdash \Gamma \qquad \alpha : \mathsf{Typ} \in \Gamma}{\Gamma \vdash \alpha : \mathsf{Typ} \simeq \alpha : \mathsf{Typ}}$$

  Then, by hypothesis on (γ_1, γ_2), $(\gamma_1(\alpha), \gamma_2(\alpha)) \in [\![\Gamma' \vdash \gamma_1(\kappa_1)]\!]$.

- Suppose the last rule is an mML reduction (rules TEq-Red-Meta-*). Then, the two ends of the equality normalize to the same well-formed type. We conclude by the reflexivity result.

- For conversion rules (TEq-SubSch,TEq-SubTyp), observe that Typ, Sch and Met have the same interpretation up to subkinding: we apply the corresponding normal rule for normal derivations.

- Suppose the last rule is a congruence for an mML type constructor (ie. one of the rules TEq-TyMetaArr, TEq-TermMetaArr, TEq-EqMetaArr, TEq-PtrMetaArr, TEq-Thunk). Take for example TEq-TyMetaArr:

$$\text{TEq-TyMetaArr}\quad \frac{\Gamma \vdash \kappa_1 \simeq \kappa_2 \qquad \Gamma, \alpha : \kappa_1 \vdash \tau_1 : \mathsf{Met} \simeq \tau_2 : \mathsf{Met}}{\Gamma \vdash \Pi(\alpha : \kappa_1)\, \tau_1 : \mathsf{Met} \simeq \Pi(\alpha : \kappa_2)\, \tau_2 : \mathsf{Met}}$$

  Then we have $\Gamma \vdash \kappa_1 \simeq \kappa_2$ and $\Gamma, \alpha : \kappa_1 \vdash \tau_1 : \mathsf{Met} \simeq \tau_2 : \mathsf{Met}$. We need to prove, that $(\gamma(\Pi(\alpha : \kappa_1)\, \tau_1), \gamma(\Pi(\alpha : \kappa_2)\, \tau_2)) \in [\![\Gamma' \vdash \gamma(\mathsf{Met})]\!] = [\![\Gamma' \vdash$

$\mathsf{Met}]\!]$, *i.e.* $\Gamma' \vdash_{norm} \Pi(\alpha : \gamma_1(\kappa_1))\|\gamma_1(\tau_1)\| : \mathsf{Met} \simeq \Pi(\alpha : \gamma_2(\kappa_2))\|\gamma_2(\tau_2)\| : \mathsf{Met}$. We intend to apply TEqN-TyMetaArr: we need to prove that $\Gamma' \vdash \gamma_1(\kappa_1) \simeq \gamma_2(\kappa_2)$ (this is true by substitution), and that $\Gamma' \vdash_{norm} \|\gamma_1(\tau_1)\| : \mathsf{Met} \simeq \|\gamma_1(\tau_2)\| : \mathsf{Met}$- Consider $\gamma_1' = \gamma_1[\alpha \leftarrow \alpha']$, $\gamma_2' = \gamma_2[\alpha \leftarrow \alpha']$. Then, we have $\Gamma', \alpha' : \gamma_1(\kappa_1) \vdash (\gamma_1', \gamma_2') : \Gamma, \alpha : \kappa_1$. By induction hypothesis, $(\gamma_1'(\tau_1), \gamma_2'(\tau_2)) \in [\![\Gamma' \vdash \gamma_1'(\mathsf{Met})]\!] = [\![\Gamma' \vdash \mathsf{Met}]\!]$. From this we obtain the desired result.

- Suppose the last rule is a congruence for an application. Consider for example TEq-TyApp:

  TEq-TyApp
  $$\frac{\Gamma \vdash \sigma_1 : \Pi(\alpha : \kappa_1')\,\kappa_1 \simeq \sigma_2 : \Pi(\alpha : \kappa_2')\,\kappa_2 \qquad \Gamma \vdash \tau_1 : \kappa_1' \simeq \tau_2 : \kappa_2'}{\Gamma \vdash \sigma_1 \# \tau_1 : \kappa_1[\alpha \leftarrow \tau_1]\sigma_2 \# \tau_2\kappa_2[\alpha \leftarrow \tau_2]}$$

  Consider $\Gamma' \vdash (\gamma_1, \gamma_2) : \Gamma$: by induction hypothesis, $(\gamma_1(\sigma_1), \gamma_2(\sigma_2)) \in [\![\Gamma' \vdash \Pi(\alpha : \gamma_1(\kappa_1'))\,\gamma_1(\kappa_1)]\!]$, and $(\gamma_1(\tau_1), \gamma_2(\tau_2)) \in [\![\Gamma' \vdash \gamma_1(\kappa_1')]\!]$. Then, by definition of $[\![\Gamma' \vdash \Pi(\alpha : \gamma_1(\kappa_1'))\,\gamma_1(\kappa_1)]\!]$, $(\gamma_1(\sigma_1) \# \gamma_1(\tau_1), \gamma_2(\sigma_2) \# \gamma_2(\tau_2)) \in [\![\Gamf' \vdash \gamma_1(\kappa_1)[\alpha \leftarrow \gamma_1(\tau_1)]]\!]$.

- Suppose the last rule is a congruence for an abstraction. Consider for example TEq-TyAbs:

  TEq-TyAbs
  $$\frac{\Gamma \vdash \kappa_1' \simeq \kappa_2' \qquad \Gamma, \alpha : \kappa_1' \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2}{\Gamma \vdash \lambda^{\#}(\alpha : \kappa_1').\,\tau_1 : \Pi(\alpha : \kappa_1')\,\kappa_1 \simeq \lambda^{\#}(\alpha : \kappa_2').\,\tau_2 : \Pi(\alpha : \kappa_2')\,\kappa_2}$$

  Consider γ such that $\Gamma' \vdash (\gamma_1, \gamma_2) : \Gamma$. We first need to prove that $\Gamma' \vdash_{norm} \|(\|\gamma_1(\lambda^{\#}(\alpha : \kappa_1').\,\tau_1)) : \gamma_1(\Pi(\alpha : \kappa_1')\,\kappa_1) \simeq \|\gamma_2(\lambda^{\#}(\alpha : \kappa_2').\,\tau_2\|) : \gamma_2(\Pi(\alpha : \kappa_2')\,\kappa_2)$. This is very similar to the case of congruence for constructors treated above.

  Now consider $(\sigma_1, \sigma_2) \in [\![\Gamma' \vdash \gamma_1(\kappa_1')]\!]$. We want to show $(\gamma_1(\lambda^{\#}(\alpha : \kappa_1').\,\tau_1) \# \sigma_1, \gamma_2(\lambda^{\#}(\alpha : \kappa_2').\,\tau_1) \# \sigma_2) \in [\![\Gamma' \vdash \gamma_1(\kappa_1[\alpha \leftarrow \sigma_1])]\!]$, This is equivalent to showing that: $(\gamma_1(\tau_1[\alpha \leftarrow \sigma_1]), \gamma_2(\tau_2[\alpha \leftarrow \sigma_2])) \in [\![\Gamf' \vdash \gamma'(\kappa_1[\alpha \leftarrow \sigma_1])]\!]$. Let $\gamma_1' = \gamma_1[\alpha \leftarrow \sigma_1]$, $\gamma_2' = \gamma_2[\alpha \leftarrow \sigma_2]$ be the extension of γ_1, γ_2 with values for α. Then, $\Gamf' \vdash (\gamma_1', \gamma_2') : \Gamma, \alpha : \kappa_1'$, and the previous inclusion can be rewritten as: $(\gamma_1'(\tau_1), \gamma_2'(\tau_2)) \in [\![\Gamf' \vdash \gamma_1'(\kappa_1)]\!]$, true by induction hypothesis.

□

As an immediate corollary, if two terms are equal, there is a normal equality derivation between their normal forms.

We can now prove a projection result very similar to the one for eML (Lemma 6.14).

Lemma 7.33 (Projection)**.** *Consider an evironment* Γ. *and types* $(\tau_i)^i$ *and* $(\tau_i')^i$.

- *If* $\Gamma \vdash [\tau_1] : \mathsf{Met} \simeq [\tau_2] : \mathsf{Met}$.
- *If* $\Gamma \vdash \Pi(x : \tau_1)\sigma_1 : \mathsf{Met} \simeq \Pi(x : \tau_2)\sigma_2 : \mathsf{Met}$, *then* $\Gamma \vdash \tau_1 : \mathsf{Met} \simeq \tau_2 : \mathsf{Met}$ *and* $\Gamma, x : \tau_1 \vdash \sigma_1 : \mathsf{Met} \simeq \sigma_2 : \mathsf{Met}$.

- *If* $\Gamma \vdash \Pi(\alpha : \kappa_1)\, \sigma_1 : \mathsf{Met} \simeq \Pi(\alpha : \kappa_2)\, \sigma_2 : \mathsf{Met}$*, then* $\Gamma \vdash \kappa_1 \simeq \kappa_2 \mathsf{Met}$ *and* $\Gamma, \alpha : \kappa_1 \vdash \sigma_1 : \mathsf{Met} \simeq \sigma_2 : \mathsf{Met}$.
- *If* $\Gamma \vdash \Pi((u_1 \simeq u_1') : \tau_1)\, \sigma_1 : \mathsf{Met} \simeq \Pi((u_2 \simeq u_2') : \tau_2)\, \sigma_2 : \mathsf{Met}$*, then* $\Gamma \vdash \tau_1 : \mathsf{Sch} \simeq \tau_2 : \mathsf{Sch}$*,* $\Gamma \vdash u_1 : \tau_1 \simeq u_2 : \tau_2$*,* $\Gamma \vdash u_1' : \tau_1 \simeq u_2' : \tau_2$*, and* $\Gamma, (u_1 \simeq u_2) : \tau_1 \vdash \sigma_1 : \mathsf{Met} \simeq \sigma_2 : \mathsf{Met}$.
- *If* $\Gamma \vdash \Pi((u_1)*p : \tau_1)\, \sigma_1 : \mathsf{Met} \simeq \Pi((u_2)*p : \tau_2)\, \sigma_2 : \mathsf{Met}$*, then* $\Gamma \vdash u_1 : \mathsf{bool} \simeq u_2 : \mathsf{bool}$*,* $\Gamma, (u_1 \simeq \mathsf{True}) : \mathsf{bool} \vdash \tau_1 : \mathsf{Sch} \simeq \tau_2 : \mathsf{Sch}$*, and* $\Gamma, (u_1)*p : \tau_1 \vdash \sigma_1 : \mathsf{Met} \simeq \sigma_2 : \mathsf{Met}$

Proof. All cases are similar. Let us consider the first one: suppose we have $\Gamma \vdash [\tau_1] : \mathsf{Met} \simeq [\tau_2] : \mathsf{Met}$. Let γ be the identity environment for Γ; we have $\Gamma \vdash (\gamma, \gamma) : \Gamma$, thus by Lemma 7.32, we have $(\|[\tau_1]\|, \|[\tau_2]\|) \in [\![\Gamma \vdash \mathsf{Met}]\!]$. We have $\|[\tau]\| = [\tau]$. Then, by definition of $[\![\Gamma \vdash \mathsf{Met}]\!]$, we have $\Gamma \vdash_{norm} [\|\tau_1\|] : \mathsf{Met} \simeq [\|\tau_2\|] : \mathsf{Met}$.

We will then show that such an equality implies $\Gamma \vdash \|\tau_1\| : \mathsf{Sch} \simeq \|\tau_2\| : \mathsf{Sch}$. This is done by induction on the equality derivation, generalizing to have any kinds instead of Met in the equality, and anything on the right-hand side: suppose $\Gamma \vdash_{norm} [\tau_1] : \kappa_1 \simeq \sigma_2 : \kappa_2$. Then, $\sigma_2 = [\tau_2]$ for some τ_2, and $\Gamma \vdash \tau_1 : \mathsf{Sch} \simeq \tau_2 : \mathsf{Sch}$.

- If the last rule is a congruence rule, it must be TEqN-Thunk. Then, the result we need is an hypothesis of the rule.
- If the last rule is a subkinding rule, apply the induction hypothesis to the premise.
- The last rule cannot be TEqN-eML: this would imply $\Gamma \vdash [\|\tau_1\|] : \mathsf{Sch}$ which is impossible.
- If the last rule is TEqN-Trans,

$$\text{TEqN-Trans}\quad \frac{\Gamma \vdash_{norm} [\tau_1] : \kappa_1 \simeq \sigma_2 : \kappa_2 \qquad \Gamma \vdash_{norm} \sigma_2 : \kappa_2 \simeq \sigma_3 : \kappa_3}{\Gamma \vdash_{norm} [\tau_1] : \kappa_1 \simeq \sigma_3 : \kappa_3}$$

  apply the induction hypothesis to the first premise: we obtain $\sigma_2 = [\tau_2]$, with $\Gamma \vdash \tau_1 : \mathsf{Sch} \simeq \tau_2 : \mathsf{Sch}$. Then, the induction hypothesis can also be applied to the second premise: we obtain $\sigma_3 = [\tau_3]$, with $\Gamma \vdash \tau_2 : \mathsf{Sch} \simeq \tau_3 : \mathsf{Sch}$. Finally, by transitivity (TEq-Trans), we conclude that $\Gamma \vdash \tau_1 : \mathsf{Sch} \simeq \tau_3 : \mathsf{Sch}$. □

This is sufficient to prove subject reduction for eML:

Lemma 7.34 (Subject reduction, mML)**.** *Let* Γ *be an environment, a a term such that* $\Gamma \vdash^{\pi} a : \tau \Rightarrow \Delta$*. Suppose* $a \xrightarrow{l}_{\#} b$*. Then there exists* Δ' *such that* $\Gamma \vdash^{\pi} b : \tau[l] \Rightarrow \Delta'$ *and:*

- *If* $l = 0$*, we have* $\Gamma \vdash \Delta \supseteq \Delta'$.
- *If* $l = q \mapsto u$*, we have* $\Delta' = \Delta[*q \leftarrow u] - q \cup S$*, where* S *is a environment of labels with all labels prefixed by* q.

Moreover, $\Gamma, \Delta' \vdash a[l] : \tau \simeq b : \tau$.

Proof. Identical to the proof for eML, using the projection result. □

7.4 mML elimination

The last result we need to prove about mML is that any mML term that can be typed in an environment without any meta construct normalizes by $\longrightarrow_{\#}$ to an eML term of the same type. It does not suffice to normalize the term, check that it does not contain any mML syntactic construct and conclude by subject reduction: we have to show the existence of an eML typing derivation of the term.

To distinguish mML derivations from eML derivations, we will write eML derivations with a subscript, for example $\Gamma \vdash_{eML} u_1 : \tau_1 \simeq u_2 : \tau_2$ for typing derivations. In this section, we take $\|_\|$ to perform every possible meta-reduction in a term or type.

The proof is similar to the subject reduction proof: we construct a logical relation for terms and types stating that (under the right typing conditions) they reduce to terms that have an eML typing derivation. The logical relation is defined by mutual induction on the syntax of types and kinds.

The interpretation of types as binary relations is given in Figure 7.27. It is written $[\![\Gamma' \vdash (\tau)_{\gamma:\Gamma}]\!]$, where Γ' is the eML environment in which the reduction is performed, τ the type we interpret and γ an environment mapping variables in a mML environment Γ to terms and types scoped in Γ'. γ is useful because, as we descend inside mML abstractions, we accumulate some mML-typed variables in the environment that we need to keep separated (as they must be handled by reduction). For eML types, $[\![\Gamma' \vdash (\tau)_{\gamma:\Gamma}]\!]$ is simply the set of pairs of non-expansive terms of the correct type that can be proved to be equal in eML. For mML constructs, the interpretation is still a set of pairs of non-expansive terms, but since we need to evaluate them, we check that they can be properly applied. Finally, we interpret type-level abstractions and applications as set-theoretic abstractions and applications.

We define in Figure 7.29, given a type τ of kind κ, the set $\langle\!\langle \Gamma' \vdash (\tau : \kappa)_{\gamma:\Gamma} \rangle\!\rangle$ of acceptable interpretations of τ. This is useful because, in type abstractions, it breaks a circular dependency: we cannot abstract on a type and then compute the interpretation of this type, as the type we are given is not a subtype of the type abstraction. Instead, we ask for an acceptable interpretation only. To interpret transitivity, we require the binary relations $\mathcal{R}$ to have the *zig-zag* property, noted $\mathsf{zigzag}(\mathcal{R})$: for all a, b, c, d, if $a\ \mathcal{R}\ b\ \mathcal{R}^{-1}\ c\ \mathcal{R}\ d$, then $a\ \mathcal{R}\ d$. This is a variant of transitivity for heterogenous relations where the left and right-hand side are taken from different sets. For Typ and Sch, we only allow one possible interpretation. We check (Lemma 7.35) that this interpretation has the zig-zag property too.

Similarly, we define in Figure 7.28 an interpretation of kinds as sets of pairs of equal types, noted $[\![\Gamma' \vdash (\kappa)_{\gamma:\Gamma}]\!]$. For the kinds Typ and Sch, we require that the types are equal in eML. For Met, we simply require that they be equal in mML, and functional kinds are interpreted as pairs of type-level functions that map valid pairs of terms or types to valid pairs of types. We also define the set of possible interpretations of kinds $\langle\!\langle \Gamma' \vdash (\kappa)_{\gamma:\Gamma} \rangle\!\rangle$ in Figure 7.30. As for sets of interpretations of types, we require all relations to be between types of the correct kind, and to have the zig-zag property.

Finally, we define in Figure 7.31 the set $[\![\Gamma' \vdash \Gamma]\!]$ of acceptable environments γ representing an mML typing environment Γ in the eML environment Γ'.

$$
\begin{array}{l}
[\![\Gamma' \vdash (\alpha)_{\gamma:\Gamma}]\!] = \gamma_R(\alpha) \\
[\![\Gamma' \vdash (\sigma)_{\gamma:\Gamma}]\!] = \{(u_1, u_2) \mid \Gamma' \vdash_{eML} u_1 : \|\gamma_1(\sigma)\| \simeq u_2 : \|\gamma_2(\sigma)\|\} \\
\qquad\qquad \text{when} \begin{array}{l} \sigma = \tau_1 \to \tau_2 \\ \sigma = \forall(\alpha : \mathsf{Typ})\, \tau \\ \sigma = \zeta\, (\tau_i)^i \\ \sigma = \mathsf{match}\ u\ \mathsf{with}\ (d_j(\tau_k)^k(x_{ji})^i \to \sigma_j)^j \end{array} \\
[\![\Gamma' \vdash (\Pi(x : \tau)\, \sigma)_{\gamma:\Gamma}]\!] = \\
\left\{ \begin{array}{c} \left(\begin{array}{l} \lambda^{\#}(x : \tau_1).\, u_1, \\ \lambda^{\#}(x : \tau_2).\, u_2 \end{array} \right) \end{array} \middle| \begin{array}{ll} & \Gamma' \vdash \tau_1 : \mathsf{Met} \simeq \gamma_1(\tau) : \mathsf{Met} \\ \wedge & \Gamma' \vdash \tau_2 : \mathsf{Met} \simeq \gamma_2(\tau) : \mathsf{Met} \\ \wedge & \forall(u'_1, u'_2) \in [\![\Gamma' \vdash (\tau)_{\gamma:\Gamma}]\!] \\ & \quad (\|u_1[x \leftarrow u'_1]\|, \|u_2[x \leftarrow u'_2]\|) \\ & \quad\quad \in [\![\Gamma' \vdash (\sigma)_{\gamma, x \leftarrow (u'_1, u'_2):\Gamma, x:\tau}]\!] \end{array} \right\} \\
[\![\Gamma' \vdash (\Pi(\alpha : \kappa)\, \sigma)_{\gamma:\Gamma}]\!] = \\
\left\{ \begin{array}{c} \left(\begin{array}{l} \lambda^{\#}(\alpha : \kappa_1).\, u_1, \\ \lambda^{\#}(\alpha : \kappa_2).\, u_2 \end{array} \right) \end{array} \middle| \begin{array}{ll} & \Gamma' \vdash \kappa_1 \simeq \gamma_1(\kappa) \\ \wedge & \Gamma' \vdash \kappa_2 \simeq \gamma_2(\kappa) \\ \wedge & \forall(\tau_1, \tau_2) \in [\![\Gamma' \vdash (\kappa)_{\gamma:\Gamma}]\!] \\ & \forall \mathcal{R} \in \langle\!\langle \Gamma' \vdash (\tau_1 : \kappa)_{\gamma:\Gamma} \rangle\!\rangle \\ & \quad (\|u_1[\alpha \leftarrow \tau_1]\|, \|u_2[\alpha \leftarrow \tau_2]\|) \\ & \quad\quad \in [\![\Gamma' \vdash (\sigma)_{\gamma, \alpha \leftarrow (\tau_1, \tau_2, \mathcal{R}):\Gamma, \alpha:\kappa}]\!] \end{array} \right\} \\
[\![\Gamma' \vdash (\Pi((u_a \simeq u_b) : \tau)\, \sigma)_{\gamma:\Gamma}]\!] = \\
\left\{ \begin{array}{c} \left(\begin{array}{l} \lambda^{\#}(u^1_a \simeq u^1_b) : \tau_1.\, u_1, \\ \lambda^{\#}(u^2_a \simeq u^2_b) : \tau_2.\, u_2 \end{array} \right) \end{array} \middle| \begin{array}{ll} & (\Gamma' \vdash \tau_i : \mathsf{Sch} \simeq \gamma_i(\tau) : \mathsf{Sch})^{i \in \{1,2\}} \\ \wedge & (\Gamma' \vdash u^i_a : \tau_i \simeq \gamma_i(u_a) : \tau_i)^{i \in \{1,2\}} \\ \wedge & (\Gamma' \vdash u^i_b : \tau_i \simeq \gamma_i(u_b) : \tau_i)^{i \in \{1,2\}} \\ \wedge & (u_1, u_2) \\ & \quad \in [\![\Gamma', (\|\gamma_1(u_a)\| \simeq \|\gamma_1(u_b)\|) : \|\gamma_1(\tau)\| \vdash (\sigma)_{\gamma:\Gamma, (u_a \simeq u_b):\tau}]\!] \end{array} \right\} \\
[\![\Gamma' \vdash (\Pi((u')*p : \tau)\, \sigma)_{\gamma:\Gamma}]\!] = \\
\left\{ \begin{array}{c} \left(\begin{array}{l} \lambda^{\#}((u'_1)*p : \tau_1).\, u_1, \\ \lambda^{\#}((u'_2)*p : \tau_2).\, u_2 \end{array} \right) \end{array} \middle| \begin{array}{ll} & (\Gamma' \vdash \tau_i : \mathsf{Typ} \simeq \gamma_i(\tau) : \mathsf{Typ})^{i \in \{1,2\}} \\ \wedge & (\Gamma' \vdash u'_i : \tau_i \simeq \gamma_i(u') : \tau_i)^{i \in \{1,2\}} \\ \wedge & \forall(u'_1, u'_2) \in [\![\Gamma', (\|\gamma_1(u')\| \simeq \mathsf{True}) : \mathsf{bool} \vdash (\tau)_{\gamma:\Gamma, (u' \simeq \mathsf{True}):\mathsf{bool}}]\!] \\ & \quad (\|u_1[*p \leftarrow u'_1]\|, \|u_2[*p \leftarrow u'_2]\|) \\ & \quad\quad \in [\![\Gamma' \vdash (\sigma)_{\gamma, *p \leftarrow (u'_1, u'_2):\Gamma, (u')*p:\tau}]\!] \end{array} \right\} \\
[\![\Gamma' \vdash ([\sigma])_{\gamma:\Gamma}]\!] = \\
\left\{ \left(\begin{array}{l} [\pi.\, a_1], \\ [\pi.\, a_2] \end{array} \right) \middle| \begin{array}{ll} & \mathsf{TermsEqual}(\Gamma' \vdash^{\pi} a_1 : \|\gamma_1(\sigma)\| \simeq a_2 : \|\gamma_2(\sigma)\|) \\ \wedge & (a_i \in \mathcal{U}[\![\Gamma', \pi \vdash (\sigma)_{\gamma_i:\Gamma; p; \Delta}]\!])^{i \in \{1,2\}} \end{array} \right\} \\
[\![\Gamma' \vdash (\lambda^{\#}(x : \tau).\, \sigma)_{\gamma:\Gamma}]\!] = \lambda(u_1, u_2) \in [\![\Gamma' \vdash (\tau)_{.:\gamma}]\!]\ [\![\Gamma' \vdash (\sigma)_{\gamma, x \leftarrow (u_1, u_2):\Gamma, x:\tau}]\!] \\
[\![\Gamma' \vdash (\tau \# u)_{\gamma:\Gamma}]\!] = [\![\Gamma' \vdash (\tau)_{\gamma:\Gamma}]\!]\ (\|\gamma_1(u)\|, \|\gamma_2(u)\|) \\
[\![\Gamma' \vdash (\lambda^{\#}(\alpha : \kappa).\, \sigma)_{\gamma:\Gamma}]\!] = \lambda(\tau_1, \tau_2) \in [\![\Gamma' \vdash (\kappa)_{\gamma:\Gamma}]\!].\ \lambda \mathcal{R} \in \langle\!\langle \Gamma' \vdash (\tau_1 : \kappa)_{\gamma:\Gamma} \rangle\!\rangle.\ [\![\Gamma' \vdash (\sigma)_{\gamma, \alpha \leftarrow (\tau_1, \tau_2, \mathcal{R}):\Gamma, \alpha:\kappa}]\!] \\
[\![\Gamma' \vdash (\sigma \# \tau)_{\gamma:\Gamma}]\!] = [\![\Gamma' \vdash (\sigma)_{\gamma:\Gamma}]\!]\ (\|\gamma_1(\tau)\|, \|\gamma_2(\tau)\|)\ [\![\Gamma' \vdash (\tau)_{\gamma:\Gamma}]\!]
\end{array}
$$

Figure 7.27: Logical relation for mML elimination: type interpretation

$$
\begin{array}{lcl}
[\![\Gamma' \vdash (\mathsf{Typ})_{\gamma:\Gamma}]\!] & = & \{(\tau_1,\tau_2) \mid \Gamma' \vdash_{eML} \tau_1 : \mathsf{Typ} \simeq \tau_2 : \mathsf{Typ}\} \\
[\![\Gamma' \vdash (\mathsf{Sch})_{\gamma:\Gamma}]\!] & = & \{(\tau_1,\tau_2) \mid \Gamma' \vdash_{eML} \tau_1 : \mathsf{Sch} \simeq \tau_2 : \mathsf{Sch}\} \\
[\![\Gamma' \vdash (\mathsf{Met})_{\gamma:\Gamma}]\!] & = & \{(\tau_1,\tau_2) \mid \Gamma' \vdash \tau_1 : \mathsf{Met} \simeq \tau_2 : \mathsf{Met}\}
\end{array}
$$

$$
[\![\Gamma' \vdash (\Pi(x:\tau)\,\kappa)_{\gamma:\Gamma}]\!] =
\left\{ \begin{pmatrix} \lambda^{\#}(x:\tau_1).\,\sigma_1, \\ \lambda^{\#}(x:\tau_2).\,\sigma_2 \end{pmatrix} \;\middle|\; \begin{array}{ll} & \Gamma' \vdash \tau_1 : \mathsf{Met} \simeq \gamma_1(\tau) : \mathsf{Met} \\ \wedge & \Gamma' \vdash \tau_2 : \mathsf{Met} \simeq \gamma_2(\tau) : \mathsf{Met} \\ \wedge & \forall (u_1,u_2) \in [\![\Gamma' \vdash (\tau)_{\gamma:\Gamma}]\!] \\ & \quad (\|\sigma_1[x \leftarrow u_1]\|, \|\sigma_2[x \leftarrow u_2]\|) \\ & \qquad \in [\![\Gamma' \vdash (\kappa)_{\gamma,x\leftarrow(u_1,u_2):\Gamma,x:\tau}]\!] \end{array} \right\}
$$

$$
[\![\Gamma' \vdash (\Pi(\alpha:\kappa)\,\kappa')_{\gamma:\Gamma}]\!] =
\left\{ \begin{pmatrix} \lambda^{\#}(\alpha:\kappa_1).\,\sigma_1, \\ \lambda^{\#}(\alpha:\kappa_2).\,\sigma_2 \end{pmatrix} \;\middle|\; \begin{array}{ll} & \Gamma' \vdash \kappa_1 \simeq \gamma_1(\kappa) \\ \wedge & \Gamma' \vdash \kappa_2 \simeq \gamma_2(\kappa) \\ \wedge & \forall (\tau_1,\tau_2) \in [\![\Gamma' \vdash (\kappa)_{\gamma:\Gamma}]\!] \\ & \forall \mathcal{R} \in \langle\!\langle \Gamma' \vdash (\tau_1:\kappa)_{\gamma:\Gamma} \rangle\!\rangle \\ & \quad (\|u_1[\alpha \leftarrow \tau_1]\|, \|u_2[\alpha \leftarrow \tau_2]\|) \\ & \qquad \in [\![\Gamma' \vdash (\kappa')_{\gamma,\alpha\leftarrow(\tau_1,\tau_2,\mathcal{R}:\Gamma,\alpha:\kappa}]\!] \end{array} \right\}
$$

Figure 7.28: Logical relation for mML elimination: kind interpretation

$$
\begin{array}{l}
\langle\!\langle \Gamma' \vdash (\tau : \mathsf{Typ})_{\gamma:\Gamma} \rangle\!\rangle = \langle\!\langle \Gamma' \vdash (\tau : \mathsf{Sch})_{\gamma:\Gamma} \rangle\!\rangle = \\
\qquad \{\{(u_1,u_2) \mid \Gamma' \vdash_{eML} u_1 : \|\gamma_1(\tau)\| \simeq u_2 : \|\gamma_2(\tau)\|\}\} \\
\langle\!\langle \Gamma' \vdash (\tau : \mathsf{Met})_{\gamma:\Gamma} \rangle\!\rangle = \\
\qquad \{S \mid \mathsf{zigzag}(S) \wedge \forall (u_1,u_2) \in S\; \Gamma' \vdash u_1 : \gamma_1(\tau) \simeq u_2 : \gamma_2(\tau)\} \\
\langle\!\langle \Gamma' \vdash (\tau : \Pi(x:\sigma)\,\kappa)_{\gamma:\Gamma} \rangle\!\rangle = \\
\qquad \left\{ F \;\middle|\; \begin{array}{l} \forall (u_1,u_2) \in [\![\Gamma' \vdash (\sigma)_{\gamma:\Gamma}]\!] \\ F(u_1,u_2) \in \langle\!\langle \Gamma' \vdash (\tau \# x : \kappa)_{\gamma,x\leftarrow(u_1,u_2):\Gamma,x:\sigma} \rangle\!\rangle \end{array} \right\} \\
\langle\!\langle \Gamma' \vdash (\tau : \Pi(\alpha:\kappa')\,\kappa)_{\gamma:\Gamma} \rangle\!\rangle = \\
\qquad \left\{ F \;\middle|\; \begin{array}{l} \forall (\tau_1,\tau_2) \in [\![\Gamma' \vdash (\kappa')_{\gamma:\Gamma}]\!]\; \forall \mathcal{R} \in \langle\!\langle \Gamma' \vdash (\tau_1 : \kappa')_{\gamma:\Gamma} \rangle\!\rangle \\ F(u_1,u_2)(\mathcal{R}) \in \langle\!\langle \Gamma' \vdash (\tau \# \alpha : \kappa)_{\gamma,\alpha\leftarrow(\tau_1,\tau_2,\mathcal{R}):\Gamma,\alpha:\sigma} \rangle\!\rangle \end{array} \right\}
\end{array}
$$

Figure 7.29: Logical relation for mML elimination: constraints on type interpretation

$$
\begin{array}{l}
\langle\!\langle \Gamma' \vdash (\kappa)_{\gamma:\Gamma} \rangle\!\rangle = \\
\qquad \{S \mid \mathsf{zigzag}(S) \wedge \forall (\tau_1,\tau_2) \in S\; \Gamma' \vdash \tau_1 : \gamma_1(\kappa) \simeq \tau_2 : \gamma_2(\kappa)\}
\end{array}
$$

Figure 7.30: Logical relation for mML elimination: constraints on kind interpretation

$$
\begin{array}{lcl}
[\![\Gamma' \vdash \varnothing]\!] & = & \{\varnothing\} \\
[\![\Gamma' \vdash \Gamma, x : \tau]\!] & = & \{\gamma, x \leftarrow (u_1,u_2) \mid \gamma \in [\![\Gamma' \vdash \Gamma]\!] \wedge (u_1,u_2) \in [\![\Gamma' \vdash (\tau)_{\gamma:\Gamma}]\!]\} \\
[\![\Gamma' \vdash \Gamma, \alpha : \kappa]\!] & = & \\
\multicolumn{3}{l}{\qquad \{\gamma, \alpha \leftarrow (\tau_1,\tau_2,\mathcal{R}) \mid \gamma \in [\![\Gamma' \vdash \Gamma]\!] \wedge (\tau_1,\tau_2) \in [\![\Gamma' \vdash (\kappa)_{\gamma:\Gamma}]\!] \wedge \mathcal{R} \in \langle\!\langle \Gamma' \vdash (\tau_1 : \kappa)_{\gamma:\Gamma} \rangle\!\rangle\}} \\
[\![\Gamma' \vdash \Gamma, (u'){*}p : \tau]\!] & = & \\
\multicolumn{3}{l}{\qquad \left\{\gamma, {*}p \leftarrow (u_1,u_2) \;\middle|\; \gamma \in [\![\Gamma' \vdash \Gamma]\!] \wedge (u_1,u_2) \in [\![\Gamma', (u' \simeq \mathsf{True}) : \mathsf{bool} \vdash (\tau)_{\gamma:\Gamma,(u'\simeq\mathsf{True}):\mathsf{bool}}]\!]\right\}} \\
[\![\Gamma' \vdash \Gamma, \pi]\!] & = & \{\gamma, \pi \leftarrow p \mid \gamma \in [\![\Gamma' \vdash \Gamma]\!] \wedge p \perp \Gamma'\} \\
[\![\Gamma' \vdash \Gamma, (u_a \simeq u_b) : \tau]\!] & = & \\
\multicolumn{3}{l}{\qquad \{\gamma \mid \gamma \in [\![\Gamma' \vdash \Gamma]\!] \wedge \Gamma' \vdash_{eML} \|\gamma_1(u_a)\| : \|\gamma_1(\tau)\| \simeq \|\gamma_1(u_b)\| : \|\gamma_1(\tau)\|\}}
\end{array}
$$

Figure 7.31: Logical relation for mML elimination: environments

Lemma 7.35 (The interpretations of Typ and Sch have the zig-zag property)**.** *Consider* $\kappa \in \{\mathsf{Typ}, \mathsf{Sch}\}$*, and suppose* $S \in \langle\!\langle \Gamma' \vdash (\tau : \kappa)_{\gamma:\Gamma} \rangle\!\rangle$*. Then,* $\mathsf{zigzag}(S)$*.*

Proof. We necessarily have:

$$S = \{(u_1, u_2) \mid \Gamma \vdash_{eML} u_1 : \|\gamma_1(\tau)\| \simeq u_2 : \|\gamma_2(\tau)\|\}$$

Suppose $(u_0, u_1) \in S$, $(u_2, u_1) \in S$ and $(u_2, u_3) \in S$. Then, we have $\Gamma \vdash_{eML} u_0 : \|\gamma_1(\tau)\| \simeq u_1 : \|\gamma_2(\tau)\|$, $\Gamma \vdash_{eML} u_2 : \|\gamma_1(\tau)\| \simeq u_1 : \|\gamma_2(\tau)\|$, $\Gamma \vdash_{eML} u_2 : \|\gamma_1(\tau)\| \simeq u_3 : \|\gamma_2(\tau)\|$.

By symmetry and transitivity, $\Gamma \vdash_{eML} u_0 : \|\gamma_1(\tau)\| \simeq u_3 : \|\gamma_2(\tau)\|$. □

We first need to prove substitution properties for our interpretations:

Lemma 7.36 (Substitution)**.** *Suppose* $\gamma \in [\![\Gamma' \vdash \Gamma]\!]$ *and consider* $\Gamma \vdash u : \tau$*. Let* $\gamma' = \gamma, x \leftarrow (\gamma_1(u), \gamma_2(u))$*. Then,*

- $\gamma' \in [\![\Gamma' \vdash \Gamma, x : \tau]\!]$
- $[\![\Gamma' \vdash (\tau)_{\gamma':\Gamma,x:\tau}]\!] = [\![\Gamma' \vdash (\tau[x \leftarrow u])_{\gamma:\Gamma}]\!]$
- $[\![\Gamma' \vdash (\kappa)_{\gamma':\Gamma,x:\tau}]\!] = [\![\Gamma' \vdash (\kappa[x \leftarrow u])_{\gamma:\Gamma}]\!]$
- $\langle\!\langle \Gamma' \vdash (\tau : \kappa)_{\gamma':\Gamma,x:\tau} \rangle\!\rangle = \langle\!\langle \Gamma' \vdash (\tau[x \leftarrow u] : \kappa[x \leftarrow u])_{\gamma:\Gamma} \rangle\!\rangle$
- $\langle\!\langle \Gamma' \vdash (\kappa)_{\gamma':\Gamma,x:\tau} \rangle\!\rangle = \langle\!\langle \Gamma' \vdash (\kappa[x \leftarrow u])_{\gamma:\Gamma} \rangle\!\rangle$

We have the same result with type substitutions, with $\gamma' = \alpha \leftarrow (\gamma_1(\tau), \gamma_2(\tau), [\![\Gamma' \vdash (\tau)_{\gamma:\Gamma}]\!])$*.*

Proof. By induction on the definitions. □

We can then prove all the following by mutual induction:

Lemma 7.37 (Well-typed things are in relation)**.** *Suppose* $\gamma \in [\![\Gamma' \vdash \Gamma]\!]$*. Then,*

(a) Suppose $\Gamma \vdash \tau : \kappa$ *(or* $\Gamma \vdash \tau : \kappa \simeq \tau' : \kappa'$*, or* $\Gamma \vdash \tau' : \kappa' \simeq \tau : \kappa$*). Then,* $[\![\Gamma' \vdash (\tau)_{\gamma:\Gamma}]\!] \in \langle\!\langle \Gamma' \vdash (\tau : \kappa)_{\gamma:\Gamma} \rangle\!\rangle$*.*

(b) Suppose $\Gamma \vdash \kappa$ *wf (or* $\Gamma \vdash \kappa \simeq \kappa'$*, or* $\Gamma \vdash \kappa' \simeq \kappa$*). Then,* $[\![\Gamma' \vdash (\kappa)_{\gamma:\Gamma}]\!] \in \langle\!\langle \Gamma' \vdash (\kappa)_{\gamma:\Gamma} \rangle\!\rangle$*.*

(c) Suppose $\Gamma \vdash u_1 : \tau_1 \simeq u_2 : \tau_2$*. Then,* $(\|\gamma_1(u_1)\|, \|\gamma_2(u_2)\|) \in [\![\Gamma' \vdash (\tau_1)_{\gamma:\Gamma}]\!]$*.*

(d) Suppose $\Gamma \vdash u : \tau$ *(or* $\Gamma \vdash u : \tau \simeq u' : \tau'$*). Then,* $(\|\gamma_1(u)\|, \|\gamma_2(u)\|) \in [\![\Gamma' \vdash (\tau)_{\gamma:\Gamma}]\!]$*.*

(e) Suppose $\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$*. Then,* $(\|\gamma_1(\tau_1)\|, \|\gamma_2(\tau_2)\|) \in [\![\Gamma' \vdash (\kappa_1)_{\gamma:\Gamma}]\!]$*.*

(f) Suppose $\Gamma \vdash \tau : \kappa$*. Then,* $(\|\gamma_1(\tau)\|, \|\gamma_2(\tau)\|) \in [\![\Gamma' \vdash (\kappa)_{\gamma:\Gamma}]\!]$*.*

(g) Suppose $\Gamma \vdash \tau_1 : \kappa_1 \simeq \tau_2 : \kappa_2$*. Then,* $[\![\Gamma' \vdash (\tau_1)_{\gamma:\Gamma}]\!] = [\![\Gamma' \vdash (\tau_2)_{\gamma:\Gamma}]\!]$*, and* $\langle\!\langle \Gamma' \vdash (\tau_1 : \kappa_1)_{\gamma:\Gamma} \rangle\!\rangle = \langle\!\langle \Gamma' \vdash (\tau_2 : \kappa_1)_{\gamma:\Gamma} \rangle\!\rangle$*.*

(h) Suppose $\Gamma \vdash \kappa_1 \simeq \kappa_2$*. Then,* $[\![\Gamma' \vdash (\kappa_1)_{\gamma:\Gamma}]\!] = [\![\Gamma' \vdash (\kappa_2)_{\gamma:\Gamma}]\!]$*, and* $\langle\!\langle \Gamma' \vdash (\tau : \kappa_1)_{\gamma:\Gamma} \rangle\!\rangle = \langle\!\langle \Gamma' \vdash (\tau : \kappa_2)_{\gamma:\Gamma} \rangle\!\rangle$*.*

(i) *Suppose* $\mathsf{TermsEqual}(\Gamma \vdash^{p} a_1 : \tau_1 \simeq a_2 : \tau_2)$, *with* $\Gamma \vdash \tau_1 : \mathsf{Sch}$ *and* $\Gamma \vdash \tau_2 : \mathsf{Sch}$. *Then* $\mathsf{TermsEqual}(\Gamma \vdash^{p_{e\mathsf{ML}}} \|\gamma_1(a_1)\| : \|\gamma_1(\tau_1)\| \simeq \|\gamma_2(a_2)\| : \|\gamma_2(\tau_2)\|)$.

Proof. By induction on the typing/kinding/equality derivations of the judgment after "Suppose".

Consider part (a). We'll only consider the kinding derivation as a premise. We can derive equivalent properties from the equalities (because they imply that their left-hand side has the right kind). Let us consider some representative kinding rules:

- If the last rule is the type variable rule, with $\tau = \alpha$, and $(\alpha : \kappa) \in \Gamma$, then by hypothesis, $\gamma_{\mathcal{R}}(\alpha) \in \langle\!\langle \Gamma' \vdash (\gamma_1(\alpha) : \kappa)_{\gamma':\Gamma'} \rangle\!\rangle$ where Γ'', γ' is a subset of Γ, γ. By weakening and substitution, $\langle\!\langle \Gamma' \vdash (\gamma_1(\alpha) : \kappa)_{\gamma':\Gamma''} \rangle\!\rangle = \langle\!\langle \Gamma' \vdash (\alpha : \kappa)_{\gamma:\Gamma} \rangle\!\rangle$.

- If the last rule is a conversion: we have $\langle\!\langle \Gamma \vdash (\tau : \kappa)_{\gamma:\Gamma} \rangle\!\rangle = \langle\!\langle \Gamma \vdash (\tau : \kappa')_{\gamma:\Gamma} \rangle\!\rangle$ for $\Gamma \vdash \kappa \simeq \kappa'$ by (h).

- If the last rule is an $e\mathsf{ML}$ type formation rule, then the type is of the form $\tau_1 \to \tau_2$, $\forall(\alpha : \mathsf{Typ})\, \tau$, $\zeta\ (\tau_i)^i$, or a pattern matching. Thus it has kind Typ or Sch, and the interpretation matches the single allowed interpretation.

- If the last rule is the formation of an $m\mathsf{ML}$ arrow type, for example the rule K-TermMetaArr:

$$\text{K-TermMetaArr}\quad \frac{\Gamma \vdash \tau_0 : \mathsf{Met} \qquad \Gamma, x : \tau_0 \vdash \tau : \mathsf{Met}}{\Gamma \vdash \Pi(x : \tau_0)\, \tau : \mathsf{Met}}$$

  The interpretation of this kind is:

$$S = \left\{ \begin{pmatrix} \lambda^{\#}(x : \tau_1).\, u_1, \\ \lambda^{\#}(x : \tau_2).\, u_2 \end{pmatrix} \;\middle|\; \begin{array}{ll} & \Gamma' \vdash \tau_1 : \mathsf{Met} \simeq \gamma_1(\tau) : \mathsf{Met} \\ \wedge & \Gamma' \vdash \tau_2 : \mathsf{Met} \simeq \gamma_2(\tau) : \mathsf{Met} \\ \wedge & \forall (u_1', u_2') \in [\![\Gamma' \vdash (\tau)_{\gamma:\Gamma}]\!] \\ & \quad (\|u_1[x \leftarrow u_1']\|, \|u_2[x \leftarrow u_2']\|) \\ & \qquad \in [\![\Gamma' \vdash (\sigma)_{\gamma, x \leftarrow (u_1', u_2'):\Gamma, x:\tau}]\!] \end{array} \right\}$$

  The equality condition is satisfied by using the two equality conditions and by transitivity. Let us prove this satisfies the zig-zag property: suppose $(\lambda^{\#}(x : \tau_0).\, u_0, \lambda^{\#}(x : \tau_1).\, u_1) \in S$, $\lambda^{\#}(x : \tau_2).\, u_2, \lambda^{\#}(x : \tau_1).\, u_1 \in S$ and $\lambda^{\#}(x : \tau_2).u_2, \lambda^{\#}(x : \tau_3).u_3 \in S$, and consider $(u_0', u_3') \in [\![\Gamma' \vdash (\tau)_{\gamma:\Gamma}]\!]$. Let $T = [\![\Gamma' \vdash (\sigma)_{\gamma, x \leftarrow (u_1', u_2'):\Gamma, x:\tau}]\!]$. By induction hypothesis, T has the zig-zag property, and $(\|u_0[x \leftarrow u_0']\|, \|u_1[x \leftarrow u_3']\|), (\|u_2[x \leftarrow u_0']\|, \|u_1[x \leftarrow u_3']\|), (\|u_2[x \leftarrow u_2']\|, \|u_3[x \leftarrow u_3']\|) \in T$. Thus $(\|u_0[x \leftarrow u_0']\|, \|u_3[x \leftarrow u_3']\|) \in T$.

- If the last rule is a type-level abstraction, such as:

$$\text{K-TermAbs}\quad \frac{\Gamma \vdash \tau_0 : \mathsf{Met} \qquad \Gamma, x : \tau_0 \vdash \tau : \kappa}{\Gamma \vdash \lambda^{\#}(x : \tau_0).\, \tau : \Pi(x : \tau_0)\, \kappa}$$

The type is interpreted as:

$$F = \lambda(u_1, u_2) \in [\![\Gamma' \vdash (\tau)_{.:\gamma}]\!]\ [\![\Gamma' \vdash (\sigma)_{\gamma, x \leftarrow (u_1,u_2):\Gamma, x:\tau}]\!]$$

The interpretation of this kind is:

$$\langle\!\langle \Gamma' \vdash (\tau : \Pi(x:\sigma)\ \kappa)_{\gamma:\Gamma} \rangle\!\rangle = \left\{ F \;\middle|\; \begin{array}{l} \forall (u_1, u_2) \in [\![\Gamma' \vdash (\sigma)_{\gamma:\Gamma}]\!] \\ F(u_1, u_2) \in \langle\!\langle \Gamma' \vdash (\tau \# x : \kappa)_{\gamma, x \leftarrow (u_1,u_2):\Gamma, x:\sigma} \rangle\!\rangle \end{array} \right\}$$

Consider $(u_1, u_2) \in [\![\Gamma' \vdash (\sigma)_{\gamma:\Gamma}]\!]$. Then, $\gamma' = \gamma, x \leftarrow (u_1, u_2) \in [\![\Gamma' \vdash \Gamma, x : \sigma]\!]$. Thus, by induction hypothesis,
$F(u_1, u_2) = [\![\Gamma' \vdash (\sigma)_{\gamma, x \leftarrow (u_1,u_2):\Gamma, x:\tau}]\!] \in \langle\!\langle \Gamma' \vdash (\tau \# x : \kappa)_{\gamma, x \leftarrow (u_1,u_2):\Gamma, x:\sigma} \rangle\!\rangle$

- If the last rule is a type-level application, such as:

$$\textsc{K-TermApp}\quad \frac{\Gamma \vdash \tau : \Pi(x:\sigma)\ \kappa \qquad \Gamma \vdash u : \sigma}{\Gamma \vdash \tau \# u : \kappa[x \leftarrow u]}$$

  By induction hypothesis (d), $(\|\gamma_1(u)\|, \|\gamma_2(u)\|) \in [\![\Gamma' \vdash (\sigma)_{\gamma:\Gamma}]\!]$, and by induction hypothesis (a), $[\![\Gamma' \vdash (\tau)_{\gamma:\Gamma}]\!] \in \langle\!\langle \Gamma' \vdash (\tau : \Pi(x:\sigma)\ \kappa)_{\gamma:\Gamma} \rangle\!\rangle$. Then, $[\![\Gamma' \vdash (\tau \# u)_{\gamma:\Gamma}]\!] \in \langle\!\langle \Gamma' \vdash (\tau \# x : \kappa)_{\gamma, x \leftarrow (\|\gamma_1(u)\|, \|\gamma_2(u)\|):\Gamma, x:\sigma} \rangle\!\rangle = \langle\!\langle \Gamma' \vdash (\tau \# u : \kappa[x \leftarrow u])_{\Gamma:\gamma} \rangle\!\rangle$ by substitution.

For part (d), let us consider some representative rules. We will omit part (d), (e) and (f) as they are very similar.

- If the last rule is variable rule, this is immediate by hypothesis on γ.
- If the last rule is a conversion rule: by part (g) the interpretation of the original type and the converted type are equal.
- If the last rule is the transitivity rule:

$$\textsc{Eq-Trans}\quad \frac{\Gamma \vdash u_1 : \tau_1 \simeq u_2 : \tau_2 \qquad \Gamma \vdash u_2 : \tau_2 \simeq u_3 : \tau_3}{\Gamma \vdash u_1 : \tau_1 \simeq u_3 : \tau_3}$$

  By induction hypothesis (g), we have $[\![\Gamma' \vdash (\tau_1)_{\gamma:\Gamma}]\!] = [\![\Gamma' \vdash (\tau_2)_{\gamma:\Gamma}]\!]$. We also have, by (c) and (d):

  - $(\|\gamma_1(u_1)\|, \|\gamma_2(u_2)\|) \in [\![\Gamma' \vdash (\tau_1)_{\gamma:\Gamma}]\!]$
  - $(\|\gamma_1(u_2)\|, \|\gamma_2(u_2)\|) \in [\![\Gamma' \vdash (\tau_2)_{\gamma:\Gamma}]\!]$
  - $(\|\gamma_1(u_2)\|, \|\gamma_2(u_3)\|) \in [\![\Gamma' \vdash (\tau_2)_{\gamma:\Gamma}]\!]$

  Thus, by the zig-zag property, $(\|\gamma_1(u_1)\|, \|\gamma_2(u_3)\|) \in [\![\Gamma' \vdash (\tau_1)_{\gamma:\Gamma}]\!]$.

- If the last rule is an eML congruence or reduction rule, or the split rule (ie. any eML rule that has Sch-kinded output types): we will consider Eq-Let for example.

$$\textsc{Eq-Let}\quad \frac{\begin{array}{cc} \Gamma \vdash \sigma_1 : \mathsf{Sch} & \Gamma, x : \sigma_1 \vdash \tau_1 : \mathsf{Sch} \\ \Gamma \vdash u_1 : \sigma_1 \simeq u_2 : \sigma_2 & \Gamma, x : \sigma_1 \vdash u_1' : \tau_1 \simeq u_2' : \tau_2 \end{array}}{\Gamma \vdash \mathsf{let}\ x = u_1\ \mathsf{in}\ u_1' : \tau_1 \simeq \mathsf{let}\ x = u_2\ \mathsf{in}\ u_2' : \tau_2}$$

By induction hypothesis, $(\|\gamma_1(u_1)\|, \|\gamma_2(u_2)\|) \in [\![\Gamma \vdash (\sigma_1)_{\gamma:\Gamma}]\!] \in \langle\!\langle \Gamma \vdash (\sigma_1 : \mathsf{Sch})_{\gamma:\Gamma} \rangle\!\rangle$. Thus, we have $\Gamma' \vdash_{eML} \|\gamma_1(u_1)\| : \|\gamma_1(\sigma_1)\| \simeq \|\gamma_2(u_2)\| : \|\gamma_2(\sigma_2)\|$ and similarly $\Gamma', x : \|\gamma_1(\sigma_1)\| \vdash_{eML} \|\gamma_1(u_1')\| : \|\gamma_1(\tau_1)\| \simeq \|\gamma_2(u_2')\| : \|\gamma_2(\tau_2)\|$. Then, by Eq-Let, we have $\Gamma' \vdash_{eML} \mathsf{let}\ x = \|\gamma_1(u_1)\|\ \mathsf{in}\ \|\gamma_1(u_1')\| : \|\gamma_1(\tau_1)\| \simeq \mathsf{let}\ x = \|\gamma_2(u_2)\|\ \mathsf{in}\ \|\gamma_2(u_2')\| : \|\gamma_2(\tau_2)\|$. Finally, $\|\gamma_i(\mathsf{let}\ x = u_i\ \mathsf{in}\ u_i')\| = \mathsf{let}\ x = \|\gamma_i(u_i)\|\ \mathsf{in}\ \|\gamma_i(u_i')\|$.

- If the last rule is congruence for an mML abstraction, for example:

  Eq-TermMetaAbs
  $$\frac{\Gamma \vdash \tau_1 : \mathsf{Met} \simeq \tau_2 : \mathsf{Met} \qquad \Gamma, x : \tau_1 \vdash a_1 : \sigma_1 \simeq a_2 : \sigma_2}{\Gamma \vdash \lambda^{\#}(x : \tau_1).\, a_1 : \Pi(x : \tau_1)\, \sigma_1 \simeq \lambda^{\#}(x : \tau_2).\, a_2 : \Pi(x : \tau_2)\, \sigma_2}$$

  The interpretation of the type is:

  $$S = \left\{ \begin{pmatrix} \lambda^{\#}(x : \tau_1').\, u_1, \\ \lambda^{\#}(x : \tau_2').\, u_2 \end{pmatrix} \;\middle|\; \begin{array}{ll} & \Gamma' \vdash \tau_1' : \mathsf{Met} \simeq \gamma_1(\tau_1) : \mathsf{Met} \\ \wedge & \Gamma' \vdash \tau_2' : \mathsf{Met} \simeq \gamma_2(\tau_1) : \mathsf{Met} \\ \wedge & \forall (u_1', u_2') \in [\![\Gamma' \vdash (\tau_1)_{\gamma:\Gamma}]\!] \\ & \quad (\|u_1[x \leftarrow u_1']\|, \|u_2[x \leftarrow u_2']\|) \\ & \qquad \in [\![\Gamma' \vdash (\sigma)_{\gamma, x \leftarrow (u_1', u_2') : \Gamma, x : \tau_1}]\!] \end{array} \right\}$$

  We have $\|\gamma_i(\lambda^{\#}(x : \tau_i).\, a_1)\| = \lambda^{\#}(x : \|\gamma_i(\tau_i)\|).\, u_i$. The typing conditions are satisfied because normalization preserves equality.

  Consider $(u_1', u_2') \in [\![\Gamma' \vdash (\tau_1)_{\gamma:\Gamma}]\!]$. We have $\gamma' = \gamma, x \leftarrow (u_1', u_2') \in [\![\Gamma' \vdash \Gamma, x : \tau_1]\!]$. By induction hypothesis, $(\|\gamma_1'(a_1)\|, \|\gamma_2'(a_2)\|) \in [\![\Gamma' \vdash (\sigma)_{\gamma, x \leftarrow (u_1', u_2') : \Gamma, x : \tau_1}]\!]$. We conclude because $\|\gamma_i'(u_i)\| = \|\|\gamma_i(u_i)\|[x \leftarrow u_i']\| = \|u_i[x \leftarrow u_i']\|$.

- If the last rule is congruence for an mML application, for example:

  Eq-TermMetaApp
  $$\frac{\Gamma \vdash u_1 : \Pi(x : \tau_1')\, \tau_1 \simeq u_2 : \Pi(x : \tau_2')\, \tau_2 \qquad \Gamma \vdash u_1' : \tau_1' \simeq u_2' : \tau_2'}{\Gamma \vdash u_1 \# u_1' : \tau_1[x \leftarrow u_1'] \simeq u_2 \# u_2' : \tau_2[x \leftarrow u_2']}$$

  By induction hypothesis, and unpacking the definition, we learn that $(\|\gamma_1(u_1)\|, \|\gamma_2(u_2)\|)$ is a pair of abstractions. Then, reducing, we obtain a term that is guaranteed to be in the correct interpretation.

For part (i), notice that normalization distributes around term equality: we can normalize all subterms, then remove all thunks using the fact that thunk-typed terms reduce to thunkings, and normalize what was inside the thunks independently.

For part (g), let us consider some representative rules. We will omit part (h) as it is very similar.

- If the last rule is transitivity, we use transitivity of the equality.

- If the last rule is a conversion, use the induction hypothesis directly.

- Consider the case of an equality between eML types (ie. of kind $\kappa \in \{\mathsf{Typ}, \mathsf{Sch}\}$): The interpretations of τ_1 are the sets of pairs of terms (u_1, u_2) such that $\Gamma' \vdash_{eML} u_1 : \|\gamma_1(\tau_1)\| \simeq u_2 : \|\gamma_2(\tau_1)\|$, and we wish to substitute τ_2 for τ_1 there. Consider the first type (the second is similar):

– By (e), we have $(\|\gamma_1(\tau_1)\|, \|\gamma_2(\tau_2)\|) \in [\![\Gamma' \vdash (\kappa)_{\gamma:\Gamma}]\!]$, *i.e.* $\Gamma' \vdash_{eML} \|\gamma_1(\tau_1)\| : \kappa \simeq \|\gamma_2(\tau_2)\| : \kappa$.

– By(f), we have $\Gamma' \vdash_{eML} \|\gamma_1(\tau_2)\| : \kappa \simeq \|\gamma_2(\tau_2)\| : \kappa$.

Thus by symmetry and transitivity, $\Gamma \vdash_{eML} \|\gamma_1(\tau_1)\| : \kappa \simeq \|\gamma_1(\tau_2)\| : \kappa$, and we conclude by conversion.

- Consider a congruence for an mML type constructor: the interpretations of the subexpressions are equal.
- Similarly, congruences for type-level abstraction and reduction are treated simply by using the induction hypothesis to verify that the interpretations of subexpressions are equal.
- Consider a type-level reduction, for example:

$$\textsc{TEq-Reduce-MetaApp}\quad \frac{\Gamma, x : \tau \vdash \sigma : \kappa \qquad \Gamma \vdash u : \tau}{\Gamma \vdash (\lambda^{\#}(x : \tau).\, \sigma) \# u : \kappa[x \leftarrow u] \simeq \sigma[x \leftarrow u] : \kappa[x \leftarrow u]}$$

The left-hand side is interpreted as $(\lambda(u_1, u_2) \in [\![\Gamma' \vdash (\tau)_{.:\gamma}]\!]\ [\![\Gamma' \vdash (\sigma)_{\gamma, x \leftarrow (u_1,u_2):\Gamma, x:\tau}]\!])\ (\|\gamma_1(u)\|, \|\gamma_2(u)\|) = [\![\Gamma' \vdash (\sigma)_{\gamma, x \leftarrow (\|\gamma_1(u)\|, \|\gamma_2(u)\|):\Gamma, x:\tau}]\!]$ while the right-hand side is interpreted as $[\![\Gamma' \vdash (\sigma[x \leftarrow u])_{\gamma:\Gamma}]\!]$, and these two interpretations are equal by substitution.

□

We deduce the main theorem of this section:

Theorem 7.3 (mML elimination)**.** *Let Γ be an eML environment, τ an eML type. Then, if in mML $\Gamma \vdash^p a : \tau \Rightarrow \Delta$, there exists Δ' such that we have $\Gamma \vdash^p_{eML} \|a\| : \tau; \Delta'$.*

Proof. Consider $\Gamma' = \Gamma$, and γ the environment such that:

- $\gamma(x) = (x, x)$
- $\gamma(\alpha) = (\alpha, \alpha, \{(u_1, u_2) \mid \Gamma' \vdash_{eML} u_1 : \alpha \simeq u_2 : \alpha\})$
- $\gamma(\pi) = \pi$
- $\gamma(*p) = *p$

We then have $\gamma \in [\![\Gamma \vdash \Gamma]\!]$ and $\mathsf{TermsEqual}(\Gamma \vdash^p a : \tau \simeq a : \tau)$ By the previous lemma, $\mathsf{TermsEqual}(\Gamma \vdash^p_{\mathsf{eML}} \|a\| : \|\tau\| \simeq \|\gamma\| : \tau)$. This implies that there exists Δ' such that $\Gamma \vdash^p_{eML} \|a\| : \tau; \Delta'$. □

Chapter 8

A logical relation for reasoning on mML

We now define a logical relation for reasoning parametrically on $m\mathsf{ML}$. We want this relation to be compatible with term equality, so that we can reduce the left and right-hand side. The relation needs to handle non-strictly positive types and non-termination. We do this by defining a *step-indexed* logical relation [Appel and McAllester, 2001, Ahmed, 2006], indexed by the number of available execution step (here, we count only expansive steps).

8.1 A deterministic reduction

In this section we introduce another view of $m\mathsf{ML}$: instead of having two stages of reduction, first $\longrightarrow_{\#}$ then $\longrightarrow_{\beta}$, we can define a single reduction $\longrightarrow_{d}$ that extends $\longrightarrow_{\beta}$ and normalizes every term to a value.

This interleaved view of $m\mathsf{ML}$ and $e\mathsf{ML}$ evaluation helps reasoning about $m\mathsf{ML}$: we simply have two different abstractions and applications that work independently, but at the same time. This will help set up the logical relation.

We never need to evaluate types for reduction to proceed to a value. Thus, our reduction will ignore the types appearing in terms (and the terms appearing in these types, *etc.*). The reduction is defined in Figure 8.1. The evaluation contexts $\mathcal{E}$ are the evaluation contexts of $e\mathsf{ML}$, extended with $m\mathsf{ML}$ applications, and the redexes are the ML redexes plus the $m\mathsf{ML}$ redexes, restricted to value arguments. For a term meta-application $u_1 \# u_2$, we require the term u_2 to be fully evaluated to a value v_2, then u_1 to be fully evaluated to v_1. On the other hand, for a result application $u_1 \#^* u_2$, u_2 may be typed in an absurd context, and its reduction might block. For this reason, we do not require (and, indeed, do not allow) reducing the right-hand side of $\#^*$. We instead substitute it as a non-expansive term. The values v are also defined in Figure 8.1.

As for the labeled ML reduction, this reduction is deterministic, and values are irreducible for it.

This reduction coincides with the $e\mathsf{ML}$ reduction on $e\mathsf{ML}$ terms: the contexts and redexes can only match $e\mathsf{ML}$ terms and redexes.

We define an indexed version of this reduction as follows: the beta-reduction and the expansion of fixed points in ML take one step each, and all other reduc-

$$
\begin{array}{rcl}
\mathcal{E} & ::= & \mathsf{let}\ x = [\,]\ \mathsf{in}\ a \mid a^p\ [\,] \mid [\,]^p\ v \mid d\,\overline{\tau}\,(a, ..\, a, [\,], v, ..\, v) \mid \Lambda(\alpha : \mathsf{Typ}).\ [\,] \mid [\,]\ \tau \\
& \mid & \mathsf{match}\ [\,]\ \mathsf{with}\ (P \to a \mid ..\, P \to a) \mid u \# [\,] \mid [\,] \# v \mid [\,] \# \tau \\
& \mid & [\,] \# ((u \simeq u) : \tau) \mid [\,] \#^*\ u \mid ([\,])^p
\end{array}
$$

$$
\begin{array}{rcl}
v & ::= & x \mid d\,\overline{\tau}\ \overline{v} \mid \mathsf{fix}^\pi\ x\ (x : \tau) : \tau\ .\ a \mid \Lambda(\alpha : \kappa).\ v \mid \lambda^\#(\alpha : \kappa).\ u \\
& \mid & \lambda^\#(x : \tau).\ u \mid \lambda^\#(u \simeq u) : \tau.\ u \mid \lambda^\#((u)*p : \tau).\ u \mid [\pi.\ a]
\end{array}
$$

$$
\begin{array}{l}
(\mathsf{fix}^\pi\ x\ (y : \tau_1') : \tau_2'\ .\ a)^p\ v \\
\qquad \xrightarrow{*p \leftarrow \mathsf{reuse}(a[\pi \leftarrow p, x \leftarrow \mathsf{fix}^\pi\ x\ (y : \tau_1') : \tau_2'\ .\ a, y \leftarrow \tau_1'])}_d \\
a[\pi \leftarrow p, x \leftarrow \mathsf{fix}^\pi\ x\ (y : \tau_1') : \tau_2'\ .\ a, y \leftarrow \tau_1']
\end{array}
$$

$$
\begin{array}{rcl}
(\Lambda(\alpha : \mathsf{Typ}).\ v)\ \tau & \xrightarrow{0}_d & v[\alpha \leftarrow \tau] \\
\mathsf{let}\ x = v\ \mathsf{in}\ a & \xrightarrow{0}_d & a[x \leftarrow v] \\
\mathsf{match}\ d_j\ \overline{\tau'}\ (v_i)^i\ \mathsf{with}\ (d_j\,\overline{\tau}\,(x_{ji})^i \to a_j)^j & \xrightarrow{0}_d & a_j[x_{ji} \leftarrow v_i]^i \\
(\lambda^\#(\alpha : \kappa).\ a) \# \tau & \xrightarrow{0}_d & a[\alpha \leftarrow \tau] \\
(\lambda^\#(x : \tau).\ a) \# v & \xrightarrow{0}_d & a[x \leftarrow v] \\
(\lambda^\#(u_1 \simeq u_2) : \tau.\ a) \# (u_1' \simeq u_2') : \tau' & \xrightarrow{0}_d & a \\
(\lambda^\#((u')*p : \tau).\ a) \#^*\ u & \xrightarrow{0}_d & a[*p \leftarrow u] \\
([\pi.\ a])^p & \xrightarrow{*p \leftarrow \mathsf{reuse}(a[\pi \leftarrow p])}_d & a[\pi \leftarrow p]
\end{array}
$$

$$
\textsc{Red-NoLabel}\ \frac{a \xrightarrow{0}_d b}{\mathcal{E}[a] \xrightarrow{0}_d \mathcal{E}[b]}
\qquad
\textsc{Red-Label}\ \frac{a \xrightarrow{*p \leftarrow u}_d b}{\mathcal{E}[a] \xrightarrow{*p \leftarrow u}_d (\mathcal{E}[*p \leftarrow u])[b]}
$$

Figure 8.1: Deterministic reduction $\longrightarrow_d$ for $m\mathsf{ML}$

tions take 0 steps. Then, $\longrightarrow_d^i$ is the reduction of cost i, *i.e.* the composition of i one-step reductions and an arbitrary number of zero-step reductions. Since $\longrightarrow_d 0$ is a subset of the union of $\longrightarrow_\iota$ and $\longrightarrow_\#$, it terminates.

We have the following lemma describing how the counting reduction interacts with term equality:

Lemma 8.1 (Equal terms stay equal after i reductions)**.** *Suppose* $\mathsf{TermsEqual}((\tau_j : \kappa_j)^j \vdash^p a : \tau_a \simeq b : \tau_b)$*, and suppose* $a \longrightarrow_d^i a'$*. Then, there exists* b' *such that* $b \longrightarrow_d^i b'$ *and* $\mathsf{TermsEqual}((\tau_j : \kappa_j)^j \vdash^p a' : \tau_a \simeq b' : \tau_b)$*.*

Proof. By induction on the reduction: if the first reduction is a non-expansive reduction or a thunk reduction, it preserves term equality. If it is the reduction of a function application, by Lemma 6.26, there exists b' such that we can go from b to b' by some non-expansive reductions, then one expansive reduction, preserving equality. □

Also, we have a progress result for $\longrightarrow_d^0$:

Lemma 8.2 ($\longrightarrow_d^0$ reduces non-expansive terms)**.** *Let* $\vdash u : \tau$ *be a well-typed non-expansive term in the empty environment. Then there exists* v *such that* $u \longrightarrow_d^0 v$

Proof. $\longrightarrow_d^0$ terminates. Consider the non-value term in reduction position in u (if there is no such term, u is already a value). It is either a $m\mathsf{ML}$ application or a ML destructor that operates on a value. If it is an $m\mathsf{ML}$ application, we can apply $\longrightarrow_\#$ on it (otherwise the term would be stuck for $\longrightarrow_\#$) and similarly for an ML destructor. □

8.2 Interpretation of kinds

We want to define a typed, binary, step-indexed logical relation. We first define the shape of the interpretation of types depending on their kinds on Figure 8.2 by induction on the kinds. The interpretation is defined in a pair of environments γ_1, γ_2 associating term, type and label variables appearing in κ to closed values and types, and label variables to label variables. For a kind κ, $\mathcal{K}_j[\kappa]_{\gamma_1,\gamma_2}$ is a set of triples of the form $(\sigma_1, (S_k)^{k \leqslant j}, \sigma_2)$, where σ_1 and σ_2 are the possible left and right-projections of a type of kind κ and $(S_k)^{k \leqslant j}$ is a possible sequence of interpretations of the type up to rank j. The left and right projections have kind respectively $\gamma_1(\kappa)$ and $\gamma_2(\kappa)$ in the empty environment.

For types of terms (*i.e.* of kinds Typ, Sch and Met), the interpretation of a type will be a left and a right projection of a type and a relation between values of this type. This relation is step-indexed, *i.e.* it is defined as a sequence of refinements of the largest relation between these types. We also require it to be stable by equality: if two values are equal, they are related to the same values.

We interpret the types of higher-order kinds as sequences of functions. For an abstraction on terms $\Pi(x : \tau)\kappa$, the k-th function in the sequence takes a pair of values of the left and right-hand side projection of the type of the argument $\gamma_1(\tau)$ and $\gamma_2(\tau)$ and returns the interpretation of the application at rank k. We require the interpretation to be stable when replacing the arguments by equal values. The interpretation of an abstractions on type $\Pi(\alpha : \kappa)\,\kappa'$ is similar: the k-th function in the sequence takes a pair of types of kinds $\gamma_1(\kappa)$ and $\gamma_2(\kappa)$ and

$$\mathcal{K}_j[\kappa \in \{\mathsf{Typ}, \mathsf{Sch}, \mathsf{Met}\}]_{\gamma_1,\gamma_2} =$$
$$\left\{ (\sigma_1, (R_k)^{k \leqslant j}, \sigma_2) \;\middle|\; \begin{array}{ll} & (\vdash \sigma_i : \gamma_i(\kappa))^{i \in \{1,2\}} \\ \wedge & \forall k \leqslant j, \forall (v_1, v_2) \in R_k, (\vdash v_i : \sigma_i)^{i \in \{1,2\}} \\ \wedge & \forall \ell \leqslant k \leqslant j, R_\ell \supseteq R_k \\ \wedge & \forall k \leqslant j, \forall (v_1, v_2) \in R_k, \forall (v_1', v_2'), \\ & \quad (\Gamma \vdash v_i : \sigma_i \simeq v_i' : \sigma_i)^{i \in \{1,2\}} \implies (v_1', v_2') \in R_k \end{array} \right\}$$

$$\mathcal{K}_j[\Pi(\alpha : \kappa)\, \kappa']_{\gamma_1,\gamma_2} =$$
$$\left\{ (\sigma_1, (F_k)^{k \leqslant j}, \sigma_2) \;\middle|\; \begin{array}{ll} & (\vdash \sigma_i : \Pi(\alpha : \gamma_i(\kappa))\, \gamma_i(\kappa'))^{i \in \{1,2\}} \\ \wedge & \forall k \leqslant j, \forall (\tau_1, (S_\ell)^{\ell \leqslant k}, \tau_2) \in \mathcal{K}[\kappa]_{\gamma_1,\gamma_2}, \\ & \quad (\sigma_1 \# \tau_1, (F_\ell(\tau_1, (S_h)^{h \leqslant \ell}, \tau_2))^{\ell \leqslant k}, \sigma_2 \# \tau_2) \\ & \quad \in \mathcal{K}_k[\kappa']_{(\gamma_1, \alpha \leftarrow \tau_1), (\gamma_2, \alpha \leftarrow \tau_2)} \\ \wedge & \forall k \leqslant j, \forall (\tau_1, (S_\ell)^{\ell \leqslant k}, \tau_2) \in \mathcal{K}_k[\kappa]_{\gamma_1,\gamma_2}, \\ & \forall (\tau_1', \tau_2'), (\vdash \tau_i : \gamma_i(\kappa) \simeq \tau_i' : \gamma_i(\kappa))^{i \in \{1,2\}} \\ & \quad \implies F_k(\tau_1, (S_\ell)^{\ell \leqslant k}, \tau_2) = F_k(\tau_1', (S_\ell)^{\ell \leqslant k}, \tau_2') \end{array} \right\}$$

$$\mathcal{K}_j[\Pi(x : \tau)\, \kappa]_{\gamma_1,\gamma_2} =$$
$$\left\{ (\sigma_1, (F_k)^{k \leqslant j}, \sigma_2) \;\middle|\; \begin{array}{ll} & (\vdash \sigma_i : \Pi(x : \gamma_i(\tau))\, \gamma_i(\kappa))^{i \in \{1,2\}} \\ \wedge & \forall k \leqslant j, \forall (v_1, v_2), (\vdash v_i : \gamma_i(\tau))^{i \in \{1,2\}} \implies \\ & \quad (\sigma_1 \# v_1, (F_l(v_1, v_2))^{l \leqslant k}, \sigma_2 \# v_2) \\ & \quad \in \mathcal{K}_k[\kappa]_{(\gamma_1, x \leftarrow v_1), (\gamma_2, x \leftarrow v_2)} \\ \wedge & \forall k \leqslant j, \forall (v_1, v_2), \forall (v_1', v_2'), \\ & \quad (\vdash v_i : \gamma_i(\tau) \simeq v_i' : \gamma_i(\tau))^{i \in \{1,2\}} \\ & \quad \implies F_k(v_1, v_2) = F_k(v_1', v_2') \end{array} \right\}$$

Figure 8.2: Interpretation of kinds

a sequence of interpretation at levels $l \leqslant k$, and returns the interpretation at rank k. We require the triple formed by the types and the interpretation to be in $\mathcal{K}_k[\kappa]_{\gamma_1,\gamma_2}$.

We say that two environments γ_1 and γ_2 are equal, noted $\vdash \gamma_1 \simeq \gamma_2 : \Gamma$, if they map type, term, and result variables to equal type, terms and results. Then, equal kinds have equal interpretations in equal environments:

Lemma 8.3 (Equal kinds have equal interpretations in equal environments)**.** *Consider* $\vdash \gamma_1 \simeq \gamma_1' : \Gamma$ *and* $\vdash \gamma_2 \simeq \gamma_2' : \Gamma$*. Then, if* $\Gamma \vdash \kappa \simeq \kappa'$*,* $\mathcal{K}_j[\kappa]_{\gamma_1,\gamma_2} = \mathcal{K}_j[\kappa']_{\gamma_1',\gamma_2'}$*.*

Proof. By induction on equality derivations. The kind equality rules preserve the head constructor. The types and terms in the environment are only used in typing judgments which are stable by substitution. □

Lemma 8.4 (The interpretation of kinds is well-indexed)**.** *Consider a kind* κ*, and environments* γ_1*,* γ_2*. Let* $k \leqslant j$*. Then, if* $(\sigma_1, (S_\ell)^{\ell \leqslant j}, \sigma_2) \in \mathcal{K}_j[\kappa]_{\gamma_1,\gamma_2}$*, then* $(\sigma_1, (S_\ell)^{\ell \leqslant k}, \sigma_2) \in \mathcal{K}_k[\kappa]_{\gamma_1,\gamma_2}$

Proof. By induction on kinds. □

8.3 The logical relation

We define a typed binary step-indexed logical relation on $m\mathsf{ML}$ equipped with $\longrightarrow_d$. We define an interpretation of types of terms as a relation on values $\mathcal{V}_k[\tau]_\gamma$ stable by equality.

The environment γ in this interpretation maps term variables to pairs of terms related at the given type, labels to pairs of expressions representing a value for this label if the expression is true, nothing otherwise, label variables to labels, and type variables α to a triple $(\sigma_1, (S_k)^{k\leqslant j}, \sigma_2)$ where σ_1 and σ_2 are the left and right-projections of the type and the $(S_j)^{k\leqslant j}$ are the interpretations of the a relation between σ_1 and σ_2 at rank k. We note γ_1 (*resp.* γ_2) the left (*resp.* right) projection of γ, *i.e.* the environment mapping variables to their left (*resp.* right) projection in γ. Thus, γ_1, γ_2 is a pair of environments we could use with the interpretation of kinds. If $\gamma(\alpha)$ is $(\sigma_1, (S_k)^{k\leqslant j}, \sigma_2)$, we also write $\gamma_R(\alpha)$ for the sequence of interpretations $(S_k)^{k\leqslant j}$.

The constraints of environments are given in Figure 8.3: an environment γ is in $\mathcal{G}_j[\Gamma]$ if it respects these conditions at step-index j. For type variables $\alpha : \kappa$, we require the triple $(\sigma_1, (S_k)^{k\leqslant j}, \sigma_2)$ to be in the interpretation of the corresponding kind κ. For term variables $x : \tau$, we require the pair of left and right projections v_1, v_2 to be related by the relation at type τ. For equalities, we check that the left and right projections of the terms respect the equality in the empty environment. For results $(u)*p : \tau$, either we require the condition u to reduce to false in the left projection γ_1, or we require that γ maps $*p$ to pairs of related non-expansive terms.

The projection $\mathcal{T}[\sigma]_\gamma$ of a relational type σ in γ maps σ to a pair of types (σ_1, σ_2). We define it here as $(\gamma_1(\sigma), \gamma_2(\sigma))$, but we will extend relational types to embed ornaments as a first-class concept in Chapter 10. It does not depend on the index.

The interpretation $\mathcal{V}_j[\sigma]_\gamma$ of σ in γ at step-index j, is given in Figure 8.4. Types of terms σ are interpreted as sets of values of types σ_1, σ_2 if $\mathcal{T}[\sigma]_\gamma = (\sigma_1, \sigma_2)$ (the typing condition is left implicit to make the definitions lighter), while type-level functions are interpreted as functions transforming interpretations into interpretations. Type-level abstraction maps to abstraction and type-level application to application. When a type variable appears, its interpretations is looked up in the context γ. Related values at function types map related arguments to related results. When the body of the function is non-expansive (in $m\mathsf{ML}$ functions and ML type abstractions), we can simply reduce it to a value in zero steps by $\longrightarrow_d^0$. For thunks and ML functions, we need to check that terms are equal. We delegate this to an interpretation of terms $\mathcal{E}_k[\gamma]_\sigma$ built on the interpretation of types. We test this at an index $k < j$: this is because we performed one expensive step of reduction already by substituting the argument into the body. Finally, for type-level pattern matching, we reduce the term matched upon in zero steps, look at the head-constructor to select the branch, and use the interpretation of this branch with the variables representing the constructor arguments added to the environment.

The interpretation of types as set of terms $\mathcal{E}_j[\gamma]_\sigma$ is defined in Figure 8.4. Two terms (a_1, a_2) are in relation at rank j if they are well-typed and when we can reduce a_2 in $k \leqslant j$ steps to a value, a_1 also reduces in an unspecified number of steps to a value and the values are related at index $j - k$: we use some of the step-index to "pay" for the reduction. This definition requires that

$$
\begin{array}{ll}
\mathcal{G}_j[\varnothing] & = \{\varnothing\} \\
\mathcal{G}_j[\Gamma, x : \tau] & = \{\gamma, x \leftarrow (v_1, v_2) \mid \gamma \in \mathcal{G}_j[\Gamma] \wedge (v_1, v_2) \in \mathcal{V}_j[\gamma]_\tau\} \\
\mathcal{G}_j[\Gamma, \alpha : \kappa] & = \{\gamma, \alpha \leftarrow S \mid \gamma \in \mathcal{G}_j[\Gamma] \wedge S \in \mathcal{K}_j[\kappa]_{\gamma_1,\gamma_2}\} \\
\mathcal{G}_j[\Gamma, (u_a \simeq u_b) : \tau] & = \{\gamma \in \mathcal{G}_k[\Gamma] \mid (\vdash \gamma_i(u_a) : \gamma_i(\tau) \simeq \gamma_i(u_n) : \gamma_i(\tau))^{i \in \{1,2\}}\} \\
\mathcal{G}_j[\Gamma, (u){*}p : \tau] & = \\
\qquad \{\gamma \mid \gamma \in \mathcal{G}_j[\Gamma] \wedge \gamma_1(u) \longrightarrow_d^0 \mathsf{False}\} & \\
\qquad \cup \{\gamma, {*}p \leftarrow (u'_1, u'_2) \mid \gamma \in \mathcal{G}_j[\Gamma] \wedge \gamma_1(u) \longrightarrow_d^0 \mathsf{True} \wedge (u'_1, u'_2) \in \mathcal{E}_j[\tau]_\gamma\} & \\
\mathcal{G}_j[\Gamma, \pi] & = \{\gamma, \pi \leftarrow p \mid \gamma \in \mathcal{G}_j[\Gamma] \wedge p \perp \gamma\}
\end{array}
$$

Figure 8.3: Interpretation of environments

a_1 terminates whenever a_2 terminates, *i.e.* that a_1 terminates more often. In particular, every program of the correct type is related to the never-terminating program. This is not a problem: if we need to consider termination, we can use the reverse relation, where the left and right side are exchanged. This is what we will do on ornaments: we will first show that, for arbitrary patches, the ornamented program is equivalent but terminates less, and then, assuming the patches terminate, we show that the base program and the lifted program are linked by both the normal and the reverse relation.

We need to ensure that this is well-defined. The interpretation is first defined by induction on the index k. For a given k, the interpretation is defined by induction on the type τ, except in the case of datatypes, where we define the relation by induction on the term: the terms inside data constructors are smaller, and their types are either data constructors or function types, and functions types strictly decrease the index. This justification is sufficient to handle ML datatypes, but it would not work if we had (ML) quantified types or mML constructs inside datatypes.

We now prove that, in a well-formed context, the interpretation of a well-kinded type is included in the set of valid interpretations at its kind.

Lemma 8.5 (Type interpretation (as values) is well-defined)**.** *Suppose* $\Gamma \vdash \sigma : \kappa$*, and* $\gamma \in \mathcal{G}_j[\Gamma]$*. If* $(\sigma_1, \sigma_2) = \mathcal{T}[\sigma]_\gamma$*, then* $(\sigma_1, (\mathcal{V}_k[\sigma]_\gamma)^{k \leqslant j}, \sigma_2) \in \mathcal{K}_j[\kappa]_{\gamma_1,\gamma_2}$*.*

Simultaneously, we prove that term interpretation is monotonic and stable by equality (this is sufficient because the types of terms necessarily have kinds Typ, Sch or Met):

Lemma 8.6 (Type interpretation (as terms) is well-defined)**.** *Suppose* $\kappa \in \{\mathsf{Typ}, \mathsf{Sch}, \mathsf{Met}\}$*,* $\Gamma \vdash \sigma : \kappa$ *and* $\gamma \in \mathcal{G}_j[\Gamma]$*. Suppose* $(\sigma_1, \sigma_2) = \mathcal{T}[\sigma]_\gamma$*. Suppose* $(a_1, a_2) \in \mathcal{E}_j[\sigma]_\gamma$*. Consider* a'_1*,* a'_2 *such that* $\mathsf{TermsEqual}(\varnothing \vdash^p a_1 : \sigma_1 \simeq a'_1 : \sigma_1)$ *and* $\mathsf{TermsEqual}(\varnothing \vdash^p a_2 : \sigma_2 \simeq a'_2 : \sigma_2)$*. Then, for all* $k \leqslant j$*,* $(a'_1, a'_2) \in \mathcal{E}_k[\sigma]_\gamma$

Proof. By induction on the well-kindedness judgment.

For the lemma on interpretation as values, the typing constraints are always (implicitly) implied by the definition of $\mathcal{V}_j[\sigma]_\gamma$. We need to prove monotonicity and stability by equality. Consider the last rule:

- If it is K-Var: by hypothesis on γ, $\gamma(\alpha) = (\sigma_1, (S_k)^{k \leqslant j}, \sigma_2) \in \mathcal{K}_j[\kappa]_{\gamma_1,\gamma_2}$.
- For K-SubTyp and K-SubSch: apply the inductive hypothesis, and the same rule on the kinding constraints.

$$\mathcal{E}_j[\tau]_\gamma = \{(a_1, a_2) \mid \forall k \leqslant j,\, \forall v_2,\, a_2 \longrightarrow_d^k v_2 \implies \exists v_1,\, a_1 \longrightarrow_d^* v_1 \wedge (v_1, v_2) \in \mathcal{V}_{j-k}[\tau]_\gamma\}$$

$$\begin{array}{lcl}
\mathcal{V}_j[\alpha]_\gamma & = & \gamma_R(\alpha)_j \\
\mathcal{V}_j[\tau_1 \# \tau_2]_\gamma & = & \mathcal{V}_j[\tau_1]_\gamma\ \mathcal{V}_j[\tau_2]_\gamma \\
\mathcal{V}_j[\tau \# u]_\gamma & = & \mathcal{V}_j[\tau]_\gamma\ (v_1, v_2) \qquad \text{where } (\gamma_i(u) \longrightarrow_d^0 v_i)^{i\in\{1,2\}} \\
\mathcal{V}_j[\lambda^\#(\alpha : \kappa).\, \tau]_\gamma & = & \lambda(\mathcal{R} \in \mathcal{K}_j[\kappa]_{\gamma_1,\gamma_2}).\, \mathcal{V}_j[\tau]_{\gamma,\alpha\leftarrow\mathcal{R}} \\
\mathcal{V}_j[\lambda^\#(x : \kappa).\, \tau]_\gamma & = & \lambda(v_1, v_2).\, \mathcal{V}_j[\tau]_{\gamma,x\leftarrow(v_1,v_2)}
\end{array}$$

$$\mathcal{V}_j[\tau \to \sigma]_\gamma =$$
$$\left\{ \begin{pmatrix} \mathsf{fix}^\pi\, x\, (y : \tau_1) : \sigma_1\ .\, a_1, \\ \mathsf{fix}^\pi\, x\, (y : \tau_2) : \sigma_2\ .\, a_2 \end{pmatrix} \;\middle|\; \begin{array}{l} \forall k < j, \forall p_1, p_2, \forall (v_1, v_2) \in \mathcal{V}_k[\tau]_\gamma, \\ \begin{pmatrix} a_1[x \leftarrow (\mathsf{fix}^\pi\, x\, (y : \tau_1) : \sigma_1\ .\, a_1), y \leftarrow v_1, \pi \leftarrow p_1] \\ a_2[x \leftarrow (\mathsf{fix}^\pi\, x\, (y : \tau_2) : \sigma_2\ .\, a_2), y \leftarrow v_2, \pi \leftarrow p_2] \end{pmatrix} \in \mathcal{E}_k[\sigma]_\gamma \end{array} \right\}$$

$$\mathcal{V}_j[\forall(\alpha : \mathsf{Typ})\, \sigma]_\gamma =$$
$$\left\{ \begin{pmatrix} \Lambda(\alpha : \mathsf{Typ}).\, u_1, \\ \Lambda(\alpha : \mathsf{Typ}).\, u_2 \end{pmatrix} \;\middle|\; \begin{array}{l} \forall k \leqslant j, \forall (\tau_1, S, \tau_2) \in \mathcal{K}_k[\mathsf{Typ}]_{\gamma_1,\gamma_2}, \\ (u_1[\alpha \leftarrow \tau_1], u_2[\alpha \leftarrow \tau_2]) \in \mathcal{E}_k[\sigma]_{\gamma,\alpha\leftarrow(\tau_1,S,\tau_2)} \end{array} \right\}$$

$$\mathcal{V}_j[\zeta\, (\tau_i)^i]_\gamma =$$
$$\left\{ (d\, \overline{\tau_1}\, (v_{1,k})^k, d\, \overline{\tau_2}\, (v_{2,k})^k) \;\middle|\; \begin{array}{ll} & (d : \forall(\alpha_i : \mathsf{Typ})^i\, (\sigma_k)^k \to \zeta\, (\alpha_i)^i) \\ \wedge & \forall k\, (v_{1,k}, v_{2,k}) \in \mathcal{V}_j[\sigma_k[\alpha_i \leftarrow \tau_i]^i]_\gamma \end{array} \right\}$$

$$\mathcal{V}_j[\Pi(x : \tau)\, \sigma]_\gamma =$$
$$\left\{ \begin{pmatrix} \lambda^\#(x : \tau_1).\, u_1, \\ \lambda^\#(x : \tau_2).\, u_2 \end{pmatrix} \;\middle|\; \begin{array}{l} \forall k \leqslant j, \forall (v_1, v_2) \in \mathcal{V}_k[\tau]_\gamma, \\ (u_1[x \leftarrow v_1], u_2[x \leftarrow v_2]) \in \mathcal{E}_k[\sigma]_{\gamma,x\leftarrow(v_1,v_2)} \end{array} \right\}$$

$$\mathcal{V}_j[\Pi((u \simeq u') : \tau)\, \sigma]_\gamma =$$
$$\left\{ \begin{pmatrix} \lambda^\#((u_1 \simeq u_1') : \tau_1).\, u_1'', \\ \lambda^\#((u_2 \simeq u_2') : \tau_2).\, u_2'' \end{pmatrix} \;\middle|\; \begin{array}{l} (\varnothing \vdash \gamma_i(u_i) : \gamma_i(\tau) \simeq \gamma_i(u_i') : \gamma_i(\tau))^{i\in\{1,2\}} \\ \qquad \implies (u_1'', u_2'') \in \mathcal{E}_j[\sigma]_\gamma \end{array} \right\}$$

$$\mathcal{V}_j[\Pi(\alpha : \kappa)\, \sigma]_\gamma =$$
$$\left\{ \begin{pmatrix} \lambda^\#(\alpha : \kappa_1).\, u_1, \\ \lambda^\#(\alpha : \kappa_2).\, u_2 \end{pmatrix} \;\middle|\; \begin{array}{l} \forall k \leqslant j, \forall (\tau_1, S, \tau_2) \in \mathcal{K}_k[\kappa]_{\gamma_1,\gamma_2}, \\ (u_1[\alpha \leftarrow \tau_1], u_2[\alpha \leftarrow \tau_2]) \in \mathcal{E}_k[\sigma]_{\gamma,\alpha\leftarrow(\tau_1,S,\tau_2)} \end{array} \right\}$$

$$\mathcal{V}_j[\Pi((u)*p : \tau)\, \sigma]_\gamma =$$
$$\left\{ \begin{pmatrix} \lambda^\#((u_1)*p : \tau_1).\, u_1', \\ \lambda^\#((u_2)*p : \tau_2).\, u_2' \end{pmatrix} \;\middle|\; \begin{array}{l} (\gamma_1(u) \longrightarrow_d^0 \mathsf{False} \wedge (u_1', u_2') \in \mathcal{E}_k[\sigma]_\gamma) \\ \vee \begin{pmatrix} \gamma_1(u) \longrightarrow_d^0 \mathsf{True} \\ \wedge\ \forall k \leqslant j, \forall (u_1'', u_2'') \in \mathcal{E}_k[\tau]_\gamma, \\ (u_1'[p \leftarrow u_1''], u_2'[p \leftarrow u_2'']) \in \mathcal{V}_k[\sigma]_{\gamma,*p\leftarrow(u_1'',u_2'')} \end{pmatrix} \end{array} \right\}$$

$$\mathcal{V}_j[[\sigma]]_\gamma =$$
$$\left\{ \begin{pmatrix} [\pi.\, a_1], \\ [\pi.\, a_2] \end{pmatrix} \;\middle|\; \forall k < j, \forall p_1, p_2, (a_1[\pi \leftarrow p_1], a_2[\pi \leftarrow p_2]) \in \mathcal{E}_k[\sigma]_\gamma \right\}$$

$$\mathcal{V}_j[\mathsf{match}\ u\ \mathsf{with}\ (d_i(\tau_\ell)^\ell(x_{ik})^k \to \tau_i)^i]_\gamma =$$
$$\begin{cases} \mathcal{V}_j[\tau_k]_{\gamma,(x_{ij}\leftarrow(v_{1,j},v_{2,j}))^j} & \text{if } \gamma_1(u) \longrightarrow_d^0 d_i(v_{1,j})^j \wedge \gamma_2(u) \longrightarrow_d^0 d_i(v_{2,j})^j \\ & \quad \wedge (d_i : \forall(\alpha_\ell : \mathsf{Typ})^\ell\, (\sigma_k)^k \to \zeta\, (\alpha_\ell)^\ell) \\ & \quad \wedge \forall k\, (v_{1,k}, v_{2,k}) \in \mathcal{V}_j[\sigma_k[\alpha_\ell \leftarrow \tau_\ell]^i]_\gamma \\ \varnothing & \text{otherwise} \end{cases}$$

Figure 8.4: Interpretation of types

- For K-Datatype: monotonicity derives from the monotonicity of the interpretations we use for the subvalues. For stability by equality: suppose $(v_1, v_2) \in \mathcal{V}_j[\zeta\,(\tau_i)^i]_\gamma$. Then, we have $v_1 = d(\tau_{1,i})^i(w_{1,k})^k$ and $v_2 = d(\tau_{1,i})^i(w_{2,k})^k$. Suppose $\vdash v_1 : \zeta\,(\tau_{1,i})^i \simeq v_1' : \zeta\,(\tau_{1,i})^i$. and $\vdash v_2 : \zeta\,(\tau_{2,i})^i \simeq v_2' : \zeta\,(\tau_{2,i})^i$. By $m\mathsf{ML}$ elimination (Theorem 7.3), we have $\vdash_{eML} \|v_1\| : \zeta\,(\tau_{1,i})^i \simeq \|v_1'\| : \zeta\,(\tau_{1,i})^i$. We have $\|v_1\| = d(\|\tau_{1,i}\|)^i(\|w_{1,k}\|)^k$, and $\|v_1'\|$ is a value. By the logical relation for $e\mathsf{ML}$ (Lemma 6.22), $v_1'' = \|v_1'\|$ must be of the form $v_1'' = d(\tau_{1,i}'')^i(w_{1,k}'')^k$, with $\vdash_{eML} w_{1,k} : \sigma_{1,k} \simeq w_{1,k}'' : \sigma_{1,k}$ for all k. Since v_1' is a value, it starts with a constructor, so we must have $v_1' = d(\tau_{1,i}')^i(w_{1,k}')^k$, with $w_{1,k}'' = \|w_{1,k}'\|$. Normalization preserves equality. Thus, for all k, $\vdash w_1 : \sigma_{1,k} \simeq w_1' : \sigma_{1,k}$, and similarly for v_2, and we use the stability by equality of their interpretations.

- The other rules for function types follow identical patterns.

- For K-Match, the conditions for the interpretation not to be empty are stronger if the index j is larger, and ensure that γ stays valid. We then use monotonicity and stability by equality of the interpretation of the branch.

- For higher-order constructs: consider K-TyAbs (K-TermAbs is similar). When the function is applied to a valid interpretation, it gives back a valid interpretation by induction hypothesis. Moreover, it is stable by type equality: equal types as inputs translate to equal environments.

- Consider K-TyApp (K-TermApp is similar): by definition of the interpretation of function kinds, the result of the application is valid.

For the lemma on type interpretation as terms, monotonicity comes from monotonicity of the interpretation of values, and stability by equality is implied by Lemma 8.1 and stability of the interpretation of values. □

Lemma 8.7 (Substitution commutes with interpretation)**.** *For all environments γ and indices j, we have:*

$$\begin{array}{lcl} \mathcal{V}_j[\tau[\alpha \leftarrow \tau']]_\gamma & = & \mathcal{V}_j[\tau]_{\gamma[\alpha \leftarrow (\gamma_1(\tau'), (\mathcal{V}_k[\tau']_\gamma)^{k \leqslant j}, \gamma_2(\tau'))]} \\ \mathcal{V}_j[\tau[x \leftarrow u]]_\gamma & = & \mathcal{V}_j[\tau]_{\gamma[x \leftarrow (\gamma_1(u), \gamma_2(u))]} \end{array}$$

Proof. By induction on the structure of the interpretation. □

We want to prove that terms are related to themselves by the relation. For the induction to work properly, we need a finer version of $\mathcal{E}_j[\tau]_\gamma$ that also checks emitted labels: while the current version works to interpret the expansive bodies of functions and thunks, it ignores the emitted labels because they are inaccessible to the surrounding terms. We thus define a set $\mathcal{E}_j[\tau \Rightarrow \Delta]_\gamma$ that takes emitted labels into account: we require

- the emitted labels one each side to be present in Δ or to be suffixes of labels presents in Δ;

- the label in Δ to be emitted on both sides whenever their condition in True, and not emitted otherwise;

- for emitted labels in Δ, the emitted values on both sides to be related at the type given in Δ.

Definition 8.1 (Equivalence, including results)**.** *Consider an index j, a type σ such that $\Gamma \vdash \sigma : \kappa$ with $\kappa \in \{\mathsf{Typ}, \mathsf{Sch}, \mathsf{Met}\}$, and $\gamma \in \mathcal{G}_j[\Gamma]$, and let $(\sigma_1, \sigma_2) = \mathcal{T}[\sigma]_\gamma$.*

Consider (a_1, a_2) such that $\Gamma \vdash^p a_1 : \sigma_1 \Rightarrow \Delta$ and $\Gamma \vdash^p a_2 : \sigma_2 \Rightarrow \Delta$. We say that $(a_1, a_2) \in \mathcal{E}_j[\tau \Rightarrow \Delta]_\gamma$ if the reduction of a_2 by $\longrightarrow_d$ does not terminate, or:

- *There exists i such that $a_2 \xrightarrow{(*p_{2,k} \leftarrow u_{2,k})^k}{}^{i}_{d}\, v_2$.*
- *We have $a_1 \xrightarrow{(*p_{1,k} \leftarrow u_{1,k})^k}{}^{*}_{d}\, v_1$.*
- *The labels in $(p_{1,k})^k$ and $(p_{2,k})^k$ are either in Δ or suffixes of labels in Δ.*
- *The labels in $(p_{1,k})^k$ present in Δ are also the labels in $(p_{2,k})^k$. Define $(p'_l)^\ell$ the subsequence formed from the labels in Δ, associated on the left with the $(u'_{1,\ell})^\ell$ and on the right with the $(u'_{2,\ell})^\ell$.*
- *We define $\gamma' = \gamma, (*p'_\ell \leftarrow (u'_{1,\ell}, u'_{2,\ell}))^\ell$. Then, we must have $\gamma' \in \mathcal{G}_{j-i}[\Gamma, \Delta]$. This implies that every label is present if its condition evaluates to true, and that the non-expansive terms associated to labels are related at the type given in Δ.*
- *Finally, we must have $(v_1, v_2) \in \mathcal{V}_{j-i}[\sigma]_{\gamma'}$*

$\diamondsuit$

We simultaneously need to prove that equal terms have equal interpretations in equivalent environments, as defined here:

Lemma 8.8 (Interpretation is compatible with equality)**.** *Suppose $\Gamma \vdash \tau : \kappa \simeq \tau' : \kappa'$ and that γ and γ' are equivalent,* i.e. *$\vdash \gamma_1 \simeq \gamma'_1 : \Gamma$ and $\vdash \gamma_2 \simeq \gamma'_2 : \Gamma$, and that for all $\alpha : \kappa \in \Gamma$, $\gamma_R(\alpha) = \gamma'_R(\alpha)$. Then, for all j, if $\gamma_1 \in \mathcal{G}_j[\Gamma]$, then $\mathcal{V}_j[\tau]_\gamma = \mathcal{V}_j[\tau']_{\gamma'}$*

We thus prove the following two lemmas by mutual induction:

Theorem 8.1 (Terms are related to themselves)**.** *Suppose $\Gamma \vdash^p a : \tau \Rightarrow \Delta$. Consider k and $\gamma \in \mathcal{G}_k[\Gamma]$. Then, $(\gamma_1(a), \gamma_2(a)) \in \mathcal{E}_k[\tau \Rightarrow \Delta]_\gamma$.*

Proof. By induction on j then on the typing derivation.

For equality:

- If the last rule is TEq-Trans: apply induction hypothesis on the left with environments γ and γ, on the right with environments γ and γ', and conclude by transitivity.
- For TEq-Split:

  TEq-Split
  $$\frac{\begin{array}{c}(\vdash d_i : \forall(\alpha_k : \mathsf{Typ})^k\, (\tau_{ij})^j \to \zeta\, (\alpha_k)^k)^i \\ \Gamma \vdash u : \zeta\, (\tau_k)^k \qquad \Gamma \vdash \sigma_1 : \mathsf{Sch} \qquad \Gamma \vdash \sigma_2 : \mathsf{Sch} \\ (\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, (d_i(\tau_k)^k(x_{ij})^j \simeq u) : \zeta\, (\tau_k)^k \vdash \sigma_1 : \kappa_1 \simeq \sigma_2 : \kappa_2)^i\end{array}}{\Gamma \vdash \sigma_1 : \kappa_1 \simeq \sigma_2 : \kappa_2}$$

By induction, we have $(\gamma_1(u), \gamma_2(u)) \in \mathcal{E}_j[\zeta\,(\tau_k)^k]_\gamma$. $\gamma_2(u)$ is non-expansive and well-typed in the empty environment, thus it reduces in 0 steps to some value v_2. Then, $\gamma_1(u) \longrightarrow_d^0 v_1$ for some v_1, and $(v_1, v_2) \in \mathcal{E}_j[\zeta\,(\tau_k)^k]_\gamma$. $\gamma_2(u)$: there exists a constructor $\vdash d_i : \forall(\alpha_k : \mathsf{Typ})^k\,(\tau_i)^i \to \zeta\,(\alpha_k)^k$ such that $v_1 = d_i(\tau_{1,k})^k(w_{1,i})^i$, $v_2 = d_i(\tau_{2,k})^k(w_{2,i})^i$, and for all i, $(w_{1,i}, w_{2,i}) \in \mathcal{V}_j[\sigma_i]_\gamma$ where σ_i is the type of the i-th parameter.

We have $\varnothing \vdash \gamma_i(u) : \zeta\,\gamma((\tau_k)^k) \simeq \gamma'_i(u) : \zeta\,\gamma'((\tau_k)^k)$ for $i \in \{1, 2\}$. Thus (applying the same reasoning as in the previous lemma), $\gamma'_1(u)$ reduces to a term $d_1(\tau'_{1,k})^k(w'_{1,i})^i$ and similarly for $\gamma'_2(u)$, with $(w_{1,i}, w_{2_i}) \in \mathcal{V}_j[\sigma_i]_{\gamma'}$ because $\mathcal{V}_j[\sigma_i]_\gamma$ is stable by equality and equal to $\mathcal{V}_j[\sigma_i]_{\gamma'}$.

Then consider the branch for d_i, with environments $\gamma'' = \gamma, (x_i \leftarrow (w_{1,i}, w_{2_i}))^i$ and $\gamma''' = \gamma', (x_i \leftarrow (v'_{1,i}, w'_{2,i}))^i$: these environments are valid and compatible and satisfy the newly introduced equality $(d_i(\tau_k)^k(x_{ij})^j \simeq u) : \zeta\,(\tau_k)^k$, so we apply the induction hypothesis in that branch.

- The sub-kinding rules do not change the interpretations.
- If the last rule is TEq-Var, the interpretations are the same.
- If the last rule is congruence for an ML or mML type constructor (*i.e.* one of the rules TEq-Datatype, TEq-Arr, *etc.*), the subtypes and sub-kinds appearing are equal so their interpretations are equal by induction hypothesis, and the interpretation only depends on the interpretations of the subtypes.
- The type-level abstractions and applications respect equality: for congruence, abstractions with equal bodies have equal interpretations, as do applications with equal function and argument. For reduction, this is implied by substitution inside the interpretation.
- For TEq-Match: as for TEq-Split, equal terms will reduce to terms starting with the same constructor and equal parameters, so the new environments stay equal.
- For TEq-ReduceMatch: like the previous case, then the interpretation of the expanded and reduced term are equal by substitution.

For typing, consider the last rule. We will examine a few representative cases:

- If it is Coerce: coercion preserves the interpretations (by induction hypothesis).
- If it is Var: from $\gamma \in \mathcal{G}_k[\Gamma]$, we get $(\gamma_1(x), \gamma_2(x)) \in \mathcal{V}_k[\tau]_\gamma$.
- If it is Reuse:

$$\frac{\begin{array}{c}\textsc{Reuse}\\ ((u)*q : \tau \in \Gamma \wedge \Gamma \vdash u : \mathsf{bool} \simeq \mathsf{True} : \mathsf{bool})\\ \vee (q \perp \Gamma \wedge \Gamma \vdash \mathsf{False} : \mathsf{bool} \simeq \mathsf{True} : \mathsf{bool})\end{array}}{\Gamma \vdash^p *q : \tau \Rightarrow \varnothing}$$

We apply the induction hypothesis on u: we have $(\gamma_1(u), \gamma_2(u)) \in \mathcal{E}_j[\mathsf{bool} \Rightarrow \varnothing]_\gamma$. Since $\gamma_2(u)$ is well-typed in the environment, it reduces in 0 steps to

a value. This value must be True, because the other case implies we have $\vdash \gamma_1(\mathsf{True}) : \mathsf{bool} \simeq \gamma_1(\mathsf{False}) : \mathsf{bool}$, which is impossible.

Thus, by hypothesis on γ, $\gamma(*p) = (u_1, u_2) \in \mathcal{E}_j[\mathsf{bool}]_\gamma$. Since they are non-expansive, their reductions do not emit labels, and we have $(u_1, u_2) \in \mathcal{E}_j[\mathsf{bool} \Rightarrow \varnothing]_\gamma$.

- Consider the Fix rule:

$$\textsc{Fix}\quad \frac{\Gamma \vdash \tau : \mathsf{Typ} \qquad \Gamma \vdash \sigma : \mathsf{Typ} \qquad \Gamma, \pi, x : \tau \to \sigma, y : \tau \vdash^q a : \sigma \Rightarrow \Delta}{\Gamma \vdash^p \mathsf{fix}^\pi\, x\, (y : \tau) : \sigma\ .\ a : \tau \to \sigma \Rightarrow \varnothing}$$

Consider $\gamma \in \mathcal{G}_j[\Gamma]$. We want to prove $(\mathsf{fix}^\pi\, x(y : \gamma_1(\tau)) : \gamma_1(\sigma)\ .\gamma_1(a), \mathsf{fix}^\pi\, x(y : \gamma_2(\tau)) : \gamma_2(\sigma)\ .\ \gamma_2(a)) \in \mathcal{V}_j[\tau \to \sigma]_\gamma$.

Consider $k < j$, and $(v_1, v_2) \in \mathcal{V}_k[\tau]_\gamma$. We need to show:
$(\gamma_1(a)[x \leftarrow \mathsf{fix}^\pi\, x\, (y : \gamma_1(\tau)) : \gamma_1(\sigma)\ .\ \gamma_1(a), y \leftarrow v_1, \pi \leftarrow p_1]$,
$\gamma_2(a)[x \leftarrow \mathsf{fix}^\pi\, x\, (y : \gamma_2(\tau)) : \gamma_2(\sigma)\ .\ \gamma_2(a), y \leftarrow v_2, \pi \leftarrow p_2]) \in \mathcal{V}_k[\sigma]_\gamma$ Note that by weakening, $\gamma \in \mathcal{G}_k[\Gamma]$. Moreover, by induction hypothesis at rank $k < j$, $(\mathsf{fix}^\pi\, x\, (y : \gamma_1(\tau)) : \gamma_1(\sigma)\ .\ \gamma_1(a), \mathsf{fix}^\pi\, x\, (y : \gamma_2(\tau)) : \gamma_2(\sigma)\ .\ \gamma_2(a)) \in \mathcal{V}_k[\tau \to \sigma]_\gamma$. Consider
$\gamma' = \gamma[x \leftarrow (\mathsf{fix}^\pi\, x\, (y : \gamma_1(\tau)) : \gamma_1(\sigma)\ .\ \gamma_1(a),$
$\mathsf{fix}^\pi\, x\, (y : \gamma_2(\tau)) : \gamma_2(\sigma)\ .\ \gamma_2(a)), y \leftarrow (v_1, v_2), , \pi \leftarrow (p_1, p_2)]$. Then, $\gamma' \in \mathcal{G}_k[\Gamma, x : \tau \to \sigma, y : \tau]$. Thus, by induction hypothesis on a, $(\gamma'_1(a), \gamma'_2(a)) = (\gamma_1(a)[x \leftarrow \mathsf{fix}^\pi\, x\, (y : \gamma_1(\tau)) : \gamma_1(\sigma)\ .\ \gamma_1(a), y \leftarrow v_1, \pi \leftarrow p_1]$
$, \gamma_2(a)[x \leftarrow \mathsf{fix}^\pi\, x\, (y : \gamma_2(\tau)) : \gamma_2(\sigma)\ .\ \gamma_2(a), y \leftarrow v_2, \pi \leftarrow p_2]) \in \mathcal{E}_k[\sigma \Rightarrow \Delta]_{\gamma'} \subseteq \mathcal{E}_k[\sigma]_{\gamma'} = \mathcal{E}_k[\sigma]_\gamma$.

- Consider the App rule:

$$\textsc{App}\quad \frac{\begin{array}{c} p \leqslant q \\ q \perp \Gamma, \Delta_a, \Delta_b \qquad \Gamma \vdash^p a : \tau \to \sigma \Rightarrow \Delta_a \qquad \Gamma, \Delta_a \vdash^p b : \tau \Rightarrow \Delta_b \end{array}}{\Gamma \vdash^p a^q\, b : \sigma \Rightarrow \Delta_a, \Delta_b, (\mathsf{True}) * q : \sigma}$$

Let $\gamma \in \mathcal{G}_j[\Gamma]$. Suppose $\gamma_2(a)\ \gamma_2(b) \longrightarrow^k_d v_2$.

Then, there must exist $k_a < k$ such that: $\gamma_2(a) \xrightarrow{l_{a,2}}{}^{k_a}_d v_{a,2}$ where $l_{a,2}$ is a sequence of labels. By induction hypothesis on a, we have $\gamma_1(a) \xrightarrow{l_{a,1}}{}^*_d v_{a,1}$. The labels in $l_{a,1}$ and $l_{a,2}$ are either in Δ_a or strict suffixes of labels in Δ_a. By the typing conditions, only the labels in Δ_a can appear in b: the other labels are neither in Δ_a not orthogonal to it. Consider $\gamma_a \in \mathcal{G}_{k-k_a}[\Gamma, \Delta_a]$ the environment obtained by adding these labels to γ. Then, $\gamma_2(a)\ \gamma_2(b) \xrightarrow{l_{a,2}}{}^k_d v_{a,2}\ \gamma_2(b)[l_{a,2}] = v_{a,2}\ \gamma_{a2}(b)$, and similarly $\gamma_1(a)\ \gamma_1(b) \xrightarrow{l_{a,1}}{}_d v_{a,1}\ \gamma_{a1}(b)$.

There also exists $k_b < k - k_a$ such that: $\gamma_{a2}(b) \xrightarrow{l_{b,2}}{}^{k_b}_d v_{b,2}$. Similarly, by induction hypothesis, we have $\gamma_1(a) \xrightarrow{l_{b,1}}{}^*_d v_{b,1}$, and there exists $\gamma_b \in \mathcal{G}_{k-k_a-k_b}[\Gamma, \Delta_a, \Delta_b]$.

We have: $(v_{a,1}, v_{a,2}) \in \mathcal{V}_{k-k_a-k_b}[\Gamma]_{\gamma_b}$: there exists a_1, a_2 such that $v_{a,1} = \mathsf{fix}^\pi\, x\, (x : \tau_1) : \sigma_1\ .\ a_1$ and $v_{a,2} = \mathsf{fix}^\pi\, x\, (x : \tau_2) : \sigma_2\ .\ a_2$. Then, we have

$$v_{a,2}\ v_{b,2} \xrightarrow{*p \leftarrow \mathsf{reuse}(a_2)[\pi \leftarrow p, x \leftarrow \mathsf{fix}^\pi\, x\, (x:\tau_2):\sigma_2\ .\ a_2, y \leftarrow v_{b,2}]}{}^1_d a_1[\pi \leftarrow p, x \leftarrow, y \leftarrow v_{b,2}]$$

and similarly for $v_{a,1}$ $v_{b,1}$. These two terms are related by $\mathcal{E}_{k-k_a-k_b-1}[\sigma]_{\gamma'}$, and all labels they emit are prefixed by p. Then, when they reduce to values v_1 and v_2 in $k_c \leqslant k - k_a - k_b - 1$ steps, these values are in $\mathcal{V}_{k-k_a-k_b-1-k_c}[\sigma]_{\gamma'}$. Then, their non-expansive versions are related too. Finally,

$$\gamma' = \gamma_b, *p \leftarrow (\mathsf{reuse}(a_1)[\pi \leftarrow p, x \leftarrow, y \leftarrow v_{b,1}], \mathsf{reuse}(a_2)[\pi \leftarrow p, x \leftarrow, y \leftarrow v_{b,2}]) \in \mathcal{G}_{k-k_a-k_b-k_c-1}[\Gamma, \Delta_a, \Delta_b, (\mathsf{True})*p : \sigma]$$

$\square$

Chapter 9

From eML to ML

In this chapter, we'll prove that any eML term well-typed in an ML environment can be transformed into an equivalent ML term, such that the original term and the final terms are equal for the eML equality.

There are three features of eML we need to simplify. First, eML has pattern matching on types, and such types cannot exist in ML. We remove these types by moving the pattern matching to the term level: for example, the term

$$\left(\begin{array}{l}\mathsf{match}\ x\ \mathsf{with}\\ \quad \mathsf{True} \to \lambda(y : \mathsf{unit}).\ \mathsf{True}\\ \quad \mathsf{False} \to \lambda(y : \mathsf{bool}).\ y\end{array}\right)^p \left(\begin{array}{l}\mathsf{match}\ x\ \mathsf{with}\\ \quad \mathsf{True} \to \mathsf{Unit}\\ \quad \mathsf{False} \to x\end{array}\right)$$

contains a type-level pattern matching because the type of both the function and the argument depend on x, but we can move this pattern matching to the term level, around the application, and obtain the following simplified term that only requires ML types:

$$\begin{array}{l}\mathsf{match}\ x\ \mathsf{with}\\ \quad \mathsf{True} \to (\lambda(y : \mathsf{unit}).\ \mathsf{True})^p\ \mathsf{Unit}\\ \quad \mathsf{False} \to (\lambda(y : \mathsf{bool}).\ y)^p\ x\end{array}$$

We also need to get rid of reused results. This is done by lifting the evaluation of the function application producing this result into a separate let binding so we can give it a name. For example, the term $\mathsf{match}\ f^p\ x\ \mathsf{with}\ \mathsf{True} \to *p \mid \mathsf{False} \to \mathsf{True}$ can be rewritten as $\mathsf{let}\ y = f^p\ x\ \mathsf{in}\ \mathsf{match}\ y\ \mathsf{with}\ \mathsf{True} \to y \mid \mathsf{False} \to \mathsf{True}$. Some cases are more complex: consider

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{match}\ y\ \mathsf{with}\ \mathsf{True} \to \mathsf{True} \mid \mathsf{False} \to f^1\ \mathsf{False}\ \mathsf{in}\\ \mathsf{match}\ y\ \mathsf{with}\ \mathsf{True} \to x \mid \mathsf{False} \to *1\end{array}$$

There, the result is only present conditional on y. We again solve this by matching on y first, and having two branches:

$$\begin{array}{l}\mathsf{match}\ y\ \mathsf{with}\\ \quad \mathsf{True} \to \mathsf{let}\ x = \mathsf{True}\ \mathsf{in}\ x\\ \quad \mathsf{False} \to \mathsf{let}\ z = f^1\ \mathsf{False}\ \mathsf{in}\ \mathsf{let}\ x = z\ \mathsf{in}\ z\end{array}$$

Finally, we have conversions in typing derivations: if a type is equal to another for eML, all values of this type are values of the other type. These type

$$
\begin{array}{rcl}
s & ::= & x \mid *p \mid \Lambda(\alpha : \mathsf{Typ}).\, s \mid s\ \nu \mid \mathsf{fix}^p\, x\ (x : \nu) : \nu\ .\ e \mid d\,\overline{\nu}\,\overline{s} \\
\nu & ::= & \alpha \mid \forall(\alpha : \mathsf{Typ})\ \nu \mid \nu \to \nu \mid \zeta\,\overline{\nu} \\
e & ::= & s \mid \mathsf{let}\ x = s^p\ s\ \mathsf{in}\ e \mid \mathsf{match}\ s\ \mathsf{with}\ (P \to e \mid ..\, P \to e)
\end{array}
$$

Figure 9.1: Expanded terms, simple terms and types

equalities can be proved using value equalities deduced from let bindings and pattern matching. We can eliminate the equalities from the context: we make sure that pattern matching is always on a variable. Then, equalities of the form $(x \simeq d(\tau_i)^i(x_j)^j) : \tau$ can be eliminated from the environment by replacing all instances of x by $d(\tau_i)^i(x_j)^j$. Then, all equalities are provable in an environment without equalities. We prove that in this case an equality between ML types is either trivial (the two types are syntactically equal), or there are variables of some un-inhabited types in the contexts.

For technical reasons, we restrict ourselves to environments where all polymorphic definition are definitions of functions. This guarantees that pattern matching only occurs on monomorphic values, which allows us to substitute the variables that are matched on by their values in the branches.

9.1 Expanded terms

The simplification we present here is a maximal version of the transformations described above: pattern matchings and bindings are extruded even if they actually would not need to be. In practice, the implementation performs the maximal simplification then tries to move let bindings and pattern matchings back to their original position.

Our goal is to end up with a term where there is no type-level pattern matching, all function applications are given a name, and all pattern matching is extruded such that we never match on the result of a match of a let. We might match on a variable or a constructor (the latter case would reduce).

Thus we define in Figure 9.1 the syntax of *expanded terms* e. An expanded term is formed by a series of bindings of the results of function applications, pattern matchings, and finally a simple term. The function and argument in application, the things we are matching on, and the terms at the end are required to be *simple terms* s, which are a subset of non-expansive terms: a simple term may be a variable applied to some type arguments, a reused result, a type abstraction whose body is simple, a function definition whose body is expanded or a constructor. Finally all types must be ML types, *i.e.* types that do not feature type-level pattern matching, We note these types ν.

After terms are expanded, we will need to perform a second pass on them to actually eliminate coercions.

9.2 Simplification

The simplification is then described as a small-step reduction $\rightarrowtail$ on terms. The reduction context, defined in Figure 9.2, is the composition of a binding context

$$
\begin{array}{rcl}
F & ::= & [] \mid \mathsf{let}\ x = F\ \mathsf{in}\ a \mid \Lambda(\alpha : \mathsf{Typ}).\ F \mid F\ \tau \mid s\ F_t \mid \mathsf{fix}^p\ x\ (x : \tau) : F_t\ .\ a \\
 & \mid & \mathsf{fix}^p\ x\ (x : F_t) : \nu\ .\ a \mid s^p\ F \mid F^p\ a \mid d\,(\overline{\nu}, F_t, \overline{\tau})\,\overline{a} \\
 & \mid & d\,\overline{\nu}\,(\overline{s}, F, \overline{a}) \mid \mathsf{match}\ F\ \mathsf{with}\ (P \to a \mid ..\ P \to a) \\
F_t & ::= & [] \mid \forall(\alpha : \mathsf{Typ})\ F_t \mid F_t \to \tau \mid \nu \to F_t \mid \zeta\,(\overline{\nu}, F_t, \overline{\tau}) \\
 & \mid & \mathsf{match}\ F\ \mathsf{with}\ (P \to \tau \mid ..\ P \to \tau) \\
B & ::= & [] \mid \mathsf{let}\ x = s^p\ s\ \mathsf{in}\ B \mid \mathsf{match}\ x\ \mathsf{with}\ P \to B \mid \overline{P \to a} \mid F[\mathsf{fix}^p\ x\ (x : \nu) : \nu\ .\ B]
\end{array}
$$

Figure 9.2: Binding contexts, expansion contexts

$$
B[F[\mathsf{let}\ x = s\ \mathsf{in}\ a]] \rightarrowtail B[F[a[x \leftarrow s]]]
$$

$$
\frac{F \neq \mathsf{let}\ _ = []\ \mathsf{in}\ _}{B[F[{s_1}^p\ s_2]] \rightarrowtail B[\mathsf{let}\ x = {s_1}^p\ s_2\ \mathsf{in}\ F[x]]}
$$

$$
\frac{F \neq []}{\begin{array}{c} B[F[\mathsf{match}\ s\ \mathsf{with}\ (d_i(\nu_j)^j(x_{ik})^k \to a_i)^i]] \rightarrowtail \\ B[\mathsf{match}\ s\ \mathsf{with}\ (d_i(\nu_j)^j(x_{ik})^k \to \mathsf{relabel}(F[a_i]))^i] \end{array}}
$$

$$
\begin{array}{c} B[F[\mathsf{match}\ x\ \mathsf{with}\ (d_i(\nu_j)^j(x_{ik})^k \to \sigma_i)^i]] \rightarrowtail \\ B[\mathsf{match}\ x\ \mathsf{with}\ (d_i(\nu_j)^j(x_{ik})^k \to \mathsf{relabel}(F[\sigma_i]))^i] \end{array}
$$

Figure 9.3: Simplification rules

B that binds variables, and is composed of bindings of results of applications and pattern matching, and an expansion context F that does not bind variables and is similar to the evaluation contexts E of the ML reduction, with values replaced by simple terms s. The expansion contexts F can also have their hole in the type part of the terms. Contexts that create a type are written F_t.

The simplification rules are given in Figure 9.3. For a let binding $\mathsf{let}\ x = a\ \mathsf{in}\ b$, once a has been simplified to a simple value s, we substitute it inside the term (as it is non-expansive). When we encounter a function application, we lift it at the end of the binding context. We ask for the expansion context to not be just a let binding to avoid endlessly lifting the same application. When we encounter a pattern matching on a simple term, the pattern matching is similarly added to the end of the binding environment B. This duplicates the term that enclosed the pattern matching. To keep the labeling consistent, we *relabel* all labels produced in F with distinct labels in each branch. This is done by a function $\mathsf{relabel}(a)$ whose definition is given next. This rule can be applied both for type-level and term-level pattern matching.

Definition 9.1 (Relabeling)**.** *Consider a term a introducing labels $(p_i)^i$ (not counting the labels under abstractions, similarly to the decomposition* $\mathsf{atoms}(a)$ *given in Figure 6.10). Then, consider a substitution σ from labels in $(p_i)^i$ to fresh labels. We write $\sigma(a)$ for the term obtained by replacing all $(p_i)^i$ and their uses by the corresponding labels in σ. Then, we note* $\mathsf{relabel}(a)$ *any such relabeled term.* ♢

Lemma 9.1 (Relabeling preserves typing)**.** *Consider a such that $\Gamma \vdash^p a : \tau \Rightarrow \Delta$. Suppose $b = \mathsf{relabel}(a)$ with label substitution σ. Then, $\Gamma \vdash^p b : \sigma(\tau) \Rightarrow$*

$\sigma(\Delta)$.

Proof. By induction on the typing derivation of the term. □

We first prove that we can ignore the binding environment when proving equalities. This is true because the binding environment is always *before* the hole for the order of evaluation, and thus of propagation of results.

Lemma 9.2 (Equality in a binding context)**.** *Consider a binding context B and a term a, with $\Gamma \vdash^p B[a] : \sigma \Rightarrow \Delta$. Suppose the typing derivation uses a typing $\Gamma, \Gamma' \vdash^{p'} a : \tau \Rightarrow \Delta'$ of a, and moreover suppose* $\mathsf{TermsEqual}(\Gamma, \Gamma' \vdash^{p'} a : \tau \simeq a' : \tau)$*. Then,* $\mathsf{TermsEqual}(\Gamma \vdash^p B[a] : \sigma \simeq B[a'] : \sigma)$*.*

Proof. By induction on B:

- Suppose $B = F[\mathsf{fix}^\pi\, x\ (y : \tau_1) : \tau_2\ .\ B'[a]$. We have $\Gamma, \pi, x : \tau_1 \to \tau_2, y : \tau_1 \vdash^\pi B'[a] : \tau_2 \Rightarrow \Delta'$. By induction hypothesis, we have $\mathsf{TermsEqual}(\Gamma, \pi, x : \tau_1 \to \tau_2, y : \tau_1 \vdash^\pi B'[a] : \tau_2 \simeq B'[a'] : \tau_2)$. Thus, by congruence for function definition, $\mathsf{TermsEqual}(\Gamma \vdash^p \mathsf{fix}^\pi\, x\ (y : \tau_1) : \tau_2\ .\ B'[a] : \tau_1 \to \tau_2 \simeq \mathsf{fix}^\pi\, x\ (y : \tau_1) : \tau_2\ .\ B'[a'] : \tau_1 \to \tau_2)$ We conclude by congruence in F: we obtain a non-expansive equality that implies the term equality.

- Suppose $B = \mathsf{let}\ x = {s_1}^q\ s_2\ \mathsf{in}\ B'[a]$. We have $\Gamma, (\mathsf{True})*q : \tau_x, x : \tau_x, (x \simeq *q) : \tau_x \vdash^p B'[a] : \sigma \Rightarrow \Delta''$. By induction hypothesis, we have $\mathsf{TermsEqual}(\Gamma, (\mathsf{True})*q : \tau_x, x : \tau_x, (x \simeq *q) : \tau_x \vdash^p B'[a] : \sigma \simeq B'[a'] : \sigma)$. This means the decompositions (§6.2.5) of $B'[a]$ and $B'[a']$ match. Then the decompositions of $B[a]$ and $B[a']$ match (since they are the same with the indices shifted by one).

- The same reasoning is used when the binding context is a pattern matching.

□

We prove that simplification preserves term equality:

Lemma 9.3 (Expansion preserves term equality)**.** *Consider a well-typed term a with $\Gamma \vdash^p a : \tau \Rightarrow \Delta$ and suppose $a \rightarrowtail a'$. Then,* $\mathsf{TermsEqual}(\Gamma \vdash^p a : \tau \simeq a' : \tau)$*.*

This implies $\Gamma \vdash^p a' : \tau \Rightarrow \Delta'$ for some Δ', i.e. *that expansion preserves types (but not results).*

Proof. The simplification of let bindings is a non-expansive reduction. It preserves term equality by Lemma 6.25.

By Lemma 9.2, we only have to consider reductions with an empty binding context B.

We have three rules to consider. Let us start with lifting of a labeled application into a let binding:

$$F[{s_1}^p\ s_2] \rightarrowtail \mathsf{let}\ x = {s_1}^p\ s_2\ \mathsf{in}\ F[x]$$

Consider the decomposition of $F[{s_1}^p\ s_2]$: the first atom in the decomposition is $*p \leftarrow s_1\ s_2$ with condition True (simple terms are non-expansive so they are

equal to their reusable version). The subsequent atoms are some $(*p_i \leftarrow u_i\ u'_i)^i$ with conditions $u_{c,i}$. The reusable version of the term is $u = \mathsf{reuse}(F[*p])$.

Then, the decomposition of $\mathsf{let}\ x = s_1{}^p\ s_2\ \mathsf{in}\ F[x]$ has first atom $*p \leftarrow s_1\ s_2$ with condition True, the subsequent atoms are $(*p \leftarrow (u_i\ u'_i)[x \leftarrow *p])^i$, with conditions $u_{c,i}[x \leftarrow *p]$. The reusable version of the term is $\mathsf{let}\ x = p\ \mathsf{in}\ \mathsf{reuse}(F[x])$.

Then, all atoms of both terms have equal expressions and conditions: the first atom is the same; for subsequent atoms, we have $(u_i\ u'_i)[x \leftarrow *p] = u_i\ u'_i$, because x does not occur in u_i, and the same for the condition $u_{c,i}$.

Moreover, the reusable versions of the tests are equal by reduction. Then, the terms are equal.

Consider now the lifting of a term-level pattern matching. We will only look at the lifting of terms, the lifting of types is similar. We can consider only the case $B = []$ as well. The rule to consider is then:

$$\begin{array}{l} F[\mathsf{match}\ s\ \mathsf{with}\ (d_i(\nu_j)^j(x_{ik})^k \to a_i)^i] \rightarrowtail \\ \quad \mathsf{match}\ s\ \mathsf{with}\ (d_i(\nu_j)^j(x_{ik})^k \to \mathsf{relabel}(F[a_i]))^i \end{array}$$

We only need to show that $F[\mathsf{match}\ s\ \mathsf{with}\ (d_i(\nu_j)^j(x_{ik})^k \to a_i)^i]$ is equal to $\mathsf{match}\ s\ \mathsf{with}\ (d_i(\nu_j)^j(x_{ik})^k \to \mathsf{relabel}(F[\mathsf{match}\ x\ \mathsf{with}\ (d_i(\nu_j)^j(x_{ik})^k \to a_i)^i]))^i$: the latter term is equal to $\mathsf{match}\ s\ \mathsf{with}\ (d_i(\nu_j)^j(x_{ik})^k \to \mathsf{relabel}(F[\mathsf{match}\ d_i(\nu_j)^j(x_{ik})^k\ \mathsf{with}\ (d_i(\nu_j)^j(x_{ik})^k \to a_i)^i]))^i$ by substitution using the equalities in each branch of the outer match, and this term reduces (preserving equality) to the term on the right-hand side of the reduction.

We suppose the relabeling transforms labels $(p_j)^j$ into labels $(p_{ij})^j$ in the branch of the constructor i. Consider the decomposition of the left-hand side: it is $(*p_j \leftarrow u_j)^j$, with conditions $u_{c,j}$ and types τ_j. Consider the decomposition of the right-hand side. It is $(*p_{ij} \leftarrow \mathsf{match}\ s\ \mathsf{with}\ d_i(\nu_j)^j(x_{ik})^k \to u_j\ \mathsf{match}\ s\ \mathsf{with}\ d_i(\nu_j)^j(x_{ik})^k \to u'_j)^{ij}$, with conditions $\mathsf{match}\ s\ \mathsf{with}\ d_i(\nu_j)^j(x_{ik})^k \to u_{c,j} \mid (d_\ell(\nu_j)^j(x_{\ell k})^k \to \mathsf{False})^{\ell \neq i}$ and types $\mathsf{match}\ s\ \mathsf{with}\ d_i(\nu_j)^j(x_{ik})^k \to \tau_j$.

We want to show that the indexings of the atoms of these two terms match: consider an index m. We want to prove: $\Gamma \vdash \textsc{Index}_m(u_{c,j} \to \mathsf{App}(u_j, u'_j))^j : \textsc{Index}_m(u_{c,j} \to \mathsf{app}\ (\tau_j, \tau'_j))^j \simeq \textsc{Index}_m((\mathsf{match}\ x\ \mathsf{with}\ d_i(\nu_j)^j(x_{ik})^k \to u_{c,j} \mid _ \to \mathsf{False}) \to \mathsf{App}(\mathsf{match}\ s\ \mathsf{with}\ d_i(\nu_j)^j(x_{ik})^k \to u_j, \mathsf{match}\ s\ \mathsf{with}\ d_i(\nu_j)^j(x_{ik})^k \to u'_j))^{\ell j} : \textsc{Index}_m((\mathsf{match}\ s\ \mathsf{with}\ d_i(\nu_j)^j(x_{ik})^k \to u_{c,j} \mid _ \to \mathsf{False}) \to \mathsf{app}\ (\tau_j, \tau'_j))^{\ell j}$. Let us start by splitting on s: we need to prove this equality in $\Gamma, (x_{ik} : \tau_{ik})^{ik}, (s \simeq d_i(\nu_j)^j(x_{ik})^k) : \zeta\ (\nu_j)^j$ for each i. We apply this substitution everywhere and reduce: the items on the right-hand side become:

- For $\ell = i$, the conditions are $u_{c,j}$, the terms are $\mathsf{App}(u_j, u'_j)$ and the types are $\mathsf{app}\ (\tau_j, \tau'_j)$
- For $\ell \neq i$, the conditions are False.

Then, the conditions for $p_{\ell j}$ on the right-hand side are False for $\ell \neq i$. We can reduce the corresponding matches in $\textsc{Index}_m(\ldots)$: this yield exactly an $\textsc{Index}_m(\ldots)$ where these options have been removed. We thus obtain the equality.

We also have to prove that the reusable versions of the terms are equal. Again, we first split on x and substitute and reduce on the right-hand side:

we have to prove $\Gamma, \Delta, \Delta', ((\textsc{Index}_m(u_{c,j} \to p_j)^j \simeq \textsc{Index}_m(u_{c,j} \to p_{ij})^j) : \textsc{Index}_\ell(u_{c,j} \to \tau_j)^j)^m, (x_{ik} : \tau_{ik})^k \vdash F[a_i] : \tau \simeq F[a_i][*p_j \leftarrow *p_{ij}]^j : \tau$.

This is true as long as for all j $\Gamma, \Delta, \Delta', ((\textsc{Index}_m(u_{c,j} \to p_j)^j \simeq \textsc{Index}_m(u_{c,j} \to p_{ij})^j) : \textsc{Index}_\ell(u_{c,j} \to \tau_j)^j)^m, (u_{c,j} \simeq \mathsf{True}) : \mathsf{bool} \vdash *p_{ij} : \tau'_j \simeq *p_j : \tau'_j$, *i.e.* the labels match between the left and right-hand side. We proceed by splitting on all the $u_{c,j}$: then, there exists at least one index m such that the m-th index matches p_j. It also matches p_{ij} since the conditions are equal. Otherwise, $u_{c,j}$ is false and we derive the equality from this contradiction.

Then the two terms are equal since their decompositions are equal. □

We now have to prove that the simplification terminates and yields and expanded term:

Lemma 9.4 (Simplification terminates)**.** *There is not infinite sequence of terms* $(a_i)^i$ *such that* $a_i \rightarrowtail a_{i+1}$

Proof. Consider a reduction in a bind context B. Then, there is no rule that allows reduction in a prefix of this bind context, thus bind contexts always grow.

We define the weight of a term a as the number of let bindings, function applications, and pattern matchings it contains (not counting inside functions), and the set of weights of a term as the multiset of weights of residuals, *i.e.* terms that are in the holes of maximal B contexts.

All reductions take a residual of weight n and transform it into multiple residuals of weight $n-1$. Then consider the sequence $(w_n)^{n \in \mathbf{N}}$ counting the terms of weight n. Each transformation leads to a lower value of this sequence for the lexical ordering. Since the lexical ordering of sequence of natural numbers is a well-order, there cannot be infinite sequences of simplifications. □

Lemma 9.5 (Un-simplifiable terms are expanded terms)**.** *Suppose that there is no* a' *such that* $a \rightarrowtail a'$*. Then,* a *is an expanded term.*

Proof. We are proving three results by simultaneous induction, proving that subterms of a are either expanded terms or simple terms depending on their contexts, and that subtypes of a are ML types.

Consider a term a in B such that $B[a]$ is irreducible. Then a is an expanded term e. By case analysis:

- If the term starts with a let binding $\mathsf{let}\ x = a\ \mathsf{in}\ b$: by induction hypothesis in the context $B[\mathsf{let}\ x = []\ \mathsf{in}\ b]$, a must be a simple term s or of the form ${s_1}^p\ s_2$. If it were a simple term s, we could reduce the let, so it must be an application. Then, by induction hypothesis in $B[\mathsf{let}\ x = {s_1}^p\ s_2\ \mathsf{in}\ []]$, the term b must be an expanded term, thus $\mathsf{let}\ x = {s_1}^p\ s_2\ \mathsf{in}\ b$ is an expanded term.

- If the term starts with a pattern matching, the term we are matching on must be a simple term (by induction hypothesis, since $\mathsf{match}\ []\ \mathsf{with}\ \ldots$ is a valid F context). Then, we can apply the induction hypothesis on each branch (since each branch forms a valid context B'): all branches are expanded terms, thus the pattern matching is an expanded term.

- Otherwise, B is of the form $B[F[]]$ with $F = []$, and by induction hypothesis, the term must be a simple tern.

Consider a in $B[F[]]$ such that $B[F[a]]$ is irreducible. Then:

- either a is a simple term s
- or F is $\mathsf{let}\ x = []\ \mathsf{in}\ b$ and a is ${s_1}^p\ s_2$
- or F is $[]$ and a is $\mathsf{match}\ s\ \mathsf{with}\ \ldots$

By looking at the different cases for a:

- If a is x or $*p$, it is a simple term.
- If a is $\Lambda(\alpha : \mathsf{Typ}).\ u$, apply the induction hypothesis to u in $B[F[\Lambda(\alpha : \mathsf{Typ}).\ []]]$. Only the first case is possible, thus u is simple and a is simple.
- If a is $b\ \tau$: by induction hypothesis applied to a in $B[F[[]\ \tau]]$, b must be a simple term. Then, by induction hypothesis applied to τ in $B[F[b\ []]]$, τ must be an ML type, thus a is simple.
- If a is $\mathsf{fix}^p\ x\ (y : \tau) : \sigma\ .\ b$, we first apply the induction hypothesis to τ in $B[F[\mathsf{fix}^p\ x\ (y : []) : \sigma\ .\ b]$ to find out it is an ML type, then similarly for σ. Finally, we apply the induction hypothesis to b in the context B' equal to $B[F[\mathsf{fix}^p\ x\ (y : \tau) : \sigma\ .\ []]$ and conclude it is an expanded term. Thus a is simple.
- If a is ${b_1}^p\ b_2$: by induction hypothesis, both b_1 and b_2 are simple terms. Then, F must be $\mathsf{let}\ x = []\ \mathsf{in}\ b$ for some b, otherwise $B[F[a]$ would reduce.
- If a is a constructor, we similarly show that all types and terms appearing in the constructor must be simple.
- If a is $\mathsf{match}\ b\ \mathsf{with}\ \ldots$, b must be a simple term by induction hypothesis. Then, if $F \neq []$, we could move the pattern matching around F. Thus $F = []$.

Finally consider τ in $B[F[]]$ such that $B[F[\tau]]$ is irreducible. Then τ is a simple type τ, by a similar case analysis. □

9.3 Removing equalities

Simplification removes all apparent result reuse and term-level pattern matching. But we also need the typing derivation of the final term to be a valid ML typing derivation. This is ensured by performing a final rewriting step on the term that also rewrites its typing derivation to be in ML.

We need to take care of equalities. We can prove that in an environment containing only type variables, two equal ML types are syntactically equal. We can extend this result to equalities in *inhabited* environments, *i.e.* environments for which we can provide an instantiation of each type and term variable.

First, we prove a stronger version of inversion (Lemma 6.23):

Lemma 9.6 (Equalities in almost empty environments)**.** *Suppose that* $(\alpha_i : \mathsf{Typ}, x_i : \alpha_i)^i \vdash \nu_1 : \mathsf{Sch} \simeq \nu_2 : \mathsf{Sch}$*. Then,* $\nu_1 = \nu_2$*.*

Proof. We already know (Lemma 6.14) that if the two types have the same head (*i.e.* arrow, type constructor, or quantification), their tails are equal, thus are syntactically equal by induction.

We therefore only to prove that the heads are equal.

- If the heads are both type variables α_1 and α_2: if they are identical, the types are equal. Otherwise, take $\gamma(\alpha_1) = \mathsf{unit}$ (and $\gamma(x) = \mathsf{Unit}$ for the corresponding term); $\gamma(\alpha_2) = \mathsf{bool}$ (and $\gamma(x) = \mathsf{True}$). Then we would have $\vdash \mathsf{unit} : \mathsf{Sch} \simeq \mathsf{bool} : \mathsf{Sch}$, which is impossible (for the same reasons as in Lemma 6.23: their interpretations are distinct).
- If one head is a type variable (let's say ν_1) and the other not a type variable, instantiate the type variable with unit or bool, whichever is different from ν_2.
- Otherwise proceed as in Lemma 6.23.

□

We now consider more closely the environments. We prove that an environment Γ is either inhabited, *i.e.* we can provide an instance of this environment, or that we can prove in $e\mathsf{ML}$ that it is absurd, by exhibiting a non-expansive term returning void (or any type). For example, the environment Γ_1 equal to $x : \forall(\alpha : \mathsf{Typ})\ \alpha \to \alpha, y : \mathsf{nat}$ is inhabited by γ_1 such that $\gamma_1(x)$ is the identity function and $\gamma_1(y)$ is $\mathsf{S}\ \mathsf{Z}$, while Γ_2 equal to $x : \forall(\alpha : \mathsf{Typ})\ \alpha$ is not, since $x\ \mathsf{void}$ has type void. This is made slightly more complicated by free type variables, so we instead consider instances of an environment that type in an almost empty environment $(\alpha_i : \mathsf{Typ}, x_i : \alpha_i)^i$.

Lemma 9.7 (Environment inhabitant)**.** *Consider an* ML *environment* Γ *with type variables* $(\alpha_i)^i$*. Then, either there exists* γ *such that* $(\alpha_i : \mathsf{Typ}, x_i : \alpha_i)^i \vdash \gamma : \Gamma$ *mapping each type variable to itself and we say the environment is inhabited, or there exists a non-expansive term* u *such that* $\Gamma \vdash u : \mathsf{void}$*.*

Proof. Construct γ as follows: map type variables to themselves. For a term variable of type $\forall(\beta_j : \mathsf{Typ})^j\ \tau$ with τ of kind Typ, construct a value as follows by induction on the type τ:

- If the type is α_i with α_i in Γ, take x_i.
- If the type is β_j for some j, we cannot construct a term.
- If the type is $\tau_1 \to \tau_2$, take the term $\mathsf{fix}^\pi\ x\ (y : \tau_1) : \tau_2\ .\ x\ y$.
- If the type is $\zeta\ (\tau_j)^j$, choose a constructor d_i such that its arguments are constructible with the same process.

Now consider the case where we do not succeed in building γ: it means that there is some variable $x : \forall(\beta_j : \mathsf{Typ})^j\ \tau$ for which we were not able to construct a value. Then, we construct a term that matches on $u = x\ (\mathsf{void})^j$ of type $\tau[\beta_j \leftarrow \mathsf{void}]^j$ and that is of type void. By induction on τ:

- If the type is α_i with $\alpha_i \in \Gamma$ we were able to construct a value.
- If the type is β_j for some j, it is substituted by void. Then we return u.

- If the type is $\tau_1 \to \tau_2$, we were able to construct a value.
- If the type is $\zeta\,(\tau_j)^j$: the constructors are d_i taking arguments of type $(\tau_{ik})^k$. For each constructor there must be at least one argument k for which we cannot construct a term: we then have a term u_k of type σ matching on x_{ik} of type τ_{ik}. Then, we construct $\mathsf{match}\ u\ \mathsf{with}\ (d_i(\tau_j)^j(x_{ik})^k \to u_k)^i$.

□

Lemma 9.8 (Equalities are almost trivial)**.** *Consider an environment* Γ *without equalities or results and two* ML *types* τ_1 *and* τ_2*. Suppose* $\Gamma \vdash \tau_1 : \mathsf{Sch} \simeq \tau_2 : \mathsf{Sch}$*. Then either* Γ *is inhabited* $\tau_1 = \tau_2$ *or there exists a non-expansive* ML *term* u *such that for any type* σ*, we have* $\Gamma \vdash u : \sigma$*, and for any* a *such that* $\Gamma \vdash^p a : \sigma \Rightarrow \Delta$*, we have*$\mathsf{TermsEqual}(\Gamma \vdash^p u : \sigma \simeq a : \sigma)$*.*

Proof. We either construct an environment γ that gives a value all bindings of Γ, or a term u of type void by Lemma 9.7. If we have a non-expansive term of type void, it is equal to any other term (as each equality required for term equality can be proved by splitting on void).

Otherwise, we use Lemma 9.6. □

Then we can proceed with simplification:

Lemma 9.9 (Expanded terms type without equalities and with ML types only)**.** *Suppose* Γ *is an* ML *environment and* τ *an* ML *type. Consider an expanded term* e *such that* $\Gamma \vdash^p e : \tau \Rightarrow \Delta$*. Then there exists an* ML *term* a *such that* $\mathsf{TermsEqual}(\Gamma \vdash^p e : \tau \simeq a : \tau)$ *and* $\Gamma \vdash a : \tau$ *in* ML*.*

Proof. During simplification, we accumulate a substitution γ describing how variables and results must be replaced by simple terms to obtain a term that types in an ML environment. We write $\Gamma \vdash^{\mathsf{ML}} \gamma : \Gamma'$ if:

- for all $(x : \tau) \in \Gamma'$, $\gamma(x)$ is a simple term and $\Gamma \vdash^{\mathsf{ML}} \gamma(x) : \tau$;
- for all $(\mathsf{True})*p : \tau \in \Gamma'$, $\Gamma \vdash^{\mathsf{ML}} \gamma(\alpha) : \tau$;
- and for all equalities $(u_1 \simeq u_2) : \tau$ in Γ', $\Gamma \vdash \gamma(u_1) : \tau \simeq \gamma(u_2) : \tau$.

The proof is by simultaneous induction on e and s.

For simple terms s, we prove that if $\Gamma' \vdash s : \tau$ and Γ is an inhabited ML environment such that $\Gamma \vdash^{\mathsf{ML}} \gamma : \Gamma'$, then there exists s' and ν such that $\Gamma \vdash^{\mathsf{ML}} s' : \nu$ and $\Gamma \vdash \gamma(s) : \tau \simeq s' : \nu$ We can always assume that the type derivation for the term does not end with a conversion (as we can always remove all conversions and reapply the equalities as conversion in the equality proof afterwards).

- If the term is x, with $\Gamma' \vdash x : \tau$: we have $(x : \tau) \in \Gamma'$. Then, $\gamma(x)$ is an ML type with $\Gamma \vdash^{\mathsf{ML}} \gamma(x) : \tau$.
- If the term is $s\ \nu$, with $\Gamma' \vdash s : \forall(\alpha : \mathsf{Typ})\,\tau$: we have $\Gamma' \vdash s\ \nu : \tau[\alpha \leftarrow \nu]$. There exists a term s' and a type σ such that $\Gamma \vdash \gamma(s) : \forall(\alpha : \mathsf{Typ})\ \tau \simeq s' : \sigma$, with σ an ML type. By extraction, $\Gamma \vdash \forall(\alpha : \mathsf{Typ})\,\tau : \mathsf{Sch} \simeq \sigma : \mathsf{Sch}$. Since Γ is inhabited, σ must be of the form $\forall(\alpha : \mathsf{Typ})\ \tau'$. Then, we have, in ML (because ν is an ML type), $\Gamma \vdash s'\ \nu : \tau'[\alpha \leftarrow \nu]$, and $\Gamma \vdash \gamma(s\ \nu) : \tau[\alpha \leftarrow \nu] \simeq s'\ \nu : \tau'[\alpha \leftarrow \nu]$

- We reason similarly for constructor applications, universal quantification and function definitions.

For expanded term, we prove that if $\Gamma' \vdash^p e : \nu \Rightarrow \Delta$ and Γ is an inhabited ML environment such that $\Gamma \vdash^{\mathsf{ML}} \gamma : \Gamma'$, then there exists a such that $\Gamma \vdash^{\mathsf{ML}} s : \nu$ and $\mathsf{TermsEqual}(\Gamma \vdash^p \gamma(e) : \nu \simeq a : \nu)$. First, we check if Γ is inhabited. Otherwise, we replace the term by the equal, non-expansive term proving that Γ is not inhabited (Lemma 9.7). Then by case analysis:

- If we return a simple term s at the ML type ν: there exists an equivalent term s' at ML type ν' that types in ML. Since Γ is inhabited, by Lemma 9.8, $\tau = \tau'$.

- For let bindings $\mathsf{let}\ x = {s_1}^p\ s_2\ \mathsf{in}\ e$: there exists τ_1 and τ_2 such that $\Gamma \vdash s_1 : \tau_1 \to \tau_2$ and $\Gamma \vdash s_2 : \tau_1$.

  By induction hypothesis, there exists s'_1 and s'_2, σ_1 and τ'_2 such that $\Gamma \vdash \gamma(s_1) : \tau_1 \to \tau_2 \simeq s'_1 : \sigma_1$, and $\Gamma \vdash \gamma(s_2) : \tau_1 \simeq s'_2 : \tau'_1$. Since Γ is inhabited, by Lemma 9.8, there exists τ''_1 and τ'_2 such that $\sigma_1 = \tau''_1 \to \tau'_2$. We thus have $\Gamma \vdash \tau'_1 : \mathsf{Sch} \simeq \tau''_1 : \mathsf{Sch}$, thus we have $\tau'_1 = \tau''_1$. Then we have, in ML, $\Gamma \vdash s'_1\ s'_2 : \tau'_2$.

  Then, define $\gamma' = \gamma, *p \leftarrow x$. We have $\Gamma, x : \tau'_2 \vdash^{\mathsf{ML}} \gamma' : \Gamma, x : \tau_2, (\mathsf{True})*p : \tau_2, (x \simeq *p) : \tau_2$. Thus, by induction hypothesis, there exists a such that $\Gamma, x : \tau'_2 \vdash^{\mathsf{ML}} a : \nu$ and $\mathsf{TermsEqual}(\Gamma, x : \tau'_2 \vdash^{\gamma(e)} \nu :\simeq a : \nu)$. Then, we have $\Gamma \vdash^{\mathsf{ML}} \mathsf{let}\ x = s'_1\ s'_2\ \mathsf{in}\ a : \nu$, and $\mathsf{TermsEqual}(\Gamma \vdash^p \gamma(\mathsf{let}\ x = {s_1}^p\ s_2\ \mathsf{in}\ e) : \nu \simeq \mathsf{let}\ x = s'_1\ s'_2\ \mathsf{in}\ a : \nu)$.

- For pattern matching $\mathsf{match}\ s\ \mathsf{with}\ (d_i(\nu_j)^j(x_{ik})^k \to e_i)^i$: By induction hypothesis, there exists s' and an ML type τ such that $\Gamma \vdash^{\mathsf{ML}} s' : \tau$ and $\Gamma \vdash \gamma(s) : \zeta\,(\nu_j)^j \simeq s' : \tau$. Then, $\tau = \zeta\,(\nu_j)^j$ by Lemma 9.8 since Γ is inhabited. We can reduce s' so that there is no type abstraction/application redex, preserving the equality. Then, consider the different cases for s': it can only be a variable, a constructor, or a type application. If it is a type application $s' = s''\ \sigma$, consider again s'': it can only be a variable or a type application (it cannot be a type abstraction as we removed these redexes, and it cannot be a constructor because it would be ill-typed). But then, that would mean that we have a variable in context of type $\forall\overline{\alpha}\,\zeta\,(\nu_j)^j$, which is forbidden.

  If s' is a variable x, take, for all i, $\gamma_i = \gamma, x \leftarrow d_i(\nu_j)^j(x_{ik})^k$. We then have $\Gamma, (x_{ik} : \sigma_{ik})^k \vdash^{\mathsf{ML}} \gamma_i : \Gamma', (x_{ik} : \sigma_{ik})^k, (x \simeq d_i(\nu_j)^j(x_{ik})^k) : \zeta\,(\nu_j)^j$. Then, we can apply the induction hypothesis in each branch and build an ML term.

  Otherwise, $s' = d_i(\nu_j)^j(s_k)^k$ for some i. Then, we take $\gamma' = \gamma, (x_{ik} \leftarrow)^k s_k$ and apply the induction hypothesis in the branch to obtain a term a. By congruence and reduction, the original term reduces to $\gamma(e_i)[x_{ik} \leftarrow s_k]^k = \gamma'(e_i)$, thus we get the required equality.

□

Theorem 9.1 (Simplification)**.** *Let Γ be an ML environment whose polymorphic bindings are all functions. Suppose $\Gamma \vdash^p a : \tau \Rightarrow \Delta$ with τ an ML type. Then, there exists an ML term a' such that $\Gamma \vdash^{\mathsf{ML}} a' : \tau$, and $\mathsf{TermsEqual}(\Gamma \vdash^p a : \tau \simeq a' : \tau)$.*

Proof. By Lemma 9.4, there exists a maximal reduction a'' of a by $\rightarrowtail$. By Lemma 9.5, this maximal reduction is an expanded term. By Lemma 9.3, $\mathsf{TermsEqual}(\Gamma \vdash^p a : \tau \simeq a'' : \tau)$. By Lemma 9.9, there exists a' such that $\Gamma \vdash^{\mathsf{ML}} a' : \tau$ and $\mathsf{TermsEqual}(\Gamma \vdash^p a'' : \tau \simeq a' : \tau)$. By transitivity, $\mathsf{TermsEqual}(\Gamma \vdash^p a : \tau \simeq a' : \tau)$. $\square$

Part III

Encoding ornaments

Chapter 10

Encoding ornaments in mML

We now consider how ornaments are described and represented in the lifting process. This section bridges the gap between mML, a language for meta-programming that does not have any built-in notion of ornamentation, and the interface presented to the user for ornamentation.

We define both the datatype ornaments, *i.e.* the base ornaments specified by the user, and the higher-order functional ornaments that can be built from them.

As a running example, we use the ornament $\mathsf{natlist}\ \alpha$ from natural numbers to lists:

$$\begin{array}{l}\mathsf{type\ ornament\ natlist}\ \alpha : \mathsf{nat} \to \mathsf{list}\ \alpha\ \mathsf{with}\\ \quad \mid \mathsf{Z} \to \mathsf{Nil}\\ \quad \mid \mathsf{S}\ w \to \mathsf{Cons}\ (_, w)\ \mathsf{when}\ w : \mathsf{natlist}\ \alpha\end{array}$$

The ornament $\mathsf{natlist}\ \alpha$ defines, for all types α, a relation between values of its *base type* nat, which we write $(\mathsf{natlist}\ \alpha)^-$, and its *lifted type* $\mathsf{list}\,\alpha$, written $(\mathsf{natlist}\ \alpha)^+$: the first clause says that Z is related to Nil; the second clause says that if w_- is related to w_+, then $\mathsf{S}\ w_-$ is related to $\mathsf{Cons}\ (v, w_+)$ for any value v. As a notation shortcut, the variables w_- and w_+ are identified in the definition above.

A higher-order ornament type $\mathsf{natlist}\ \alpha \to \mathsf{natlist}\ \alpha$ relates two functions f_- of type $\mathsf{nat} \to \mathsf{nat}$ and f_+ of type $\mathsf{list}\ \tau \to \mathsf{list}\ \tau$ when for related inputs v_- and v_+, the outputs $f_-\ v_-$ and $f_+\ v_+$ are related.

The ornament $\mathsf{list}\ (\mathsf{natlist}\ \alpha)$ relates a list of natural numbers to lists of lists whose lengths are given by the corresponding element in the base list.

10.1 Ornamentation as a logical relation

We formalize this idea by defining a family of *ornament types* corresponding to the ornamentation definitions given by the user and giving them an interpretation in the logical relation. Then, we say that one term is a lifting of another if they are related at the desired ornament type.

The syntax of ornament types, given on Figure 10.1, mirrors the syntax of types. An ornament type, written ω, may be an ornament variable φ, a datatype ornament $\chi\ \overline{\omega}$, a higher-order ornament $\omega_1 \to \omega_2$, or an *identity* ornament $\zeta\ (\omega)^i$, which is automatically defined for any datatype of the same name (ω_i indicates

$$
\begin{array}{rll}
\chi & ::= & \text{Base ornaments} \\
 & \mid \ \mathsf{natlist} & \\
 & \mid \ \ldots & \\
\omega & ::= & \text{Ornament types} \\
 & \mid \ \varphi & \text{Ornament variables} \\
 & \mid \ \chi\,(\omega)^i & \text{Datatype ornament} \\
 & \mid \ \zeta\,(\omega)^i & \text{Identity ornament} \\
 & \mid \ \omega \to \omega & \text{Function}
\end{array}
$$

Figure 10.1: Ornament types

$$
\begin{array}{c}
\alpha^\epsilon = \alpha \\
(\omega_1 \to \omega_2)^\epsilon = \omega_1^\epsilon \to \omega_2^\epsilon \\
(\zeta\,(\omega_i)^i)^\epsilon = \zeta\,(\omega_i^\epsilon)^i
\end{array}
$$

$$
\frac{\chi\,(\alpha_i)^i : \tau \Rightarrow \sigma}{(\chi\,(\omega_i)^i)^- \quad = \quad \tau[\alpha_i \leftarrow \omega_i^-]^i}
\qquad
\frac{\chi\,(\alpha_i)^i : \tau \Rightarrow \sigma}{(\chi\,(\omega_i)^i)^+ \quad = \quad \sigma[\alpha_i \leftarrow \omega_i^+]^i}
$$

Figure 10.2: Projection of ornament types

how the i-th type argument of the datatype is ornamented). An ornament type ω is interpreted as a relation between terms of type ω^- and ω^+. The projection operation, defined on Figure 10.2, depends on the projections of the datatype ornaments: they are given by the global judgment $\chi\,\overline{\alpha} : \tau \Rightarrow \tau$. We also define a well-formedness judgment $(\alpha_i)^i \vdash \omega$, for ornaments given an environment of type variables in Figure 10.3. For example, the ornament $\mathsf{list}\ (\mathsf{natlist\ nat})$ describes the relation between lists whose elements have been ornamented using the ornament $\mathsf{natlist\ nat}$. Thus, its projections are $(\mathsf{list}\ (\mathsf{natlist\ nat}))^-$ equal to $\mathsf{list\ nat}$ and $(\mathsf{list}\ (\mathsf{natlist\ nat}))^+$ equal to $\mathsf{list}\ (\mathsf{list\ nat})$.

The projection is defined and well-kinded for any well-formed ornament type:

Lemma 10.1 (Projection is a type)**.** *If $\overline{\alpha} \vdash \omega$ holds, then we have $\overline{\alpha} \vdash (\omega)^\epsilon : \mathsf{Typ}$.*

Proof. By induction on the derivation of $\overline{\alpha} \vdash \omega$. □

We define in the next section how to interpret the base ornaments χ, and focus here on the interpretation of higher-order ornaments $\omega_1 \to \omega_2$ and identity

$$
(\alpha_i)^i \vdash \alpha_i
\qquad
\frac{\overline{\alpha} \vdash \omega_1 \qquad \overline{\alpha} \vdash \omega_2}{\overline{\alpha} \vdash \omega_1 \to \omega_2}
\qquad
\frac{\zeta : (\mathsf{Typ})^j \to \mathsf{Typ} \qquad (\overline{\alpha} \vdash \omega_j)^j}{\overline{\alpha} \vdash \zeta\,(\omega_j)^j}
$$

$$
\frac{\begin{array}{c}\chi\,(\alpha_j)^j : \ldots \Rightarrow \ldots \\ (\overline{\alpha} \vdash \omega_j)^j\end{array}}{\overline{\alpha} \vdash \chi\,(\omega_j)^j}
$$

Figure 10.3: Well-formedness of ornament types

$$
\begin{array}{rll}
P, Q & ::= & \text{Deep patterns} \\
& \mid\ x & \text{Variable} \\
& \mid\ d\,\overline{\tau}\,\overline{P} & \text{Constructor} \\
& \mid\ _ & \text{Wildcard} \\
& \mid\ \bot & \text{Empty} \\
& \mid\ (P \mid P) & \text{Alternative}
\end{array}
$$

Figure 10.4: Deep patterns

ornaments $\zeta\,(\omega_i)^i$.

The interpretation we want for higher-order ornaments is as functions sending arguments related by ornamentation to results related by ornamentation. But this is exactly what the interpretation of the arrow *type* $\tau_1 \to \tau_2$ gives us, if we replace the types τ_1 and τ_2 by ornament types $\omega_1 \to \omega_2$. Thus, we do not have to define a new interpretation for higher-order ornaments, it is already included in the logical relation. For this reason, we use the function arrow and the ornament arrow interchangeably (when talking about the logical relation).

We have the same phenomenon for the identity ornament: constructors are related at the identity ornament if their arguments are related. Once more, we can simply take the interpretation of a datatype $\zeta\,(\tau_i)^i$ and, by replacing the type parameters $(\tau_i)^i$ by ornament parameters $(\omega_i)^i$, reinterpret it as an interpretation of the identity ornament.

Finally, ornament variables must be interpreted by getting the corresponding relation in the relational environment. This is exactly the interpretation of a *type* variable.

Thus, the common subset between types and ornament specifications can be identified, because their interpretations are the same. This property plays a key role in the instantiation: from a relation at a type, we deduce, by proving the correct instantiation, a relation at an ornament.

10.2 Deep pattern matching in *e*ML

Ornaments are defined through deep pattern matching, and our presentation of *e*ML only allows for shallow pattern matching. In this section, we present an encoding of deep pattern matching in *e*ML, based on *e*ML features. An alternative way of handling pattern matching would be to compile it down to a tree of simple pattern matching [Maranget, 2008], but this would introduce deeper changes to the term. Instead, the encoding given here maintains a structure very close to the original deep pattern matching. The syntax of patterns we support is defined in Figure 10.4. A pattern may be a variable x, a wildcard pattern $_$, the void pattern $\bot$ that matches nothing, an alternative pattern $P \mid Q$ that matches if either P or Q matches, or a constructor pattern $d\,(\tau_i)^i\,(P_j)^j$ matching a constructor d whose arguments match the pattern P_j. We do not support pattern synonyms P as Q as they cannot be used in ornament definitions.

Patterns are checked by the judgment $\vdash P : \sigma \rightsquigarrow \Sigma$, asserting that the pattern P matches values of type σ and returns variables in Σ. The definition of the judgment is given in Figure 10.5. Apart from checking typing, the judgment

$$\textsc{P-Var}\quad \frac{}{\vdash x : \sigma \rightsquigarrow x : \sigma} \qquad \textsc{P-Wild}\quad \frac{}{\vdash _ : \sigma \rightsquigarrow \varnothing} \qquad \textsc{P-Empty}\quad \frac{}{\vdash \bot : \sigma \rightsquigarrow \Sigma}$$

$$\textsc{P-Alternative}\quad \frac{\vdash P_1 : \sigma \rightsquigarrow \Sigma \qquad \vdash P_2 : \sigma \rightsquigarrow \Sigma}{\vdash P_1 \mid P_2 : \sigma \rightsquigarrow \Sigma}$$

$$\textsc{P-Con}\quad \frac{\begin{array}{c}\vdash d : \forall(\alpha_i : \mathsf{Typ})^i\ (\tau_j)^j \to \zeta\ (\alpha_i)^i \\ \vdash P_j : \tau_j[\alpha_i \leftarrow \sigma_i]^i \rightsquigarrow \Sigma_j \qquad \forall j, k, \Sigma_j \# \Sigma_k\end{array}}{\vdash d(\tau_i)^i(P_j)^j : \zeta\ (\sigma_i)^i \rightsquigarrow (\Sigma_j)^j}$$

Figure 10.5: Typing for patterns

also checks that each variable is defined only once (we write $\Sigma_1 \# \Sigma_2$ if the variables in Σ_1 and Σ_2 are distinct), and that alternatives define the same variables with the same types.

We provide an encoding of patterns into eML. This is interesting, because it allows us to add deep patterns to eML without changing the core of the language and adding reasoning and reduction rules for these patterns. For each pattern P, we define a non-expansive term $\textsc{Pat}(u, P)$ that is equal to None when the pattern does not match, and $\mathsf{Some}\ ((v_i)^i)$ when it matches, where the v_i are the values matched by the pattern. This definition is given in Figure 10.6. For matching on constructors, we use an operation $\textsc{Combine}(u_i)^i$ on the result of matching on each argument: it returns Some with a tuple of all the variables of each result concatenated, or None if any of the matches fail. For example, $\textsc{Combine}(\mathsf{Some}\ 1, \mathsf{None})$ is equal to None and $\textsc{Combine}(\mathsf{Some}\ 1, \mathsf{Some}\ (\mathsf{True}, 2))$ is equal to $\mathsf{Some}\ (1, \mathsf{True}, 2)$. Its definition is also given in Figure 10.6.

The expression $\textsc{Pat}(u, P)$ is well-typed:

Lemma 10.2 (The match term is well-typed)**.** *Suppose* $\Gamma \vdash u : \sigma$ *and* $\vdash P : \sigma \rightsquigarrow (x_i : \tau_i)^i$*. Then,* $\Gamma \vdash \textsc{Pat}(u, P) : \mathsf{option}\ (\tau_i)^i$*.*

Proof. By induction on the derivation of the pattern typing, using as a lemma the fact that if $(\Gamma \vdash u_i : \mathsf{option}\ (\tau_{ij})^j)^i$, then $\Gamma \vdash \textsc{Combine}(u_i)^i : \mathsf{option}\ (\tau_{ij})^{ij}$. □

From a non-expansive term u, a set of patterns $(P_i)^i$, associated expressions (terms, types, or non-expansive terms) $(X_i)^i$ and a default value Y, we define in Figure 10.6 the pattern matching with default value $\textsc{Match}(u, (P_i \to X_i)^i, Y)$: it returns the X_i of the first clause that matches a pattern, or Y if nothing matches.

We say that a set of patterns $(P_i)^{1 \leqslant i \leqslant n}$ is *complete* for σ in an environment Γ if one pattern in Γ always matches, *i.e.* if we have:

$$\Gamma, x : \sigma \vdash \mathsf{True} : \mathsf{bool} \simeq \textsc{Match}(x, (P_i \to \mathsf{True})^i, \mathsf{False}) : \mathsf{bool}$$

$$
\begin{array}{lcl}
\textsc{Pat}(u, x) & = & \mathsf{Some}\ u \\
\textsc{Pat}(u, _) & = & \mathsf{Some}\ () \\
\textsc{Pat}(u, \bot) & = & \mathsf{None} \\
\textsc{Pat}(u, P \mid Q) & = & \mathsf{match}\ \textsc{Pat}(u, P)\ \mathsf{with} \\
 & & \quad \mathsf{Some}\ x \to \mathsf{Some}\ x \\
 & & \quad \mathsf{None} \to \textsc{Pat}(u, Q) \\
\textsc{Pat}(u, d_i(\tau_j)^j (P_k)^k) & = & \mathsf{match}\ u\ \mathsf{with} \\
 & & \quad d_i(\tau_j)^j (x_k)^k \to \textsc{Combine}(\textsc{Pat}(x_k, P_k))^k \\
 & & \quad (d_\ell(\tau_j)^j (x_{k\ell})^k \to \mathsf{None})^{\ell \neq i} \\
\textsc{Combine}(u_i)^i & = & \textsc{Combine'}((u_i)^i; \varnothing) \\
\textsc{Combine'}(\varnothing; (x_j)^j) & = & \mathsf{Some}\ (x_j)^j \\
\textsc{Combine'}((u_0, (u_i)^{i \neq 0}); (x_j)^j) & = & \\
\quad \mathsf{match}\ u_0\ \mathsf{with} & & \\
\quad\quad \mathsf{None} \to \mathsf{None} & & \\
\quad\quad \mathsf{Some}\ y \to \mathsf{match}\ y\ \mathsf{with} & & \\
\quad\quad\quad (y_k)^k \to \textsc{Combine'}((u_i)^{i \neq 0}; ((x_j)^j, (y_k)^k)) & & \\
\textsc{Match}(u, (P_i \to X_i)^{i \in \{1,\ldots,n\}}, Y) & = & \mathsf{match}\ \textsc{Pat}(u, P_1)\ \mathsf{with} \\
 & & \quad \mathsf{Some}\ (x_{1j})^j \to X_1 \\
 & & \quad \mathsf{None} \to \\
 & & \qquad \vdots \\
 & & \quad \mathsf{match}\ \textsc{Pat}(u, X_n)\ \mathsf{with} \\
 & & \quad\quad \mathsf{Some}\ (x_{nj})^j \to b_n \\
 & & \quad\quad \mathsf{None} \to Y
\end{array}
$$

Figure 10.6: The matching function

We say that two patterns P and Q matching σ are non-overlapping in Γ if

$$\begin{array}{lll} \Gamma, x : \sigma \vdash \mathsf{True} : \mathsf{bool} \simeq & \mathsf{match}\ \textsc{Pat}(x, P)\ \mathsf{with} & : \mathsf{bool} \\ & \quad \mid \mathsf{None} \to \mathsf{True} & \\ & \quad \mid \mathsf{Some}\ x \to \mathsf{match}\ \textsc{Pat}(x, Q)\ \mathsf{with} & \\ & \qquad \mid \mathsf{None} \to \mathsf{True} & \\ & \qquad \mid \mathsf{Some}\ y \to \mathsf{False} & \end{array}$$

A set of patterns is non-overlapping (in Γ) if the patterns are pairwise non-overlapping.

From this, we can define the deep pattern matching:

Definition 10.1 (Deep pattern matching)**.** *We define:*

$$\begin{array}{rl} \mathsf{match}\ a\ \mathsf{with}\ (P_i \to b_i)^i = & \mathsf{let}\ x = a\ \mathsf{in} \\ & \textsc{Match}(x, (P_i \to b_i)^i, \mathsf{Unit}) \end{array}$$

where the $(x_{ij})^j$ are the variables matched by P_i.

Similarly, for types, we define:

$$\mathsf{match}\ u\ \mathsf{with}\ (P_i \to \tau_i)^i = \textsc{Match}(u, (P_i \to \tau_i)^i, \mathsf{unit})$$

$\diamondsuit$

Deep pattern matching verifies the following (derived) typing rule:

$$\textsc{Match-Deep}\quad \frac{\Gamma \vdash u : \sigma \qquad (P_i)^i \text{ complete for } \Gamma \vdash \sigma \qquad (\vdash P_i : \sigma \rightsquigarrow (x_{ik} : \tau_{ik})^{ik})^i \quad \begin{pmatrix} \Gamma, ((\textsc{Pat}(u, P_j) \simeq \mathsf{None}) : \mathsf{option}\ (\tau_{jk})^k)^{j<i}, \\ (x_{ik} : \tau_{ik})^{ik}, (\textsc{Pat}(u, P_i) \simeq \mathsf{Some}\ (x_{ik})^k) : \mathsf{option}\ (\tau_{ik})^k \vdash a_i : \tau \end{pmatrix}^i}{\Gamma \vdash \mathsf{match}\ u\ \mathsf{with}\ (P_i \to a_i)^i : \tau}$$

Proof. This follows from applying the (syntax directed) pattern matching rule. The last (absurd) branch of the pattern matching is unreachable: it can only be reached if all matches return None, but then by the completeness condition, we obtain True equal to False. $\square$

10.3 Defining datatype ornaments

In general, an ornament definition is a mutually recursive group of definitions, each of the form:

$$\mathsf{type\ ornament}\ \chi\,(\alpha_j)^{j \in J} : \zeta\,(\tau_k)^{k \in K} \Rightarrow \sigma_+\ \mathsf{with}\ (P_i \Rightarrow Q_i\ \mathsf{when}\ (x_{i\ell} : \omega_{i\ell})^{\ell \in L_i})^{i \in I}$$

This defines a datatype ornament χ, parametrized by a set of types $(\alpha_j)^{j \in J}$. The base type is a datatype $\sigma_- = \zeta\,(\tau_k)^{k \in K}$ and the ornamented type is σ_+. The ornamented type is most often a datatype but it needs not be. For example, we can define an ornament $\mathsf{unwrap}\,\alpha : \mathsf{option}\ \alpha \Rightarrow \alpha$ that maps $\mathsf{Some}\ x$ to x:

$$\mathsf{type\ ornament\ unwrap}\,\alpha : \mathsf{option}\ \alpha \Rightarrow \alpha\ \mathsf{with}\ \mathsf{Some}\ x \Rightarrow x\ \mathsf{when}\ x : \alpha$$

The ornament is defined by a series of clauses $P_i \Rightarrow Q_i$ indexed by a set I and mapping values of the base type to values of the ornamented type. The variables

$$
\begin{array}{lcl}
\mathsf{Values}_\gamma(x : \tau) & = & \{\gamma(x)\} \\
\mathsf{Values}_\gamma(_ : \tau) & = & \{v \mid \vdash v : \tau\} \\
\mathsf{Values}_\gamma(\bot : \tau) & = & \varnothing \\
\mathsf{Values}_\gamma(P \mid Q : \tau) & = & \mathsf{Values}_\gamma(P : \tau) \cup \mathsf{Values}_\gamma(Q : \tau) \\
\mathsf{Values}_\gamma(d\,\overline{\tau_k}\,(P_i)^i : \zeta\,(\tau_k)^k) & = & \\
\multicolumn{3}{c}{\left\{ d(\tau_k)^k(v_i)^i \;\middle|\; \begin{array}{c} (d : \forall(\alpha_k)^k\,(\sigma_i)^i \to \zeta\,(\alpha_k)^k) \\ \wedge \forall i, v_i \in \mathsf{Values}_\gamma(P_i : \sigma_i[\alpha_k \leftarrow \tau_k]^k) \end{array} \right\}}
\end{array}
$$

Figure 10.7: Values matched by a pattern

present in the patterns are exactly the $(x_{i\ell})^{\ell \in L_i}$. On the left-hand side, their types are the left-projections of the $\omega_{i\ell}$, while on the right-hand side their types are the right-projections. The ornamentation process ensures that the value of this variable on the left and right-hand side are related at the ornament $\omega_{i\ell}$.

The patterns P_i should be both expressions and patterns, *i.e.* they can only match variables or constructors. This ensures that from a P_i and the values of the variables we can recover the unique value that may have been matched. The typing condition, for each P_i, is: $\vdash P_i : \zeta\,(\tau_k)^k \rightsquigarrow (x_{i\ell} : (\omega_{i\ell})^-)^\ell$. We require for convenience that the $(P_i)^i$ do not overlap. We also require that they form a complete pattern matching: for ornament definitions, an incomplete pattern matching can be made complete by adding some clauses mapping the unmatched values to $\bot$ (our surface language accepts specifying a clause $_ \Rightarrow \bot$ to request the generation of all such patterns).

The patterns $(Q_i)^i$ on the right-hand side can be any patterns matching on the ornamented type. They must obey the typing condition $\vdash Q_i : \sigma_+ \rightsquigarrow (x_{i\ell} : (\omega_{i\ell})^+)^{\ell \in L_i}$. Moreover, we require that they are complete and do not overlap: this allows any term of the ornamented type to be non-ambiguously projected to the base type (as long as we can project the subcomponents).

We give meaning to these definitions by adding them to the logical relation. The interpretation of a datatype ornament is the union of the relations defined by each clause of the ornament. For each clause, the left and right hand-side values of the variables must be related at ornament type given by the user. Since the pattern on the left is also an expression, the value on the left is uniquely defined once values have been chosen for the variables. The pattern on the right can still represent a set of different values (none, one, or many, depending on whether the empty pattern, an or-pattern or a wildcard is used). We define a function $\mathsf{Values}_\gamma(P : \tau)$ that returns the set of values of types τ matching P where the values of the free variables in P are taken from γ. Its definition is given in Figure 10.7.

Then, the interpretation is:

$$
\mathcal{T}[\chi\,(\omega_j)^j]_\gamma \;=\; (\sigma_-[\alpha_j \leftarrow \tau_{j\,-}]^j, \sigma_+[\alpha_j \leftarrow \tau_{j\,+}]^j) \;\text{ where } \forall j, \mathcal{T}[\omega_j]_\gamma = (\tau_{j\,-}, \tau_{j\,+})
$$

$$
\mathcal{V}_p[\chi\,(\omega_j)^j]_\gamma = \left\{ (P_i[x_{i\ell} \leftarrow v_{\ell\,-}]^\ell, v_+) \;\middle|\; \begin{array}{l} i \in I \\ \forall \ell \in L_i, (v_{\ell\,-}, v_{\ell\,+}) \in \mathcal{V}_p[\omega_{i\ell}[\alpha_j \leftarrow \omega_j]^j]_\gamma \\ v_+ \in \mathsf{Values}_{(x_{i\ell} \leftarrow v_{\ell\,+})^\ell}(Q_i) \end{array} \right\}
$$

It is in fact quite similar to the interpretation of datatypes: the recursive conditions are the same, but instead of merely applying a constructor, we build the left and right-hand side values using the patterns of the ornament definition.

For example, on $\mathsf{natlist}$, we get the following definition (omitting the typing conditions):

$$\mathcal{V}_k[\mathsf{natlist}\ \tau]_\gamma = \{(\mathsf{Z}, \mathsf{Nil})\} \ \cup\ \{\ (\mathsf{S}(v_-),\ \mathsf{Cons}(_, v_+)\ \mid\ (v_-, v_+) \in \mathcal{V}_k[\mathsf{natlist}\ \tau]_\gamma\}$$

As expected, we have, *e.g.* $(\mathsf{S}\ (\mathsf{S}\ \mathsf{Z}), \mathsf{Cons}\ (v_1, \mathsf{Cons}\ (v_2, \mathsf{Nil}))) \in \mathcal{V}_k[\mathsf{natlist}\ \mathsf{bool}]$ for any boolean values v_1 and v_2.

10.4 Shallow ornaments

This definition of ornamentation is convenient for the surface language, but it unfortunately does not exactly match the encoding. In this section, we'll give a slightly lower-level definition of datatype ornaments that allows us to bridge the gap between the original user-facing definition and the lower-level workings of lifting.

The issue appears when defining ornaments with deep pattern matching on the left-hand side. Suppose we have a type t with one constructor T: $\mathsf{nat} \to \mathsf{t}$. We may define the following ornament broken:

$$\begin{array}{l}\mathsf{type\ ornament\ broken} : \mathsf{option\ t} \Rightarrow \mathsf{list\ bool\ with}\\ \quad \mid \mathsf{Some}\ (\mathsf{T}\ x) \Rightarrow \mathsf{Cons}\ (_, x)\ \mathsf{when}\ x : \mathsf{natlist\ bool}\\ \quad \mid \mathsf{None} \Rightarrow \mathsf{Nil}\end{array}$$

For example, it relates $\mathsf{Some}\ (\mathsf{T}\ \mathsf{Z})$ to $\mathsf{Cons}\ (\mathsf{True}, \mathsf{Nil})$.

Unfortunately this breaks down when trying to write the construction function inj:

$$\lambda^{\#}(x : ???).\ \lambda^{\#}(y : \mathsf{bool}).\ \mathsf{match}\ x\ \mathsf{with}\ \mathsf{Some}\ (\mathsf{T}\ (z : \mathsf{list\ bool})) \to \mathsf{Cons}\ (y, z)$$

This function does not type, as T can only contain a natural number, but at this stage of lifting, we expect the value inside the $\mathsf{Some}\ (\mathsf{T}\ [])$ context to be already lifted to $\mathsf{list\ bool}$.

We can still write this ornament by instead proceeding in two steps: we need to ensure that when we are lifting the Some constructor, the type inside the constructor has already been lifted to a type t' that can hold a $\mathsf{list\ bool}$ (thus we take T': $\mathsf{list\ bool} \to \mathsf{t}'$). We start by defining an ornament $\mathsf{aux} : \mathsf{t} \Rightarrow \mathsf{t}'$ as follows:

$$\begin{array}{l}\mathsf{type\ ornament\ aux} : \mathsf{t} \Rightarrow \mathsf{t}'\ \mathsf{with}\\ \quad \mid \mathsf{T}\ x \Rightarrow \mathsf{T}'\ x\ \mathsf{when}\ x : \mathsf{natlist\ bool}\end{array}$$

Then, we require that first the argument of Some is ornamented with aux, and instead of simply forwarding it, we match on it to remove the $\mathsf{T}\ 0$ constructor. This is a form of *shallow* ornamentation, where each ornament only reads one layer of constructor from the base term. This is easy to express as a construction function, but the surface syntax does not allow for this. We thus offer an extended syntax supporting this: when specifying an ornament of a datatype, one requests a specific ornamentation for each parameter of each constructor, then one matches on the transformed arguments of the constructors.

With this extended syntax, the broken ornament definition can be fixed as follows:

$$\begin{array}{l}\mathsf{type\ ornament\ fixed} : \mathsf{option\ t} \Rightarrow \mathsf{list\ bool}\\ \qquad [\mathsf{Some\ aux} \mid \mathsf{None}]\ \mathsf{with}\\ \quad \mid \mathsf{Some}\ (\mathsf{T}'\ x) \Rightarrow \mathsf{Cons}\ (_, x)\\ \quad \mid \mathsf{None} \Rightarrow \mathsf{Nil}\end{array}$$

These definitions do not use when clauses: instead, we specify which ornament must be used for which argument to each constructor. This is a restriction compared with the previous definition, because it prevents ornaments such as the following from being translated:

$$\begin{array}{l}\mathsf{type\ ornament\ illegal} : \mathsf{pair}\ (\mathsf{bool}, \mathsf{nat}) \Rightarrow \mathsf{either}\ (\mathsf{nat}, \mathsf{list\ bool})\ \mathsf{with} \\ \quad \mid \mathsf{Pair}\ (\mathsf{False}, x) \Rightarrow \mathsf{Left}\ x\ \mathsf{when}\ x : \mathsf{nat} \\ \quad \mid \mathsf{Pair}\ (\mathsf{True}, x) \Rightarrow \mathsf{Right}\ x\ \mathsf{when}\ x : \mathsf{natlist\ bool}\end{array}$$

This cannot be transformed into a valid shallow ornament because we would need to specify two different ornament types for the second argument of Pair. This is in fact a limitation of our encoding of ornaments: to decide how the parameters should be ornamented, we can only look at one level of constructor. Thus, we will reject such ornament definitions when converting to the lower-level representation (§10.5).

Instead of using the aux ornament, we prefer to use a generic ornament that allows to transform the type inside T into any type we want. The target type for this can be the *skeleton* of t (we first introduced the skeleton when informally discussing the ornamentation process in Section 3.1).

Consider a datatype $\zeta\,(\alpha_i)^{i\in I}$ defined by a family of constructors $(d_k)^{k\in K}$ each taking arguments of types $(\tau_{kj})^{j\in J_k}$:

$$\left(d_k : \forall(\alpha_i : \mathsf{Typ})^{i\in I}\ (\tau_{kj})^{j\in J_k} \to \zeta\,(\alpha_i)^{i\in I}\right)^{k\in K}$$

We define the skeleton by *abstracting out* the concrete types from the constructors and replacing them by type parameters: the skeleton of ζ, written $\hat{\zeta}$, is parametrized by types $(\alpha_{kj})^{k\in K, j\in J_k}$ and has constructors:

$$\left(\hat{d}_k : \forall(\beta_{\ell j} : \mathsf{Typ})^{\ell\in K, j\in J_\ell}\ (\beta_{kj})^{j\in J_k} \to \hat{\zeta}\,(\beta_{\ell j})^{\ell\in K, j\in J_\ell}\right)^{k\in K}$$

Let us write $\mathcal{A}_\zeta(\tau_i)^{i\in I}$ for $(\tau_{kj}[\alpha_i \leftarrow \tau_i]^i)^{k\in K, j\in J_k}$, *i.e.* the function that expands arguments of the datatype into arguments of its skeleton. The types $\zeta\,(\tau_i)^i$ and $\hat{\zeta}\,(\mathcal{A}_\zeta(\tau_i)^i)$ are in bijection by construction (the bijection is obtained by adding hats on the constructors, or removing them when going the other way).

For example, the skeleton of t is the type $\hat{\mathsf{t}}\ \alpha$ with a single constructor $\hat{\mathsf{T}} : \alpha \to \hat{\mathsf{t}}\ \alpha$, and there is a bijection between t and $\hat{\mathsf{t}}$ nat.

We would like to define, for each datatype ζ, an ornament $\widetilde{\zeta}$ mapping its values to values of its skeleton with the argument suitably ornamented. For t, we would have, written as a shallow ornament:

$$\begin{array}{l}\mathsf{type\ ornament}\ \widetilde{\mathsf{t}}\ \alpha : \mathsf{t} \Rightarrow \hat{\mathsf{t}}\ \alpha\ \mathsf{with} \\ \qquad [\mathsf{T}\ \alpha] \\ \quad \mid \mathsf{T}\ x \Rightarrow \hat{\mathsf{T}}\ x\end{array}$$

However, this does not quite work, because we cannot accept any ornament for α, but only ornaments whose base type is nat. Thus, we need *constrained* type parameters for ornament definitions: we write $\alpha \lhd \mathsf{nat}$ (or, in code, <:) to abstract over ornaments whose base type is nat. Then our definition becomes:

$$\begin{array}{l}\mathsf{type\ ornament}\ \widetilde{\mathsf{t}}\ (\alpha \lhd \mathsf{nat}) : \mathsf{t} \Rightarrow \hat{\mathsf{t}}\ \alpha\ \mathsf{with} \\ \qquad [\mathsf{T}\ \alpha] \\ \quad \mid \mathsf{T}\ x \Rightarrow \hat{\mathsf{T}}\ x\end{array}$$

For a datatype $\zeta\,(\alpha_i)^i$ defined by a family of constructors $(d_k)^{k\in K}$ each taking arguments of types $(\tau_{kj})^{j\in J_k}$, the ornament $\widetilde{\zeta}$ is defined as:

$$\begin{array}{l}\mathsf{type\ ornament}\ \widetilde{\zeta}\,(\alpha_i)^i\ (\beta_{kj} \lhd \tau_{kj})^{k\in K, j\in J_k} : \zeta\,(\alpha_i)^i \Rightarrow \hat{\zeta}\,(\beta_{kj})^{k\in K, j\in J_k}\ \mathsf{with}\\ \qquad [d_k(\tau_{kj})^{j\in J_k}]^{k\in K}\\ \quad (d_k\ (x_{kj})^{j\in J_k} \Rightarrow \widehat{d_k}\ (x_{kj})^{j\in J_k})^{k\in K}\end{array}$$

Then, we can define the ornament fixed, mechanically, as follows:

$$\begin{array}{l}\mathsf{type\ ornament\ fixed : option\ t \Rightarrow list\ bool\ with}\\ \qquad [\mathsf{Some}\ (\widetilde{\mathsf{t}}\ (\mathsf{natlist\ bool}))\ |\ \mathsf{None}]\\ \quad |\ \mathsf{Some}\ (\hat{\mathsf{T}}\ x) \Rightarrow \mathsf{Cons}\ (_, x)\\ \quad |\ \mathsf{None} \Rightarrow \mathsf{Nil}\end{array}$$

Let us now define the constraints and the interpretation of shallow ornament definitions. Suppose the type ζ has type arguments indexed by L and data constructors indexed by M with arguments indexed by N_m. Consider a shallow ornament definition:

$$\begin{array}{l}\mathsf{type\ ornament}\ \chi\,(\alpha_j)^j\ (\beta_k \lhd \tau_k)^k : \zeta\,(\sigma_\ell)^{\ell\in L} \Rightarrow \sigma_+\ \mathsf{with}\\ \qquad [d_m(\omega_{mn})^{n\in N_m}]^{m\in M}\\ \quad (P_i \Rightarrow Q_i)^{i\in I}\end{array}$$

We first state the conditions for the types appearing in the definition:

- The constraints on the base types are formed with well-defined types: for all k, $(\alpha_j : \mathsf{Typ})^j \vdash \tau_k : \mathsf{Typ}$.
- For all $\ell \in L$, $(\alpha_j : \mathsf{Typ})^j \vdash \sigma_\ell : \mathsf{Typ}$. The β_k are not in scope in the σ_ℓ because their projections are already known.
- σ_+ is a well-defined type: $(\alpha_j : \mathsf{Typ})^j, (\alpha_k : \mathsf{Typ})^k \vdash \sigma_+ : \mathsf{Typ}$.
- The types for each constructor argument are valid ornament types: for all $m \in M$ and $n \in N_m$, $(\alpha_j : \mathsf{Typ})^j, (\alpha_k : \mathsf{Typ})^k \vdash \omega_{mn}$.
- The left-projection of the types for each constructor argument matches the original (base) type of the arguments:

$$((\omega^-_{mn})[\beta_k \leftarrow \tau_k]^k)^{m\in M, n\in N_m} = \mathcal{A}_\zeta(\sigma_\ell)^{\ell\in L}$$

Then, we require the following for the patterns $(P_i)^i$ and $(Q_i)^i$. We assume that all patterns $(P_i)^i$ start by a constructor of ζ. Let us define $\hat{P}_i$ the pattern obtained by replacing the head constructor by its skeleton version in P_i. We require:

- The patterns $(\hat{P}_i)^i$ are both expressions and patterns, and are complete and non-overlapping for $\hat{\zeta}\,(\omega^+_{mn})^{m\in M, n\in N_m}$, the type obtained by projecting all the arguments to the skeleton to their ornamented type.
- The patterns $(Q_i)^i$ on the right-hand side are complete and non-overlapping for σ_+, (but not necessarily expressions).
- For each $i \in I$ there exists Σ_i such that $\vdash \hat{P}_i : \hat{\zeta}\,(\omega^+_{mn})^{m\in M, n\in N_m} \rightsquigarrow \Sigma_i$ and $\vdash Q_i : \sigma_+ \rightsquigarrow \Sigma_i$

The definition of $\widetilde{\zeta}$ we provided satisfies these criteria.

For such a definition, we can define its interpretation in the logical relation. For a clause $i \in I$, we note $P_i = d_i(P_{in})^{n \in N_i}$, binding variables $(x_{ip})^{p \in P_i}$. Two values are related if:

- we can choose a clause $i \in I$ with head constructor d_i such that the left-hand side is $d_i(v_{n-})^{n \in N_i}$;
- for each $n \in N_i$ we have a lifting v_{n+} of v_{n-} at the ornament type ω_{in} specified in the skeleton;
- for all $n \in N_i$, the left-hand side patterns P_{in} match the lifted values v_{n+}, binding the variables $(x_{ip})^{p \in P_i}$ appearing in the pattern to values $(w_{ip})^{p \in P_i}$;
- the right-hand side matches Q_i given the values of the variables $(w_{ip})^{p \in P_i}$

More formally, the interpretation of the ornament is the following:

$$\mathcal{V}_q[\chi\,(\omega_j)^j\ (\omega_k)^k]_\gamma = \left\{ (d_i(v_{n-})^n, v_+) \;\middle|\; \begin{array}{l} i \in I \\ \wedge\ \forall n \in N_i, (v_{n-}, v_{n+}) \in \mathcal{V}_q[\omega_{in}[(\alpha_j \leftarrow \omega_j)^j, (\beta_k \leftarrow \omega_k)^k]]_\gamma \\ \wedge\ \exists (v_p)^{p \in P_i}, \\ \quad (\forall n \in N_i, v_{n+} = P_i[x_{ip} \leftarrow v_p]^p) \\ \quad \wedge v_+ \in \mathsf{Values}_{(x_{ip} \leftarrow v_p)^{p \in P_i}}(Q_i) \end{array} \right\}$$

Our example becomes (after simplification):

$$\begin{array}{rcl} \mathcal{V}_q[\widetilde{\mathsf{t}}\ \omega]_\gamma & = & \left\{ (\mathsf{T}\ w_-, \hat{\mathsf{T}}\ v_-) \;\middle|\; (w_-, w_+) \in \mathcal{V}_q[\omega]_\gamma \right\} \\ \mathcal{V}_q[\mathsf{fixed}]_\gamma & = & \{(\mathsf{None}, \mathsf{Nil})\} \cup \\ & & \left\{ (\mathsf{Some}\ w_-, v_+) \;\middle|\; \begin{array}{l} (w_-, w_+) \in \mathcal{V}_q[\widetilde{\mathsf{t}}\ \ (\mathsf{natlist\ bool})]_\gamma \\ \wedge\ w_+ = \hat{\mathsf{T}}\ w' \\ \wedge\ v_+ \in \mathsf{Values}_{x \leftarrow w'}(\mathsf{Cons}\ (_, x)) \end{array} \right\} \end{array}$$

Inlining the first definition into the second we obtain:

$$\begin{array}{rcl} \mathcal{V}_q[\mathsf{fixed}]_\gamma & = & \{(\mathsf{None}, \mathsf{Nil})\} \cup \\ & & \left\{ (\mathsf{Some}\ (\mathsf{T}\ w_-), v_+) \;\middle|\; \begin{array}{l} (w_-, w_+) \in \mathcal{V}_q[\mathsf{natlist\ bool}]_\gamma \\ \wedge\ v_+ \in \mathsf{Values}_{x \leftarrow w_+}(\mathsf{Cons}\ (_, x)) \end{array} \right\} \end{array}$$

10.5 From high-level definition to low-level definition

We build the encoding of ornaments from shallow definitions but we allow the user to specify ornaments using the higher-level deep ornamentation syntax. In this section, we will describe the transformation from deep to shallow ornaments and prove that a high-level definition and its low-level translation have the same interpretation: this is enough to show that they are equivalent.

The transformation operates on every ornament definition independently, and never generates constraints on the base type of ornament parameters. Consider an ornament definition of the form:

$$\mathsf{type\ ornament}\ \chi\,(\alpha_j)^j : \zeta\,(\tau_k)^{k \in K} \Rightarrow \sigma_+\ \mathsf{with}\ (P_i \Rightarrow Q_i\ \mathsf{when}\ (x_{i\ell} : \omega_{i\ell})^{\ell \in I_\ell})^{i \in I}$$

Our goal is to find the ornament arguments that must be passed to the skeleton and transform the left-hand side patterns into patterns that match already-ornamented values.

We first require that every pattern in $(P_i)^i$ starts with a constructor of ζ, *i.e.* for all $i \in I$, there exists d_i and $(P_{in})^{n \in N_i}$ such that $P_i = d_i(P_{in})^{n \in N_i}$. This can always be ensured by splitting the pattern: if the only pattern is $x \Rightarrow d(x)$, we can instead match on every constructor of ζ.

Then, we need to find ornament types $(\omega_{mn})^{m \in M, n \in N_m}$ for the different arguments of the skeleton. We suppose these types given, and we give a procedure to check them. The implementation instead performs unification to build these types. The idea is to transform P_i into a pattern $\hat{P}_i^{'}$ where all constructors have been replaced by their skeleton, and check the ornament type at the same type. We define a function $\mathsf{SkelPat}(P : \omega)_\Sigma$, where Σ is $(x_i : \omega_i)^i$ and associates variables in the pattern to ornaments, as follows:

$$\begin{array}{lcl} \mathsf{SkelPat}(x : \omega)_\Sigma & = & x \quad \text{if } (x : \omega) \in \Sigma \\ \mathsf{SkelPat}(d_i \,\overline{\tau}\, (P_k)^k : \widetilde{\omega}\ (\omega_{ik})^{ik})_\Sigma & = & \hat{d}_i((\omega_{ik})^+)^{ik}(\mathsf{SkelPat}(P_k : \omega_{ik})_\Sigma)^k \end{array}$$

This function satisfies the following properties:

Lemma 10.3. *Suppose $Q = \mathsf{SkelPat}(P : \omega)_\Sigma$, with P a pattern matching a subset $(x_i)^i$ of Σ. Then:*

- *We have $\vdash (\omega)^+ : Q \rightsquigarrow (x_i : (\Sigma(x_i))^+)^i$.*
- *Suppose we have $(v_{i-})^i$ and $(v_{i+})^i$, with $(v_{i-}, v_{i+}) \in \mathcal{V}_q[\Sigma(x_i)]_\gamma$. Then, $(P[x_i \leftarrow v_{i-}]^i, Q[x_i \leftarrow v_{i+}]^i) \in \mathcal{V}_q[\omega]_\gamma$.*
- *Consider v_- and $(v_{i+})^i$ such that $(v_-, Q[x_i \leftarrow v_{i+}]^i) \in \mathcal{V}_q[\omega]_\gamma$. Then there exists $(v_{i-})^i$ such that $(v_{i-}, v_{i+}) \in \mathcal{V}_q[\Sigma(x_i)]_\gamma$ and $v_- = P[x_i \leftarrow v_{i-}]^i$.*

Proof. By induction on the pattern.

- If the pattern is $P = x$, then $Q = x$ and $\Sigma(x_i) = \omega$. Then we have $\vdash (\omega)^+ : x \rightsquigarrow x : (\Sigma(x_i))^+$. The second and third item are immediate.
- If the pattern is a constructor, looking at the definition of $\widetilde{\zeta}$, the relation only matches constructors to matching constructors, and we use the induction hypothesis on each argument.

□

We define $P_i' = \hat{d}_i(P_{in}')^{n \in N_i}$ with $P_{in}' = \mathsf{SkelPat}(P_{in} : \omega_{in})_{(x_{il} : \omega_{il})^{\ell \in L_i}}$. Then we define the shallow ornament χ' as follows:

$$\begin{array}{l} \mathsf{type\ ornament}\ \chi'\ (\alpha_j)^j : \zeta\ (\tau_k)^k \Rightarrow \sigma_+ \ \mathsf{with} \\ \qquad [d_m(\omega_{mn})^{n \in M_n}]^{m \in M} \\ \quad (P_i' \Rightarrow Q_i)^{i \in I} \end{array}$$

The first point of Lemma 10.3 ensures this is a well-formed shallow ornament. We prove that they have the same interpretation:

Lemma 10.4 (The translation from deep to shallow preserves the relation)**.** *Suppose the $(\omega_j)^j$ are valid ornaments. Then, $\mathcal{V}_q[\chi\ (\omega_j)^j]_\gamma = \mathcal{V}_q[\chi'\ (\omega_j)^j]_\gamma$*

Proof. Suppose $(v_-, v_+) \in \mathcal{V}_q[\chi\,(\omega_j)^j]_\gamma$. Suppose we match pattern i. Then, by definition, there exists $(v_{\ell-}, v_{\ell+}) \in \mathcal{V}_q[\omega_{i\ell}[\alpha_j \leftarrow \omega_j]^j]_\gamma$ such that $v_- = d_i(P_{im}[x_{i\ell} \leftarrow v_{\ell-}]^\ell)^m$, and $v_+ \in \mathsf{Values}_{(x_{i\ell} \leftarrow v_{\ell+})^\ell}(Q_i)$. Then, consider $v_{m-} = P_{im}[x_{i\ell} \leftarrow v_{\ell-}]^\ell$ and $v_{m+} = P'_{im}[x_{i\ell} \leftarrow v_{\ell+}]^\ell$. We need to show that $(v_{m-}, v_{m+}) \in \mathcal{V}_q[\omega_{im}[\alpha_j \leftarrow \omega_j]^j]_\gamma$: this suffices to have $(v_-, v_+) \in \mathcal{V}_q[\chi'\,(\alpha_j)^j]_\gamma$. This is true by Lemma 10.3

Conversely, suppose we have $(v_-, v_+) \in \mathcal{V}_q[\chi'\,(\alpha_j)^j]_\gamma$. Then, there exists i such that $v_- = d_i(v_{m-})^m$, for all m, $(v_{m-}, v_{m+}) \in \mathcal{V}_q[\omega_{im}[\alpha_j \leftarrow \omega_j]^j]_\gamma$, and there exists $(v_p)^p$ such that $v_{m+} = P'_{im}[x_{ip} \leftarrow v_p]^p$, and $v_+ \in \mathsf{Values}_{(x_{ip} \leftarrow v_p)^\ell}(Q_i)$. Then, it suffices to show that $(v_{m-}, P'_{im}[x_{ip} \leftarrow v_p]^p) \in \mathcal{V}_q[\omega_{im}[\alpha_j \leftarrow \omega_j]^j]_\gamma$ implies that there exists $(v_{p_})^p$ such that $v_{m-} = P_{im}[x_{ip} \leftarrow v_{p_}]^p$ and for all p, $(v_{p_}, v_p) \in \mathcal{V}_q[\omega_{ip}[\alpha_j \leftarrow \omega_j]^j]_\gamma$. This is also true by Lemma 10.3

□

10.6 Encoding ornaments in mML

We now describe the encoding of datatype ornaments in mML. Consider a valid ornament definition:

$$\begin{array}{l}\mathsf{type\ ornament}\ \chi\,(\alpha_j)^j\ (\beta_k \lhd \tau_k)^k : \zeta\,(\sigma_\ell)^\ell \Rightarrow \sigma_+\ \mathsf{with}\\ \qquad [d_m(\omega_{mn})^{n\in M_n}]^{m\in M}\\ \quad (P_i \Rightarrow Q_i)^{i\in I}\end{array}$$

All terms given in this section are actually parametric in the input types $(\alpha_j)^j$ and $(\beta_k)^k$.

We call $\hat{\tau}_+$ the type $\hat{\zeta}\,(\omega^+_{mn})^{m\in M, n\in M_n}$, *i.e.* the type of a subterm as it is given to the construction function, after its components have been ornamented according to the specification given in the ornament definition.

The ornament is encoded as a quadruple $(\sigma, \delta, \mathsf{proj}, \mathsf{inj})$ where $\sigma : \mathsf{Typ}$ is the lifted type; δ is the *extension*, a type-level function describing the information that needs to be added to create a value of the lifted type from a value of $\hat{\tau}_+$; and proj and inj are the projection and injection functions introduced in §3.1. More precisely, the projection function proj from the lifted type to the skeleton has type $\Pi(x : \sigma)\ \hat{\tau}_+$ and, conversely, the injection inj has type $\Pi(x : \hat{\tau}_+)\ \Pi(y : \delta \# x)\ \sigma$, where the argument y is the additional information necessary to build a value of the lifted type. The type of y is given by the *extension* type function δ of kind $\hat{\tau}_+ \to \mathsf{Typ}$, which takes the skeleton and gives the type of the missing information. This dependence allows us to add different pieces of information for different shapes of the skeleton, *e.g.* in the case of $\mathsf{natlist}\ \alpha$, we need no additional information when the skeleton is $\hat{\mathsf{Z}}$, but a value of type α when the skeleton starts with $\hat{\mathsf{S}}$. The encoding works incrementally: all functions manipulate the type $\hat{\tau}_+$, with all subterms already ornamented.

The projection $\mathsf{proj}_{\chi\,(\omega_j)^j}$ from the lifted type to the skeleton is given by reading the clauses of the ornament definition from right to left:

$$\mathsf{proj}_{\chi\,(\omega_j)^j} \ :\ \sigma_{\chi\,(\omega_j)^j} \to \hat{\tau}_+ \ \triangleq\ \lambda^\#(x : \sigma_{\chi\,(\omega_j)^j}).\ \mathsf{match}\ x\ \mathsf{with}\ (Q_i \to \hat{P}_i)^i$$

The extension $\delta_{\chi\,(\omega_j)^j}$ is determined by computing, for each clause $P_i \Rightarrow Q_i$ of the ornament, the type of the information missing to reconstruct a value. There are many isomorphic representations of this information. The representation we

$$
\begin{array}{lcl}
\textsc{Missing}((_ : \tau)) & = & \tau \\
\textsc{Missing}(x) & = & \mathsf{unit} \\
\textsc{Missing}(P \mid Q) & = & \mathsf{either}\ (\textsc{Missing}(P), \textsc{Missing}(Q)) \\
\textsc{Missing}(\bot) & = & \mathsf{void} \\
\textsc{Missing}(d(P_1, .. P_n)) & = & \textsc{Missing}(P_1) \times .. \textsc{Missing}(P_n)
\end{array}
$$

Figure 10.8: Missing part of a pattern

$$
\begin{array}{lcl}
\textsc{Patch}(x, u) & = & x \\
\textsc{Patch}(_, u) & = & u \\
\textsc{Patch}(\bot, u) & = & \mathsf{Unit} \\
\textsc{Patch}(P \mid Q, u) & = & \mathsf{match}\ u\ \mathsf{with}\ \mathsf{Left}\ x \to \textsc{Patch}(P, x) \mid \mathsf{Right}\ x \to \textsc{Patch}(Q, x) \\
\textsc{Patch}(d(P_i)^i, u) & = & \mathsf{match}\ u\ \mathsf{with}\ (x_i)^i \to d(\textsc{Patch}(P_i, x_i))^i
\end{array}
$$

Figure 10.9: Patching a pattern

use is given by the function $\textsc{Missing}(Q_i)$ mapping a pattern to a type, defined in Figure 10.8. There is no missing information in the case of variables, since they correspond to variables on the left-hand side. In the case of constructors, we expect the missing information corresponding to each subpattern, given as a tuple. For wildcards, we expect a value of the type matched by the wildcard. The empty pattern matches nothing: we cannot add any information to get a term matching the pattern, so the missing information has the void type. Finally, for an alternative pattern, we require to choose between the two sides of the alternative and give the corresponding information, representing this as a sum type $\mathsf{either}\ (\tau_1, \tau_2)$.

From the value of the variables and the missing parts, we can patch the holes in the pattern and reconstruct the value that was matched: we define an expression $\textsc{Patch}(P, u)$ that does this in Figure 10.9. This expresion matches on u to extract the information. In the case of $\bot$, we can return anything, because u must be a value of type void, which means that this branch is unreachable. The correctness of the combination of $\textsc{Missing}(P)$ and $\textsc{Patch}(P, u)$ is stated by the following lemma:

Lemma 10.5. *Suppose* $\Gamma \vdash \sigma : \mathsf{Typ}$ *and* $\vdash P : \sigma \rightsquigarrow (x_i : \tau_i)^i$. *Then,* $\Gamma \vdash \textsc{Missing}(P) : \mathsf{Typ}$*: the missing type is a valid type. Moreover, if* $\Gamma \vdash u : \textsc{Missing}(P)$*, then we have* $\Gamma, (x_i : \tau_i)^i \vdash \textsc{Patch}(P, u) : \sigma$*: patching creates a well-typed non-expansive term of the correct type. Finally, patching then matching gives back the same values for the variables:*

$$\Gamma, (x_i : \tau_i)^i \vdash \textsc{Pat}(P, \textsc{Patch}(P, u)) : \mathsf{option}\ (\tau_i)^i \simeq \mathsf{Some}\ (x_i)^i : \mathsf{option}\ (\tau_i)^i$$

Proof. By induction on the pattern. At each step, split on the non-expansive value u, apply the lemma recursively and then reduce the matches. □

We could use any pair of functions that satisfies these properties: in our implementation, we generate simpler type by avoiding empty types in sums, units in products, flattening products, etc.

Then, the extension $\delta_{\chi\,(\omega_j)^j}$ matches on the $(\hat{P}_i)^i$ to determine which clause of the ornament definition can handle the given skeleton, and returns the cor-

responding extension type:

$$\delta_{\chi\,(\omega_j)^j} \;:\; \Pi(x:\sigma)\,\hat{\tau}_+ \;\triangleq\; \lambda^{\#}(x:\hat{\tau}_+).\,\mathsf{match}\ x\ \mathsf{with}\ (\hat{P}_i \to \textsc{Missing}(Q_i))^i$$

The injection $\mathsf{inj}_{\chi\,(\omega_j)^j}$ then examines the skeleton to determine which clause of the ornament to apply, and calls the corresponding reconstruction code (writing just δ for $\delta_{\chi\,(\omega_j)^j}$):

$$\begin{array}{l}\mathsf{inj}_{\chi\,(\omega_j)^j} : \Pi(x:\hat{\tau}_+)\,\Pi(y:\delta\ \#\ x)\,T \triangleq \\ \quad \lambda^{\#}(x:\hat{\tau}_+).\,\lambda^{\#}(y:\delta\ \#\ x).\,\mathsf{match}\ x\ \mathsf{with}\ (\hat{P}_i \to \textsc{Patch}(Q_i, y))^i\end{array}$$

In the case of $\mathsf{natlist}$, we recover the definitions given in §3.1.2, with a slightly more complex (but isomorphic) encoding of the extra information:

$$\begin{array}{rcl}
\sigma_{\mathsf{natlist}\,\tau} & = & \mathsf{list}\ \tau \\
\delta_{\mathsf{natlist}\,\tau} & = & \lambda^{\#}(x:\widehat{\mathsf{nat}}(\mathsf{list}\ \tau)).\,\mathsf{match}\ x\ \mathsf{with}\ \hat{\mathsf{Z}} \to \mathsf{unit} \mid \hat{\mathsf{S}}\ x \to \tau \times \mathsf{unit} \\
\mathsf{proj}_{\mathsf{natlist}\,\tau} & = & \lambda^{\#}(x:\mathsf{list}\ \tau).\,\mathsf{match}\ x\ \mathsf{with}\ \mathsf{Nil} \to \hat{\mathsf{Z}} \mid \mathsf{Cons}\ (y, _) \to \hat{\mathsf{S}}\ y \\
\mathsf{inj}_{\mathsf{natlist}\,\tau} & = & \lambda^{\#}(x:\widehat{\mathsf{nat}}(\mathsf{list}\ \tau)).\,\lambda^{\#}(y:\delta_{\mathsf{natlist}\,\tau}\ \#\ x). \\
& & \quad \mathsf{match}\ y\ \mathsf{with}\ \hat{\mathsf{Z}} \to (\mathsf{match}\ y\ \mathsf{with}\ () \to \mathsf{Nil}) \\
& & \qquad\qquad \mid \hat{\mathsf{S}}\ x' \to (\mathsf{match}\ y\ \mathsf{with}\ (y', ()) \to \mathsf{Cons}\ (y', x'))
\end{array}$$

We also need to be able to instantiate ornamentation points with the identity ornament, so we need to define the same functions for them: the identity ornament corresponding to a datatype ζ defined as $(d_i : \forall(\alpha_j : \mathsf{Typ})^j\ (\tau_{ik})^k \to \zeta\,(\alpha_j)^j)^i$ is automatically generated and is described by the following code:

$$\begin{array}{l}\mathsf{type\ ornament}\ \zeta\,(\alpha_j)^j : \zeta\,(\alpha_j)^j \to \zeta\,(\alpha_j)^j\ \mathsf{with} \\ \qquad [(d_i\ (\tau_{ik})^k)^i] \\ \quad (d_i(x_k)^k \to d_i(x_k)^k)^i\end{array}$$

The generated definition is then:

$$\begin{array}{rcl}
\sigma & = & \zeta\,(\tau_j)^j \\
\delta & = & \lambda^{\#}(x:\hat{\zeta}\ (\tau_{ik})^{ik}).\,\mathsf{match}\ x\ \mathsf{with}\ (\hat{d}_i(x_{ik})^k \to \mathsf{unit})^i \\
\mathsf{proj} & = & \lambda^{\#}(x:\zeta\,(\alpha_j)^j).\,\mathsf{match}\ x\ \mathsf{with}\ (d_i(x_{ik})^k \to \hat{d}_i(x_{ik})^k)^i \\
\mathsf{inj} & = & \lambda^{\#}(x:\hat{\zeta}\ (\tau_{ik})^{ik}).\,\mathsf{match}\ x\ \mathsf{with}\ (\hat{d}_i(x_{ik})^k \to d_i(x_{ik})^k)^i
\end{array}$$

10.7 Correctness of the encoding

In this section, we show that the terms defined in the previous section do correspond to the ornament as interpreted by the logical relation. This will be used to prove correctness of the lifting, *i.e.* that the base term and the ornamented term are related at some ornament type.

The base term will be proved equal to the ornamented term instantiated with the identity ornament. Thus, what we need to do here is to prove that the identity ornament and an ornament that does something are related at the ornament type.

Consider an ornament definition:

$$\begin{array}{l}\mathsf{type\ ornament}\ \chi\,(\alpha_j)^j\ (\beta_k \lhd \tau_k)^k : \zeta\,(\sigma_\ell)^\ell \Rightarrow \sigma_+\ \mathsf{with} \\ \qquad [d_m(\omega_{mn})^{n\in N_m}]^{m\in M} \\ \quad (P_i \Rightarrow Q_i)^i\end{array}$$

The ornament is encoded in $m\mathsf{ML}$ by $(\sigma, \delta, \mathsf{proj}, \mathsf{inj})$. Consider the identity ornament for $\zeta\,(\sigma_\ell)^\ell$: it is encoded as $(\sigma_{\mathsf{id}}, \delta_{\mathsf{id}}, \mathsf{proj}_{\mathsf{id}}, \mathsf{inj}_{\mathsf{id}})$, with $\sigma_{\mathsf{id}} = \zeta\,(\sigma_\ell)^\ell$. Consider the ornamented skeleton type $\hat{\tau} = \hat{\zeta}(\sigma_\ell)^\ell$, and an environment $\gamma \in \mathcal{G}_k[(\alpha_j : \mathsf{Typ})^j, (\beta_k : \mathsf{Typ})^j]$. We define $\widetilde{\delta} = \lambda(v_1, v_2).\ \{(\mathsf{Unit}, w_2) \mid \varnothing \vdash w_2 : \gamma_2(\delta) \# v_2\}$ the relational version of δ: for any pair of base and ornamented skeletons, it returns the relation that relates Unit, the extension type of the identity ornament, to any value of the extension type of the ornament.

Definition 10.2 (Valid ornament)**.** *A valid ornament is an ornament that verifies the following properties:*

- $(\gamma_1(\zeta\,(\sigma_\ell)^\ell), \gamma_2(\sigma)) = \mathcal{T}[\gamma]_\tau$
- $\widetilde{\delta} \in \mathcal{V}_q[\Pi(x : \hat{\tau})\ \mathsf{Typ}]_\gamma$
- $(\gamma_1(\mathsf{proj}_{\mathsf{id}}), \gamma_2(\mathsf{proj})) \in \mathcal{V}_q[\Pi(x : \chi\,(\alpha_j)^j\ (\beta_k)^k)\ \hat{\tau}]_\gamma$.
- $(\gamma_1(\mathsf{inj}_{\mathsf{id}}), \gamma_2(\mathsf{inj})) \in \mathcal{V}_q[\Pi(x : \hat{\tau})\ \Pi y : \alpha \# x\ \chi\,(\alpha_j)^j\ (\beta_k)^k]_{\gamma, \alpha \leftarrow \widetilde{\delta}}$

$\diamondsuit$

These properties are simply relational versions of the typing rules for proj and inj: they ensure that we can relate two instances of the same term, one instantiated with the identity ornament and the other with the actual ornament.

Theorem 10.1. *The construction given in this chapter produces a valid ornament.*

Proof. The first item is by definition.

For the second one, note that $\{(\mathsf{Unit}, w_2) \mid \varnothing \vdash w_2 : \gamma_2(\delta) \# v_2\}$ is stable by equality and does not vary with the index.

For the third and fourth point: we'll examine only the third one, the fourth one is similar. $\gamma_1(\mathsf{proj}_{\mathsf{id}})$ and $\gamma_2(\mathsf{proj})$ are values. Consider an index q and $(v_1, v_2) \in \mathcal{V}_q[\chi\,(\alpha_j)^j(\beta_k)^k]_\gamma$. Then (v_1, v_2) matches some clause i of the ornament whose head constructor is d_i, and there exists $(v_{m-}, x_{m+})^m$ such that $(v_{m-}, x_{m+}) \in \mathcal{V}_q[\omega_{im}]_\gamma$ and $v_1 = d_i(v_{m-})^m$. Then, $\gamma_1(\mathsf{proj}_{\mathsf{id}}) \# v_1$ reduces to $\hat{d}_i(v_{m-})^m$.

There also exists $(v_p)^p$ such that $v_+ \in \mathsf{Values}_{(x_{ip} \leftarrow v_p)^p}(Q_i)$. Then, $\gamma_2(\mathsf{proj}) \# v_+$ reduces to $\hat{P}_i[x_{ip} \leftarrow v_p]^p$ (this matches the i-th pattern, and the i-th pattern is the only one that matches, because the patterns do not overlap). This rewrites to $\hat{d}_i(P_{im}[x_{ip} \leftarrow v_p]^p)^m$

Thus, we need to prove $(\hat{d}_i(v_{m-})^m, \hat{d}_i(P_{im}[x_{ip} \leftarrow v_p]^p)^m) \in \mathcal{V}_q[\hat{\zeta}(\omega_{im})^{im}]_\gamma$, *i.e.* for all m, $(v_{m-}, P_{im}[x_{ip} \leftarrow v_p]^p) \in \mathcal{V}_q[\omega_{im}]_\gamma$. From the definition of the relation for χ, we have $P_{im}[x_{ip} \leftarrow v_p]^p = v_{m+}$, and we have $(v_{m-}, x_{m+}) \in \mathcal{V}_q[\omega_{im}]_\gamma$. $\square$

Once processed, all well-formed ornament definitions are added to a global ornament definition: we will write

$$\vdash \chi\ \overline{\alpha}\ \ \overline{\beta \lhd \tau} \mapsto (\delta_{\chi\ \overline{\alpha}\ \overline{\beta}}, \mathsf{inj}_{\chi\ \overline{\alpha}\ \overline{\beta}}, \mathsf{proj}_{\chi\ \overline{\alpha}\ \overline{\beta}}) \lhd \hat{\omega}'_i : \sigma'_i \Rightarrow T$$

to indicate that a definition of ornament χ has been processed with the given parameters, base type, and skeleton. We also write $\delta_{\chi\ \overline{\omega_j}}$, $\mathsf{inj}_{\chi\ \overline{\omega_j}}$ and $\mathsf{proj}_{\chi\ \overline{\omega_j}}$ to stand for instantiated versions of the extension types, construction, and projection functions defining χ.

Chapter 11

Elaborating to the generic term

We now consider the problem of ornamenting terms. The ornamentation is done in two main steps: first the base term is elaborated to a generic term, which is then specialized using user-specified ornaments to generate ML code.

11.1 Preparing an ML term for lifting

When quantifying over ornaments, one expects to receive some pieces of mML code that need to be applied, patched and reduced to produce a valid ML term. Thus, we cannot simplify code that is polymorphic in ornaments back to eML: the lifted code cannot be polymorphic in ornaments.

Consider a polymorphic definition, for example $\mathsf{head} : \forall(\alpha : \mathsf{Typ})\,.\mathsf{list}\ \alpha \to \alpha$. We do not know the exact types of the ornaments we may use until the type variable α has been instantiated. We could specify some ornaments depending on the values of α: for example, we could apply the ornament $\mathsf{listoption}\ \alpha : \mathsf{list}\ \alpha \Rightarrow \mathsf{option}\ \alpha$. However, this prevents us from applying some other interesting ornaments, such as the ornament that transforms $\mathsf{Cons}\ (\mathsf{True}, x)$ to $\mathsf{Cons}_A\ (x)$ and $\mathsf{Cons}\ (\mathsf{False}, x)$ to $\mathsf{Cons}_B\ (x)$ (in OCaml, applying such a transformation saves memory, as the latter type only needs two words instead of three). Thus our approach is to instantiate the type variables, then choose some appropriate ornaments, then generalize to produce a lifted definition. Then, we have produced a lifting compact_head of head bool, but not a general lifting of head.

At the point where head is used, we need to look at its type argument to decide whether compact_head is an appropriate lifting. This treatment requires some restrictions on when instantiation and generalization are performed. We choose to restrict ML polymorphism to top-level bindings and to require all uses of polymorphic values to immediately instantiate them fully. For local polymorphic bindings, we know the set of instantiations of the term: we can thus simply duplicate the term to create one instance per use with different type arguments. This does not affect usability much, as polymorphic let bindings are rarely used [Vytiniotis et al., 2010].

To encode the restriction to toplevel polymorphism, we need to make a distinction between (generalizable) toplevel bindings and monomorphic local

Γ ::=		Environment
	$\mid G, \overline{\alpha}, \Theta$	
G ::=		Global environment
	$\mid \varnothing$	Empty environment
	$\mid G, x\ \overline{\alpha} : \tau = a$	Global binding, with value
Θ ::=		Local environment
	$\mid \varnothing$	Empty environment
	$\mid \Theta, x : \tau$	Local binding

Figure 11.1: Environments for ML restricted to top-level polymorphism

$$\textsc{Wf-G-Empty}\quad \vdash \varnothing \qquad\qquad \textsc{Wf-G-Def}\quad \frac{\vdash G \qquad G; \overline{\alpha}; \varnothing \vdash a : \tau}{\vdash G, x\ \overline{\alpha} : \tau = a}$$

Figure 11.2: Well-formedness for global environments

bindings. The environment Γ can then be split into $G, (\alpha_i : \mathsf{Typ})^i, \Theta$ where G is an environment of polymorphic variable bindings, $\overline{\alpha}$ the list of type variables of kind Typ parametrizing the current binding, and Θ a local environment binding only monomorphic term variables. The syntax of these environments is given in Figure 11.1. The definitions in G are of the form $x\ \overline{\beta} : \tau = a$. Such a definition means that x expects type parameters $\overline{\beta}$ and then is of (monomorphic) type τ with value a. We store the value in the environment because it will be convenient to prove that the lifting is correct. To save notation, we just write α instead of $\alpha : \mathsf{Typ}$ in typing contexts or polymorphic types, assuming that type variables have the Typ kind by default. The well-formedness judgment for G, noted $\vdash G$, is defined in Figure 11.2.

We give the typing rules for this restriction of ML in Figure 11.3. The typing judgment is now of the form $G; \overline{\alpha}; \Theta \vdash a : \tau$ (the input type is in ML, and its typing features neither labels nor equalities). The modified rules are in black, the rules that were only changed to pass a split environment are in grey. Types are only checked in the environment $\overline{\alpha}$, since they can only contain toplevel type variables. The rule Var-Local looks up a variable in the monomorphic environment, and the rule Var-Global looks up a toplevel definition in the polymorphic environment. We removed rules for type abstraction and application.

Generalization is handled through a separate judgment: we consider a program as a sequence s of top-level definitions $\mathsf{let}\ x\ \overline{\alpha} = a$. Such a sequence is checked by a judgment $G \vdash s \Rightarrow G'$: after typing s in the top-level environment G, the top-level environment is now G'. The rules of this judgment are given in Figure 11.4. Then, the rule Top-Let quantifies over the type variables $(\alpha_i)^i$ before typechecking the body a.

$$\textsc{Var-Local}\quad \frac{(x : \tau) \in \Theta}{G; \overline{\alpha}; \Theta \vdash x : \tau}$$

$$\textsc{Var-Global}\quad \frac{(x\ (\beta_j)^j : \sigma) \in G \qquad (\overline{\alpha} \vdash \tau_j : \mathsf{Typ})^j}{G; \overline{\alpha}; \Theta \vdash x\ (\tau_j)^j : \sigma[\beta_j \leftarrow \tau_j]^j}$$

$$\textsc{Let}\quad \frac{\overline{\alpha} \vdash \tau : \mathsf{Typ} \qquad G; \overline{\alpha}; \Theta \vdash a : \tau \qquad G; \overline{\alpha}; \Theta, x : \tau \vdash b : \sigma}{G; \overline{\alpha}; \Theta \vdash \mathsf{let}\ x = a\ \mathsf{in}\ b : \sigma}$$

$$\textsc{Fix}\quad \frac{\Gamma \vdash \tau_1 : \mathsf{Typ} \qquad \Gamma \vdash \tau_2 : \mathsf{Typ} \qquad G; \overline{\alpha}; \Theta, x : \tau_1 \to \tau_2, y : \tau_1 \vdash a : \tau_2}{G; \overline{\alpha}; \Theta \vdash \mathsf{fix}\, x\ (y : \tau_1) : \tau_2\ .\ a : \tau_1 \to \tau_2}$$

$$\textsc{App}\quad \frac{G; \overline{\alpha}; \Theta \vdash b : \tau_1 \qquad G; \overline{\alpha}; \Theta \vdash a : \tau_1 \to \tau_2}{G; \overline{\alpha}; \Theta \vdash a\ b : \tau_2}$$

$$\textsc{Con}\quad \frac{\begin{array}{c} \vdash d : \forall(\alpha_j : \mathsf{Typ})^j\ (\tau_i)^i \to \zeta\ (\beta_j)^j \\ (\overline{\alpha} \vdash \tau_j : \mathsf{Typ})^j \qquad (G; \overline{\alpha}; \Theta \vdash a_i : \tau_i[\beta_j \leftarrow \tau_j]^j)^i \end{array}}{G; \overline{\alpha}; \Theta \vdash d(\tau_j)^j (a_i)^i : \zeta\ (\tau_j)^j}$$

$$\textsc{Match}\quad \frac{\begin{array}{c} \overline{\alpha} \vdash \tau : \mathsf{Typ} \qquad (d_i : \forall(\beta_k : \mathsf{Typ})^k\ (\tau_{ij})^j \to \zeta\ (\beta_k)^k)^i \qquad G; \overline{\alpha}; \Theta \vdash a : \zeta\ (\tau_k)^k \\ (G; \overline{\alpha}; \Theta, (x_{ij} : \tau_{ij}[\beta_k \leftarrow \tau_k]^k)^j \vdash b_i : \tau)^i \qquad \vdash (d_i)^i : \zeta\ \mathsf{complete} \end{array}}{G; \overline{\alpha}; \Theta \vdash \mathsf{match}\ a\ \mathsf{with}\ (d_i(\tau_k)^k (x_{ij})^j \to b_i)^i : \tau}$$

Figure 11.3: Typing rules for ML restricted to top-level polymorphism

$$\textsc{Top-Let}\quad \frac{G; (\alpha_i : \mathsf{Typ})^i; \varnothing \vdash a : \tau}{G \vdash \mathsf{let}\ x\ (\alpha_i)^i = a \Rightarrow G, x\ (\alpha_i)^i : \tau = a}$$

$$\textsc{Top-Seq}\quad \frac{G_0 \vdash s_1 \Rightarrow G_1 \qquad G_1 \vdash s_2 \Rightarrow G_2}{G_0 \vdash s_1; s_2 \Rightarrow G_2}$$

$$\textsc{Top-Empty}\quad \frac{}{G \vdash \varnothing \Rightarrow G}$$

Figure 11.4: Processing of top-level definitions

11.2 Elaboration environments

The elaboration process is run in parallel with the typing process: each toplevel definition is elaborated in the order it appears in the program, right after being typechecked. As for typechecking, the information gathered when processing previous definitions is used to elaborate the next definition.

For each toplevel definition "$\mathsf{let}\ x = \Lambda\overline{\alpha}.\ a$", we elaborate a using the term elaboration judgment of the form $\Gamma \vdash^p a \rightsquigarrow A : \omega \Rightarrow \Delta$ (described in Figure 11.9), which implies both $\Gamma^- \vdash^{\mathsf{ML}} a : \omega_\Gamma^-$ and $\Gamma^+ \vdash^p A : \omega_\Gamma^+ \Rightarrow \Delta$ (Lemma 11.1). We use capital letters (A for terms, T for types) to denote terms and types from the generic side.

Similarly to ML with restricted polymorphism, the elaboration environment Γ, described in Figure 11.5, is composed of multiple environments $G, \overline{\alpha}, S, R, \Theta$, As in the ML typing rules, G stores the global definitions with some information derived from elaboration we will describe later. $\overline{\alpha}$ is the list of type variables the term was originally quantified over, and Θ contains the local bindings accumulated when typing the term. It binds the variables appearing in a to ornament types describing how the values of these variables are related in a and A. It also contains bindings that are only available in the ornamented term: these are equalities, noted $(u \simeq u' : \tau)^\#$ to highlight the fact that they are not available in a, as well as results $((u)*p : \tau)^\#$ and path variables $(\pi)^\#$. Similarly, the output Δ contains the list of results produced by the evaluation of the term, and is only available on the ornamented side. S and R, explained below, contain the ornaments and patches that are used by the generic term. Finally, ω describes the ornament relation between a and A once all parts of the environment have been instantiated.

A definition is typed (Elab-Decl in Figure 11.10) in the current G environment, quantified over its type variables $\overline{\alpha}$, in an initially empty local environment $\Theta = \varnothing$, and with S and R chosen by the lifting procedure (we will describe how they are chosen later). The result of the elaboration of a definition is then folded into the *global* environment G as a sequence of declarations of the form $x\langle\overline{\alpha}, S, R\rangle : \omega = a \rightsquigarrow A$.

The environments S and R are new and used to describe abstract ornaments and patches, respectively. The generic term A is usually more polymorphic than a, since we abstract over ornaments where we originally had a fixed type. It is thus parametrized by a number of ornaments, described by the *ornament specification* environment S which is a set of mutually recursive bindings, each of the form $\varphi \mapsto (\delta, \mathsf{proj}, \mathsf{inj}) \lhd \hat{\zeta}\,(\omega_k)^k : \zeta\,(\tau_i)^i \Rightarrow \beta$. This binds an ornament variable φ that can be instantiated by an ornament of base type $\zeta\,(\tau_i)^i$ with skeleton $\hat{\zeta}\,(\omega_k)^k$; it also binds the target type β and the ornament type extension, projection, and injection functions to the variables δ, proj, and inj. The skeleton is important here because it determines how the arguments of constructors must be ornamented.

The ornament variables φ can then be used in ornament types ω. They are considered distinct from the α because they represent datatype ornaments, and because the environment S specifies what their base type is. Ornament types ω with ornament variables are projected under a given environment S: φ is replaced on the left with the base type given in S, and on the right with the given type variable. The definitions of the left and right projections $(\omega)_S^-$ and $(\omega)_S^+$ are given in Figure 11.7. An ornament type ω is well-formed in an

Γ ::=		Elaboration environment
	$\mid G, \overline{\alpha}, S, R, \Theta$	
G ::=		Global environment
	$\mid \varnothing$	Empty
	$\mid G, x\langle\overline{\alpha}, S, R\rangle : \omega = a \rightsquigarrow A$	Global definition
Θ ::=		Local environment
	$\mid \varnothing$	Empty
	$\mid \Theta, x : \omega$	Variable
	$\mid \Theta, (u \simeq u : \tau)^{\#}$	Equality for the generic term
	$\mid \Theta, ((u)*p : \tau)^{\#}$	Result for the generic term
	$\mid \Theta, (\pi)^{\#}$	Path variable for the generic term
s ::=		Ornament instantiation
	$\mid \varnothing$	Empty
	$\mid s, \varphi \leftarrow \varphi$	Ornament
S ::=		Ornament requests
	$\mid \varnothing$	Empty
	$\mid S, \varphi \mapsto (\delta, \mathsf{proj}, \mathsf{inj}) \lhd \hat{\zeta}\,\overline{\omega} : \zeta\,\overline{\tau} \Rightarrow \alpha$	Ornament request
R ::=		Patch requests
	$\mid \varnothing$	Empty
	$\mid R, y :^{\#} \Pi(\Gamma).\,[\delta_\varphi \# u]$	Patch
	$\mid R, (x\langle\overline{\omega}, s\rangle \rightsquigarrow y : \omega)$	Lifting

Figure 11.5: Environments

$$\begin{aligned} \alpha^{\epsilon}_S &= \alpha \\ (\omega_1 \to \omega_2)^{\epsilon}_S &= (\omega_1)^{\epsilon}_S \to (\omega_2)^{\epsilon}_S \end{aligned} \qquad \frac{(\varphi \mapsto _ \lhd _ : \tau \Rightarrow \alpha) \in S}{\varphi^-_S = \tau \qquad \varphi^+_S = \alpha}$$

Figure 11.6: Projections for ornament types

$$\begin{aligned} (R, x :^{\#} T)^+_S &= R^+_S, x : T \\ (R, x\langle_, _\rangle \rightsquigarrow y : \omega)^+_S &= R^+_S, y : \omega^+_S \end{aligned}$$

$$\begin{aligned} (\Theta, x : \omega)^{\epsilon}_S &= \Theta^{\epsilon}_S, x : \omega^{\epsilon}_S \\ (\Theta, (u \simeq u' : \tau)^{\#})^-_S &= \Theta^-_S \\ (\Theta, (u \simeq u' : \tau)^{\#})^+_S &= \Theta^+_S, (u \simeq u') : \tau \end{aligned}$$

$$\begin{aligned} (G, \overline{\alpha}, S, R, \Theta)^- &= G^-, \overline{\alpha}, \Theta^-_S \\ (G, \overline{\alpha}, S, R, \Theta)^+ &= \overline{\alpha}, S^+, R^+_S, \Theta^+_S \\ (G, x\langle\overline{\alpha}, S, _\rangle : \omega = _ \rightsquigarrow _)^- &= G^-, x : \forall\,\overline{\alpha}\ \omega^-_S \end{aligned}$$

$$S^+ = \left\{\beta : \mathsf{Typ} \mid (\varphi \mapsto _ \lhd _ : _ \Rightarrow \beta) \in S\right\}, \left\{ \begin{array}{l} \delta : \hat{\omega}^+_S \to \mathsf{Typ}, \\ \mathsf{proj} : \Pi(x : \beta)\,\hat{\omega}^+_S, \\ \mathsf{inj} : \Pi(x : \hat{\omega}^+_S)\,\Pi(y : \delta \# x)\,\beta \end{array} \middle| \; (\varphi \mapsto (\delta, \mathsf{proj}, \mathsf{inj}) \lhd \hat{\omega} : _ \Rightarrow \beta) \in S \right\}$$

Figure 11.7: Environment projections

G-Empty
$$\vdash \varnothing$$

G-Def
$$\frac{\vdash G \qquad G, \overline{\alpha} \vdash S \qquad G, \overline{\alpha}, S \vdash \omega \text{ orn} \qquad (G, \overline{\alpha})^- \vdash a : \omega_S^- \qquad G, \overline{\alpha}, S \vdash R \qquad (G, \overline{\alpha}, S, R)^+ \vdash A : \omega_S^+}{\vdash G, x\langle \overline{\alpha}, S, R\rangle : \omega = a \rightsquigarrow A}$$

Wf-S
$$\frac{\forall(\varphi \mapsto _ \lhd \hat{\zeta}\,(\omega_k)^k : \zeta\,(\tau_j)^j \Rightarrow _) \in S, \quad (\overline{\alpha} \vdash \tau_j : \mathsf{Typ})^j \wedge (G, \overline{\alpha}, S \vdash \omega_k \text{ orn})^k \wedge ((\omega_k)_S^-)^k = \mathcal{A}_\zeta(\tau_j)^j}{G, \overline{\alpha} \vdash S}$$

Wf-R-Empty
$$G, \overline{\alpha}, S \vdash \varnothing$$

Wf-R-Patch
$$\frac{G, \overline{\alpha}, S \vdash R \qquad (\varphi \mapsto (\delta, \ldots) \lhd \hat{\omega} : _ \Rightarrow _) \in S \qquad (G, \overline{\alpha}, S, R)^+, \Theta_S^+ \vdash A : \hat{\omega}_S^+}{G, \overline{\alpha}, S \vdash R, y :^{\#} \Pi(\Theta_S^+).\, [\delta \# A]}$$

Wf-R-Inst
$$\frac{G, \overline{\alpha}, S \vdash R \qquad (x\langle(\beta_j)^j, S', R\rangle : \omega) \in G \qquad (G, \overline{\alpha}, S \vdash \omega_j \text{ orn})^j \qquad G, \overline{\alpha}, S \vdash s : S'[\beta_j \leftarrow \omega_j]^j}{G, \overline{\alpha}, S \vdash R, (x\langle(\omega_j)^j, s\rangle \rightsquigarrow y : \omega[(\beta_j \leftarrow \omega_j)^j, s])}$$

Wf-inst
$$\frac{\forall(\varphi \mapsto _ \lhd \hat{\omega} : \tau \Rightarrow _) \in S', (s(\varphi) \mapsto _ \lhd s(\hat{\omega}) : \tau \Rightarrow _) \in S}{G, \overline{\alpha}, S \vdash s : S'}$$

Wf-Orn-Arrow
$$\frac{G, \overline{\alpha}, S \vdash \omega_1 \text{ orn} \qquad G, \overline{\alpha}, S \vdash \omega_2 \text{ orn}}{G, \overline{\alpha}, S \vdash \omega_1 \rightarrow \omega_2 \text{ orn}}$$

Wf-Orn-TVar
$$\frac{\alpha \in \overline{\alpha}}{G, \overline{\alpha}, S \vdash \alpha \text{ orn}}$$

Wf-Orn-Var
$$\frac{\varphi \in S}{G, \overline{\alpha}, S \vdash \varphi \text{ orn}}$$

Wf-s
$$\frac{S'[s] \subseteq \Gamma}{\Gamma \vdash s : S'}$$

Figure 11.8: Well-formedness for elaboration

environment S with free type variables $\overline{\alpha}$, written $G, \overline{\alpha}, S \vdash \omega$ orn, if it contains only function arrows, type variables from $\overline{\alpha}$, and ornament variables bound in S (see rules Wf-Orn-Arrow, Wf-Orn-TVar, and Wf-Orn-Var, in Figure 11.5). Note that these environments do not contain concrete ornaments or identity ornaments: the generic term is made as generic as possible, and each concrete ornament can be replaced with an ornament variable with the same base type instead. The environment S is not ordered: each binding defines an ornament variable φ and a type variable β that can be used in the skeleton of any other ornament request. This is necessary to ornament recursive functions, and this is why its projection $(S)^+$ (Figure 11.6) first projects all type variables representing ornamented types before projecting the other parts of the environment.

A generic term also abstracts over patches and the liftings used to lift references to previously elaborated bindings. Since these bindings do not influence the final ornament type and are not mutually recursive, they are stored in a separate *patch* environment R. Together, S and R specify all the parts that have to be user-provided at specialization time (see Chapter 12).

The environment R contains lifting requests accumulated when processing the term. During elaboration, it is built as an *output* to the judgment: when a rule requires that something be in R, we simply add it to the output.

When the elaboration process encounters a variable x that corresponds to a global definition (Rule E-VarGlobal in Figure 11.9), we look up the signature of the elaboration of this definition $(x\langle\overline{\alpha}, S', R'\rangle : \omega) \in G$. We choose an instantiation $\overline{\omega}$ of the type parameters $\overline{\alpha}$ by ornament types, an instantiation s' of the ornament variables in S' with ornament variables of S (checked by the judgment $\Gamma \vdash s : S'[\overline{\alpha} \leftarrow \overline{\omega}]$, defined by Wf-s in Figure 11.8) , and request a value y corresponding to an instantiation of the function with the chosen type and ornament parameters (we do not ask for a specific instantiation of the patches and liftings in R' as they do not contribute to the lifting specification) We record this instantiation R in the form $(x\langle\overline{\omega}, s\rangle \rightsquigarrow y : \omega[\overline{\alpha} \leftarrow \overline{\omega}_i, s]) \in \Gamma$.

The environment R also contains patches, *i.e.* $m\mathsf{ML}$ terms of the appropriate type, written in R as $y :^{\#} \sigma$. Well-formedness rules (Wf-R-Patch) require that the type σ corresponds to meta functions of multiple arguments returning a value of type $[\delta \# u]$, *i.e.* a thunk whose evaluation gives a result of type $\delta \# u$, where δ is the extension function of some ornament in S.

The elaboration judgment $\Gamma \vdash^p a \rightsquigarrow A : \omega \Rightarrow \Delta$ implies two typing judgments, one for the base term a and one for the elaborated term A (this will be stated and prove in Lemma 11.1). This requires being able to *project* the environments for the base and ornamented terms from the elaboration environment: these projections are defined in Figure 11.7.

The projection of an environment Γ of the form $G, (\alpha_i)^i, S, R, \Theta$, noted $(\Gamma)^\epsilon$ is built from the projection of its components. On the base side, S and R are ignored, since they correspond to abstraction added to create the generic term. For the local environment, variable definitions $x : \omega$ are translated by projecting ω. Since the projection of ornament types is parametrized by S, the projection of Θ is too. The generic-only constructs $((u \simeq u' : \tau)^{\#}$, $((u)*p : \tau)^{\#}$ and $(\pi)^{\#})$ are translated to themselves on the elaborated side, and ignored on the base side.

Similarly, the patch environment R is only translated on the elaborated side. A patch request $y :^{\#} A$ makes $y : A$ available on the elaborated side. An instance request $x\langle\overline{\omega}, s\rangle \rightsquigarrow y : \omega$ makes y available on the elaborated side, with type the

projection of ω.

For the ornament environment, a type variable representing the destination type, a type variable for the extension function, and two variables for the injection and projection functions are put in scope for each ornament request. To break up the arbitrary recursion in S, we build S^+ as the concatenation of two environments: first we quantify over type variables representing the ornamented types, then we quantify over the components of the ornament. The type for the components of the ornament only depend on the type variables previously defined.

Finally, the global environment of elaborated definitions G projects on the left to a polymorphic environment G^+ and a substitution from global definitions to their non-elaborated values.

The well-formedness rules for all these constructions are given on Figure 11.8.

11.3 Elaboration

The main elaboration judgment $\Gamma \vdash^p a \rightsquigarrow A : \omega \Rightarrow \Delta$ is described in Figure 11.9. It follows the structure of the original term. We need to expand Γ as $G, \overline{\alpha}, S, R, \Theta$ to describe the flow of information during elaboration: G, $\overline{\alpha}$, and Θ are inputs, while S and R are outputs, and added to on demand (some equality constraints may force two ornament variables in S to be unified). The term a is an input while A and ω are outputs. Applications, abstractions, let-bindings, and local variables are translated to themselves. We also add labels, label variables, equalities, and compute the output results.

We have already explained the elaboration of global variables.

The key rules are pattern matching and construction of datatypes. For E-Match, consider a pattern matching $\mathsf{match}\ a\ \mathsf{with}\ (d_i(\tau_j)^j(x_{ik})^{k\in K_i} \to b_i)^{i\in I}$. We first elaborate the term a that we are matching on to a term A. We expect the ornament relation to be some ornament variable φ: it must be a datatype ornament, and it cannot be a concrete ornament). We check the ornament environment for φ: we find an ornament specification $(\varphi \mapsto (\delta, \mathsf{inj}, \mathsf{proj}) \lhd \hat{\zeta}\,(\omega_{ik})^{i\in I, k\in K_i} : \zeta\,(\tau_j)^j \Rightarrow _) \in \Gamma$. We check that the base type is the one specified for the ornament. The type of the skeleton is $\hat{\zeta}\,(\omega_{ik})^{i\in I, k\in K_i}$, given by the ornamentation specification. Then, in the elaborated term, we bind A to a variable x and match on x of type $\hat{\zeta}\,((\omega_{ik})^+_\Gamma)^{i\in I, k\in K_i}$. We then elaborate each branch independently at a common ornament type ω in an environment that holds the equality between $\mathsf{reuse}(A)$ and the pattern of the branch. The ornamented versions are the $(B_i)^i$.

The rule for constructors E-Con works similarly: we lookup the returned ornament type φ, obtain the ornaments to be used for each argument from the skeleton, then elaborate each argument at this type (adding the results computed from previous elaborated arguments). Finally, we request a patch e that will have access to all local definitions as well as the results computed when evaluating the elaborated version of the arguments. In the elaborated term we first evaluate the elaborated versions $(A_j)^j$ of each argument and bind them to variables $(x_j)^j$, evaluate the result of applying the patch and bind it to y, then call the construction function inj with these arguments.

E-VarLocal
$$\frac{x : \omega \in \Gamma}{\Gamma \vdash^p x \rightsquigarrow x : \omega \Rightarrow \varnothing}$$

E-VarGlobal
$$\frac{(x\langle \overline{\alpha}\,, S', R\rangle : \omega) \in \Gamma \quad \Gamma \vdash s : S'[\overline{\alpha} \leftarrow \overline{\omega}] \quad (\overline{\omega}\,)^-_S = \overline{\tau} \quad (x\langle \overline{\omega}\,, s\rangle \rightsquigarrow y : \omega[\,\overline{\alpha} \leftarrow \overline{\omega}\,, s]) \in \Gamma}{\Gamma \vdash^p x\ \overline{\tau} \rightsquigarrow y : \omega[\,\overline{\alpha} \leftarrow \overline{\omega}\,, s] \Rightarrow \varnothing}$$

E-Let
$$\frac{\Gamma \vdash^p a \rightsquigarrow A : \omega_0 \Rightarrow \Delta_a \quad \Gamma, \Delta_a, x : \omega_0 \vdash^p b \rightsquigarrow B : \omega \Rightarrow \Delta_b}{\Gamma \vdash^p \mathsf{let}\ x = a\ \mathsf{in}\ b \rightsquigarrow \mathsf{let}\ x = A\ \mathsf{in}\ B : \omega \Rightarrow \Delta_a, \Delta_b[x \leftarrow \mathsf{reuse}(a)]}$$

E-App
$$\frac{\Gamma \vdash^p a \rightsquigarrow A : \omega_1 \to \omega_2 \Rightarrow \Delta_a \quad \Gamma \vdash^p b \rightsquigarrow B : \omega_1 \Rightarrow \Delta_b \quad p \leqslant q \quad q \perp \Gamma, \Delta_a, \Delta_b}{\Gamma \vdash^p a\ b \rightsquigarrow A\ B : \omega_2 \Rightarrow \Delta_a, \Delta_b, (\mathsf{True}) * q : (\Gamma)^+_{\omega_2}}$$

E-Fix
$$\frac{\begin{array}{c}\Gamma, x : \omega_1 \to \omega_2, y : \omega_1 \vdash^\pi a \rightsquigarrow A : \omega_2 \Rightarrow \Delta \\ \tau_1 \to \tau_2 = (\omega_1 \to \omega_2)^-_\Gamma \quad T_1 \to T_2 = (\omega_1 \to \omega_2)^+_\Gamma\end{array}}{\Gamma \vdash^p \mathsf{fix}\, x\ (y : \tau_1) : \tau_2\ .\ a \rightsquigarrow \mathsf{fix}^\pi\ x\ (y : T_1) : T_2\ .\ A : \omega_1 \to \omega_2 \Rightarrow \varnothing}$$

E-Con
$$\frac{\begin{array}{c}\Gamma = _, _, S, _, \Theta \quad p \leqslant q \quad p \perp \Theta, (\Delta_j)^j \\ \vdash d : \forall (\alpha_i)^i\ (\tau_j)^j \to \zeta\,(\alpha_i)^i \quad (\varphi \mapsto (\delta, \mathsf{inj}, \mathsf{proj}) \lhd \hat{\zeta}\,(\omega_k)^k : \zeta\,(\sigma_i)^i \Rightarrow _) \in S \\ (\Gamma, (\Delta_\ell)^{\ell<j} \vdash^p a_j \rightsquigarrow A_j : \omega_j \Rightarrow \Delta_j)^j \\ (e :^\# \Pi(\Theta^+_S, (\Delta_j)^j)\ [\delta \# \hat{d}((\omega_k)^+_S)^k(\mathsf{reuse}(A_j))^j]) \in \Gamma\end{array}}{\begin{array}{l}\Gamma \vdash^p d(\sigma_i)^i(a_j)^j \\ \quad \rightsquigarrow \mathsf{let}\ (x_j = A_j)^j\ \mathsf{in}\ \mathsf{let}\ y = (e \# (\omega_k)^+_S \# (\Delta_j)^j)^q\ \mathsf{in}\ \mathsf{inj} \# \hat{d}((\omega_k)^+_S)^k(x_j)^j \# y : \varphi \\ \quad \Rightarrow (\Delta_j)^j, (\mathsf{True}) * q : \delta \# \hat{d}((\omega_k)^+_S)^k(\mathsf{reuse}(A_j))^j\end{array}}$$

E-Match
$$\frac{\begin{array}{c}(\varphi \mapsto (\delta, \mathsf{inj}, \mathsf{proj}) \lhd \hat{\zeta}\,(\omega_{ik})^{i\in I, k\in K_i} : \zeta\,(\tau_j)^j \Rightarrow _) \in \Gamma \quad \Gamma \vdash^p a \rightsquigarrow A : \varphi \Rightarrow \Delta \\ \left(\begin{array}{l}\Gamma, \Delta, (x_{ik} : \omega_{ik})^{k\in I_k}, \\ (\mathsf{proj} \# \mathsf{reuse}(A) \simeq \hat{d}_i((\omega_{\ell k})^+_S)^{\ell\in I, k\in K_\ell}(x_{ik})^{k\in K_i} : \hat{\zeta}\,((\omega_{\ell k})^+_S)^{\ell\in I, k\in K_\ell}) \# \\ \qquad \vdash^p b_i \rightsquigarrow B_i : \omega \Rightarrow \Delta_i\end{array}\right)^i\end{array}}{\begin{array}{l}\Gamma \vdash^p \mathsf{match}\ a\ \mathsf{with}\ (d_i(\tau_j)^j(x_{ik})^{k\in K_i} \to b_i)^{i\in I} \\ \quad \rightsquigarrow \mathsf{let}\ y = A\ \mathsf{in}\ \mathsf{match}\ \mathsf{proj} \# y\ \mathsf{with}\ (\hat{d}_i((\omega_{ik})^+_S)^{\ell\in I, \ell\in K_\ell}(x_{ik})^{k\in K_i} \to B_i)^{i\in I} : \omega \\ \quad \Rightarrow \Delta, (\mathsf{match}\ x\ \mathsf{with}\ \hat{d}_i((\omega_{\ell k})^+_S)^{\ell\in I, k\in K_\ell}(x_{ik})^{k\in K_i} \to \Delta_i)^i\end{array}}$$

Figure 11.9: Elaboration to a generalized term

Elab-Decl
$$\frac{G, \overline{\alpha}, S, R \vdash^\varepsilon a \rightsquigarrow A : \omega \Rightarrow \Delta}{G \vdash \mathsf{let}\ x = \Lambda\overline{\alpha}.\ a \Rightarrow G, (x\langle \overline{\alpha}, S, R\rangle : \omega = a \rightsquigarrow A)}$$

Figure 11.10: Elaborating a declaration

11.4 Correctness of elaboration

As announced earlier, the elaboration judgments ensure well-typedness of the projections and the definition elaboration judgment preserves the well-formedness of G:

Lemma 11.1. *If* $\Gamma \vdash^p a \rightsquigarrow A : \omega \Rightarrow \Delta$ *holds then both* $\Gamma^- \vdash a : \omega_\Gamma^-$ *and* $\Gamma^+ \vdash^p A : \omega_\Gamma^+ \Rightarrow \Delta$ *hold.*

Proof. By induction on the derivation. □

Lemma 11.2. *If* $\vdash G$ *and* $G \vdash t \Rightarrow G'$ *hold, then* $\vdash G'$ *holds.*

Proof. Expand the definitions, apply G-Def and use Lemma 11.1. □

In practice, the elaboration is obtained by inference. We first construct an elaborated term where all ornamentation records are different, and type it using the normal ML inference (this always succeeds because the term can be instantiated with records defining identity ornaments). Then, according to the constraints on elaboration environments, ornaments with the same lifted type must be the same. This is in fact sufficient: we only have to merge the ornaments whose lifted types are unified by ML inference. We thus obtain the most general generic program. We describe our implementation of this process in Section 13.1.

11.5 Identity instantiation

The correctness of the elaboration (Theorem 12.1) is based on the fact that, when instantiated with the identity ornament, the elaborated term is equal (for the mML equality) to the base term. Along with parametricity, this result is sufficient to prove the correctness of the ornamentation.

Consider a generic term $x\langle\overline{\alpha}, S, R\rangle : \omega = a \rightsquigarrow A$. We obtain the identity instantiation by instantiating all ornaments with the identity ornament, all lifting requests with the base definition, and all patches with a patch that always return Unit.

Definition 11.1 (Identity instantiation)**.** *Suppose* $x\langle\overline{\alpha}, S, R\rangle : \omega = a \rightsquigarrow A$. *Then, the identity instantiation of* A*, noted* $\mathsf{IdInst}_{S,R}$ *is defined as follows:*

- *For all* $(\varphi \mapsto (\delta, \mathsf{proj}, \mathsf{inj}) \lhd \hat{\zeta}\,(\omega_i)^i : \zeta\,(\tau_\ell)^\ell \Rightarrow \beta) \in S$*, we instantiate* φ *with the identity ornament: we have* $\mathsf{IdInst}_{S,R}(\beta) = \zeta\,(\tau_\ell)^\ell$, $\mathsf{IdInst}_{S,R}(\delta) = \delta_{\zeta\,(\tau_\ell)^\ell}$, $\mathsf{IdInst}_{S,R}(\mathsf{proj}) = \mathsf{proj}_{\zeta\,(\tau_\ell)^\ell}$, $\mathsf{IdInst}_{S,R}(\mathsf{inj}) = \mathsf{inj}_{\zeta\,(\tau_\ell)^\ell}$.
- *For all* $e :^\# \Theta \to \delta \# A \in R$*, we set* $\mathsf{IdInst}_{S,R}(e) = \lambda^\# \Theta.\,[\pi.\,\mathsf{Unit}]$.
- *For all* $(x\langle\overline{\omega}, s\rangle \rightsquigarrow y : \omega') \in R$*, we use the base term:* $\mathsf{IdInst}_{S,R}(y) = x\,(\overline{\omega})_S^-$.

◊

Then, we obtain a well-typed instantiation in the base environment:

Lemma 11.3. *If* $\vdash G, \overline{\alpha}, S, R$*, then* $\mathsf{IdInst}_{S,R}$ *exists and* $(G, \overline{\alpha})^- \vdash \mathsf{IdInst}_{S,R} : (S, R)^+$.

Proof. By induction on the well-formedness judgments of S and R. The identity ornament is well-formed, thus the functions that define it have the correct types. For any ornament in scope, the extension δ is $\lambda_.\ \mathsf{unit}$. Thus, the patches are well-typed. For the liftings, notice that each ornament is the identity ornament: thus, we ask for a function of the same type as the original function. □

Lemma 11.4 (Identity instantiation and types)**.** *Suppose ω is well formed in S. Then for all R, we have:*

$$(\omega)_S^- = (\omega)_S^+[\mathsf{IdInst}_{S,R}]$$

Proof. By induction over ω. If $(\varphi \mapsto _ \lhd _ : \tau \Rightarrow \beta) \in S$, the ornament variable φ is replaced by τ on the left-hand side by left-projection. On the right-hand side it is replaced by β by right-projection, then by τ again when substituting with the identity instantiation. □

We define a variant of equality that passes more easily to context: a label on the left-hand side is always matched with the same label on the right-hand side. This allows substituting them into the a context that may use the results to obtain equal terms again.

Definition 11.2 (Exact equality)**.** *Consider two terms a and a'. Suppose* $\mathsf{atoms}(a)$ *is* $(*p_i \leftarrow a_i)^{i \in I}$ *with conditions u_i and types τ_i, and* $\mathsf{atoms}(a')$ *is* $(*p'_j \leftarrow a'_j)^{j \in J}$ *with conditions u'_j and types τ'_j.*

Then suppose there exists a permutation σ mapping the indices in I to the indices in J such that:

- *Results with the same name are equivalent: for all i,*

  $$\Gamma, ((p_k)*u_k : \tau_k)^{k<i \vee \sigma(k)<\sigma(i)} \vdash u_i : \mathsf{bool} \simeq u'_{\sigma(i)} : \mathsf{bool}$$

  and

  $$\Gamma, ((p_k)*u_k : \tau_k)^{k<i \vee \sigma(k)<\sigma(i)} \vdash \mathsf{atom}_i : \mathsf{atomty}_i \simeq \mathsf{atom}'_{\sigma(i)} : \mathsf{atomty}'_{\sigma(i)}$$

- *The results are executed in the same order: for all i,*

  $$\Gamma, ((p_k)*u_k : \tau_k)^{k<i \vee \sigma(k)<\sigma(i)} \vdash \mathsf{count}(u_k)^{k<i} : \mathsf{nat} \simeq \mathsf{count}(u'_k)^{k<\sigma(i)} : \mathsf{nat}$$

  where $\mathsf{count}(u_i)^i$ *counts the number of expressions evaluating to* True*:* $\mathsf{count}(\varnothing) = \mathsf{Z}$ *and* $\mathsf{count}(u, (u_i)^i) = \mathsf{match}\ u\ \mathsf{with}\ \mathsf{False} \to \mathsf{count}(u_i)^i \mid \mathsf{True} \to \mathsf{S}\ (\mathsf{count}(u_i)^i)$.

- *The reusable versions of a and a' are equal:*

  $$\Gamma, ((u_i)*p_i : \tau_i)^i \vdash \mathsf{reuse}(a) : \tau \simeq \mathsf{reuse}(a') : \tau$$

  .

Then we say that they are exactly equal, and we write this equality $\mathsf{ExactlyEqual}(\Gamma \vdash^p a : \tau \simeq a' : \tau')$. ◇

Lemma 11.5. *Exact equality is:*

- *reflexive: if $\Gamma \vdash^p a : \tau \Rightarrow \Delta$, then* $\mathsf{ExactlyEqual}(\Gamma \vdash^p a : \tau \simeq a : \tau)$*;*

- *symmetric: if* $\mathsf{ExactlyEqual}(\Gamma \vdash^p a_1 : \tau_1 \simeq a_2 : \tau_2)$*, then* $\mathsf{ExactlyEqual}(\Gamma \vdash^p a_2 : \tau_2 \simeq a_1 : \tau_1)$*;*
- *transitive: if* $\mathsf{ExactlyEqual}(\Gamma \vdash^p a_1 : \tau_1 \simeq a_2 : \tau_2)$ *and* $\mathsf{ExactlyEqual}(\Gamma \vdash^p a_2 : \tau_2 \simeq a_3 : \tau_3)$*, then* $\mathsf{ExactlyEqual}(\Gamma \vdash^p a_1 : \tau_1 \simeq a_3 : \tau_3)$*.*

Proof. For reflexivity, use the identity permutation. Then everything matches by reflexivity. For symmetry, invert the permutation and the equalities For transitivity, compose the permutation and the equalities. □

Lemma 11.6 (Exact equality is a congruence)**.** *Suppose* $\mathsf{ExactlyEqual}(\Gamma \vdash^p a : \tau \simeq a' : \tau')$*. Then, if* $\Gamma' \vdash^{p'} C[a] : \sigma \Rightarrow \Delta'$*, we also have* $\mathsf{ExactlyEqual}(\Gamma' \vdash^{p'} C[a] : \sigma \simeq C[a'] : \sigma)$*.*

Proof. Looking at the decomposition: the reusable versions are equal, since non-expansive equality is a congruence. Moreover, the terms match, by taking the substitution that maps every label in C to itself in the same position, and the substitution for a and a' for the labels present in a and a'. □

Lemma 11.7 (Exact equality implies term equality)**.** *Suppose* $\mathsf{ExactlyEqual}(\Gamma \vdash^p a : \tau \simeq a' : \tau')$*. Then* $\mathsf{TermsEqual}(\Gamma \vdash^p a : \tau \simeq a' : \tau')$*.*

Proof. The main difficulty is in matching the indexing: from the count matching we can prove that the indexings also match (the count is intuitively the inverse operation of indexing). □

Theorem 11.1. *Suppose* $G, \overline{\alpha}, S, R \vdash^p a \rightsquigarrow A : \omega \Rightarrow \Delta$*.*
Then, $\mathsf{TermsEqual}((G)^-, \overline{\alpha} \vdash^p a : (\omega)^-_S \simeq A[\mathsf{IdInst}_{S,R}] : (\omega)^-_S)$*,*

Proof. We first consider A' the term obtained by reducing all patches to Unit in $A[\mathsf{IdInst}_{S,R}]$ (this preserves equality because it is an $m\mathsf{ML}$ reduction).

We then prove by induction on a a stronger result that allows for a non-empty local binding environment Θ, and using exact equality to use congruence: if $G, \overline{\alpha}, S, R, \Theta \vdash^p a \rightsquigarrow A : \omega \Rightarrow \Delta$, then $\mathsf{ExactlyEqual}((G)^-, \overline{\alpha}, (\Theta)^+_S[\mathsf{IdInst}_{S,R}] \vdash^p a : (\omega)^-_S \simeq A' : (\omega)^-_S)$.

Let us examine the rules:

- For Var-Local, we conclude by reflexivity.
- For Var-Global: this is true by our choice of instantiation.
- Other rules except E-Match and E-Con: this is true by congruence.
- For E-Con: first, apply congruence to all subexpressions. We then have to prove that $d(\tau_i)^i(a_j)^j$ is equal to $(A'_j)^j; \mathsf{Unit}; \mathsf{inj}_{\zeta\,(\tau_i)^i} \# \hat{d}(\mathsf{reuse}(A'_j))^j \# \mathsf{Unit}$. Both terms do the same computation in the same order, we only have to prove that the reusable versions are equal. By hypothesis, for all j, $\mathsf{reuse}(a_j)$ and $\mathsf{reuse}(A'_j)$ are equal. Then the right-hand side reduces to $d(\mathsf{reuse}(A'_j))^j$ so we get the equality.
- For E-Match: proceed similarly by applying congruence to all expressions. To prove the final equality of the reusable terms, split on $\mathsf{reuse}(a)$.

□

Chapter 12

Lifting by instantiation

Once the user has defined some ornaments and the base terms have been elaborated, the user specifies a set of liftings of the base definitions. We describe here how these liftings are processed.

A lifting of x to y is declared as:

$$\mathsf{let}\ y\ \overline{\beta} = \mathsf{lifting}\ x\ (\omega_j)^j\ \mathsf{with}\ s, r$$

The lifting y is polymorphic over a set of type variables β. It is a lifting of a toplevel definition x of type $\forall(\alpha_j)^j\ \tau$, with its type arguments instantiated with $(\omega_j^-)^j$. The declaration of x has been generalized: there is a generalized definition $x\langle\overline{\alpha}, S, R\rangle : \omega = a \rightsquigarrow A$. To obtain the lifting, the user provides an instantiation s of the ornaments of S and an instantiation r of R. These instantiations are then used to instantiate the generic term and obtain the lifting.

In this chapter, we describe how such a definition is checked, what ornament specification is derived for y, and we prove that the term we obtain indeed verifies this ornament specification.

12.1 Specifying liftings

Consider an elaborated term $x\langle\overline{\alpha}, S, R\rangle : \omega = a \rightsquigarrow A$ from the global elaboration environment G. To instantiate it, we need to choose first a set of types to instantiate the $\overline{\alpha}$, then a sequence of ornaments for S and finally a sequence of patches and liftings for R. If the sets of types, ornaments, patches, and liftings given match the requirement, we obtain a lifting at some specification ω' obtained by specializing ω. We then register these liftings in a lifting environment.

The ornament instantiation s is a substitution from ornament variables to concrete datatype ornaments. We need to check that it conforms to the ornament specification S. This is checked by the judgment $\overline{\beta} \vdash s : S$ given in Figure 12.1: for every ornament request $\varphi_i \mapsto _ \lhd \hat{\omega}_i : \sigma_i \Rightarrow _$ in S, we check that there is a binding $s(\varphi_i) = \chi_i\ (\omega_{ij})^j\ (\omega'_{ik})^k$ in s such that χ_i is a valid datatype ornament (checked in a global environment left implicit), that the unconstrained arguments $(\omega_{ij})^j$ are valid ornament types, and the constrained arguments types $(\omega'_{ik})^k$ have the right base types, and that the skeleton of $\chi_i\ (\omega_{ij})^j\ (\omega'_{ik})^k$ matches the skeleton given in the request, after substitution of s in the request.

$$\textsc{Wf-OrnInst}\quad \frac{\begin{array}{c}(s(\varphi_i) = \chi_i\ (\omega_{ij})^j\ (\omega'_{ik})^k)^i \qquad (\vdash \chi_i\ (\alpha_{ij})^j\ (\beta_{ik} \lhd \tau_{ik})^k \mapsto _ \lhd \hat{\omega}'_i : \sigma'_i \Rightarrow _)^i \\ (\overline{\alpha} \vdash \omega_{ij})^{ij} \qquad (\overline{\alpha} \vdash \omega'_{ik})^{ik} \qquad ((\omega'_{ik})^- = \tau_{ik}[\alpha_{ij} \leftarrow (\omega_{ij}[s])^-]^j)^{ik} \\ (\hat{\omega}_i[s] = \hat{\omega}'_i[(\alpha_{ij} \leftarrow \omega_{ij})^j, (\beta_{ik} \leftarrow \omega'_{ik})^k])^i\end{array}}{\overline{\alpha} \vdash s : (\varphi_i \mapsto _ \lhd \hat{\omega}_i : \sigma_i \Rightarrow _)^i}$$

$$(s)^+_{\varnothing} = \varnothing \qquad (s)^+_{S,\varphi\mapsto(\delta_\varphi,\mathsf{proj}_\varphi,\mathsf{inj}_\varphi)\lhd_:_\Rightarrow\beta} = \begin{array}{l}(s)^+_S, \\ \beta \leftarrow (s(\varphi))^+, \\ \delta_\varphi \leftarrow \delta_{s(\varphi)}, \\ \mathsf{inj}_\varphi \leftarrow \mathsf{inj}_{s(\varphi)}, \\ \mathsf{proj}_\varphi \leftarrow \mathsf{proj}_{s(\varphi)}\end{array}$$

Figure 12.1: Ornament instantiation: checking and projection

$$I \ ::= \ \varnothing \mid I, \forall\overline{\alpha}\,(x\langle\overline{\omega}, s\rangle \rightsquigarrow y\ \overline{\alpha} : \omega = A)$$

$$\varnothing^+ = \varnothing \qquad (I, \forall\overline{\alpha}\,(x\langle(\omega_i)^i, s\rangle \rightsquigarrow y\ \overline{\alpha} : \omega = A))^+ = I^+, y\ \overline{\alpha} : \omega^+ = A$$

$$\textsc{LiftEnv-Empty}\quad G \vdash \varnothing$$

$$\textsc{LiftEnv-Cons}\quad \frac{\begin{array}{c}G \vdash I \qquad (x\langle(\beta_j)^j, S, _\rangle : \omega') \in G \\ (\overline{\alpha} \vdash \omega_j)^j \qquad \overline{\alpha} \vdash s : S \qquad I^+; \overline{\alpha}; \varnothing \vdash A : \omega^+ \qquad \omega = \omega'[(\beta_j \leftarrow \omega_j)^j, s]\end{array}}{G \vdash I, \forall\overline{\alpha}\,(x\langle(\omega_j)^j, s\rangle \rightsquigarrow y\ \overline{\alpha} : \omega = A)}$$

Figure 12.2: Lifting environment

When $\overline{\beta} \vdash s : S$ holds, we may take the right-projection s^+_S of s defined in Figure 12.1: it binds all the variables bound associated to the ornament φ on the right-hand side to the value associated to the concrete ornament $s(\varphi)$. For example, if S binds a projection function proj_φ for ornament φ, and S substitutes φ with $\mathsf{natlist\ bool}$, then s^+_S substitutes proj_φ with $\mathsf{proj}_{\mathsf{natlistbool}}$.

Lemma 12.1. *Suppose $\overline{\beta} \vdash s : S$. Then, $\overline{\beta} \vdash s^+_S : (S)^+$*

Proof. This derives from the checking judgment and the types of the components of valid ornaments (Definition 10.2). □

From a valid s, we can deduce the specification of the lifting: it will be $\omega[s]$. We store the result A of the instantiations (we will describe how to compute it later) in a lifting environment I whose syntax is given in Figure 12.2. It is composed of bindings of the form $\forall\overline{\alpha}(x\langle\overline{\omega}, s\rangle \rightsquigarrow y\ \overline{\alpha} : \omega = A)$ indicating that the base definition x instantiated with $\overline{\omega}$ has been lifted with ornament instantiation s to y with ornament specification ω. A lifting environment projects to the ornamented side as a global definition environment, as defined in Figure 12.2: each lifting is projected to its definition. We define a judgment $G \vdash I$ that checks that a lifting environment is valid, *i.e.* that all the liftings provided have the right type and ornament specification.

$$
\begin{array}{rll}
r & ::= & \text{Patch instantiation} \\
 & \mid \varnothing & \text{Empty} \\
 & \mid y \leftarrow z\ \overline{\omega} & \text{Lifting} \\
 & \mid e \leftarrow u & \text{Patch}
\end{array}
$$

$$
(\varnothing)^+ = \varnothing \qquad (r, y \leftarrow z\ \overline{\omega})^+ = (r)^+, y \leftarrow z\ (\overline{\omega})^+ \qquad (r, e \leftarrow u)^+ = (r)^+, e \leftarrow u
$$

$$
\textsc{Wf-r-Empty}\quad I; \overline{\alpha} \vdash r : \varnothing
$$

$$
\textsc{Wf-r-Patch}\quad \frac{I; \overline{\alpha} \vdash r : R \qquad I^+, \overline{\alpha} \vdash r(e) : T[(r)^+]}{I; \overline{\alpha} \vdash r : R, e :^{\#} T}
$$

$$
\textsc{Wf-r-Lifting}\quad \frac{\begin{array}{c} I; \overline{\alpha} \vdash r : R \qquad r(y) = z\ (\omega_i)^i \qquad (\forall (\beta_i)^i\ x\langle (\omega'_j)^j, s' \rangle \rightsquigarrow z\ (\beta_i)^i : \omega' = _) \in I \\ (\omega_j = \omega'_j[\beta_i \leftarrow \omega_i]^i)^j \qquad s = s'[\beta_i \leftarrow \omega_i]^i \qquad \omega = \omega'[\beta_i \leftarrow \omega_i]^i \end{array}}{I; \overline{\alpha} \vdash r : R, (x\langle (\omega_j)^j, s \rangle \rightsquigarrow y : \omega)}
$$

Figure 12.3: Patch instantiation: grammar, projection and well-formedness

Lemma 12.2. *Suppose $G \vdash I$. Then $\vdash (I)^+$.*

Proof. This is implied by the typing condition for well-formed lifting environments. □

We now have to check the patch environment r, which is a substitution from patch names e to patches u (*i.e.* fragments of mML code) and of names of requested liftings y to names of actual liftings applied to ornament arguments $z\ (\omega_i)^i$, where z is a previously defined lifting in I. This is checked by a judgment $I; \overline{\beta} \vdash r : R$ defined in Figure 12.3. For a patch request $e :^{\#} T$, the judgment requires that $r(e)$ is a non-expansive term u of type T (with the substitution $(r)^+$ applied, so that information learned from previous patches can be reused). For a lifting request $x\langle (\omega_j)^j, s \rangle \rightsquigarrow y : \omega$, we check that it maps to an instance $z\ (\omega_i)^i$ of a lifting of x and that the instances of ornaments in the request and in the lifting match. Before checking the types, we apply the substitution s to R, so that we do not need the judgment to be parametrized by s.

We also define a projection $(r)^+$ of r in Figure 12.3.

Lemma 12.3. *Suppose $\overline{\alpha} \vdash s : S$, $G \vdash I$ and $I; \overline{\alpha} \vdash r : R[s, (s)^+_S]$. Then, $I^+, \overline{\alpha} \vdash (r)^+ : (R)^+_S$.*

Proof. This is implied by the typing conditions for patches and liftings. □

The processing of an instantiation is described by the rule Lifting given in Figure 12.4. To process an instantiation, we first find in G the binding $(x\langle \overline{\alpha}, S, R \rangle : \omega = a \rightsquigarrow A)$ for the variable x. We then check the validity of s, then r. Finally, we construct the instantiated term $A[\alpha_j \leftarrow (\omega_j)^+, (s)^+_S, (r)^+]$. This is an mML term: we can meta-reduce it back to eML (Theorem 7.3) then eliminate the eML constructs to obtain an ML term (Theorem 9.1). For a term B, we note $\mathsf{simplify}(B)$ the result of applying these two operations. We then insert the instantiated and simplified term in the lifting environment along with its lifting specification.

LIFTING

$$\frac{(x\langle(\alpha_i)^i, S, R\rangle : \omega = a \rightsquigarrow A) \in G \qquad (\overline{\beta} \vdash \omega_i)^j \qquad \overline{\beta} \vdash s : S[\alpha_i \leftarrow \omega_i]^i \qquad I; \overline{\beta} \vdash r : R[(\alpha_i \leftarrow \omega_i)^i, s, (s)^+_{S[\alpha_i \leftarrow \omega_i]^i}]}{\begin{array}{l} G; I \vdash \mathsf{let}\ y\ \overline{\beta} = \mathsf{lifting}\ x\ (\omega_j)^j\ \mathsf{with}\ s, r \\ \quad \Rightarrow I, (\forall \overline{\beta}\ x\langle(\omega_j)^j, s\rangle \rightsquigarrow y\ \overline{\beta} : \omega[(\alpha_i \leftarrow \omega_i)^i, s] \\ \quad = \mathsf{simplify}(A[(\alpha_i \leftarrow (\omega_i)^+)^i, (s)^+_{R[(\alpha_i \leftarrow \omega_i)^i]}, r^+])) \end{array}}$$

Figure 12.4: Lifting

We have the following result:

Lemma 12.4. *Suppose* $\vdash G$ *and* $G \vdash I$ *and* $G; I \vdash t \Rightarrow I'$. *Then,* $G \vdash I'$.

Proof. By hypothesis on G, we have $x\langle\overline{\alpha}, S, R\rangle : \omega = a \rightsquigarrow A$. By Lemma 12.1, $\overline{\beta} \vdash s^+_S : (S)^+$. By Lemma 12.3, $I^+, \overline{\alpha} \vdash (r)^+ : (R)^+_S$. Thus,

$$I^+, \overline{\alpha} \vdash A[(\alpha_i \leftarrow (\omega_i)^+)^i, (s)^+_{R[\alpha_i \leftarrow \omega_i]^i}, r^+] : (\omega)^+_S[(\alpha_i \leftarrow (\omega_i)^+)^i, (s)^+_{R[\alpha_i \leftarrow \omega_i]^i}, r^+]$$

The final type is $(\omega[(\alpha_i \leftarrow \omega_i)^i, s])^+$ because the patches and liftings do not appear in the ornament type, and projection commutes with substitution. Then, the term is ML-typed and can be simplified to an ML term by $\mathsf{simplify}(_)$. □

These judgments check that the lifting is valid. In our prototype, they are also used to generate constraints that are used to infer the ornaments and liftings that have not been specified by the user. We give more details of this process in Section 13.2.

12.2 Correctness of the lifting

We use the logical relation from Chapter 8 to prove that the lifted term is related to the base term by ornamentation. We prove that any valid instantiation is related to the identity instantiation, thus by parametricity of the generic term, the identity instantiation and the user-specified instantiation are related at the ornament specification.

Suppose given an elaboration environment G generated as described in Chapter 11. Every binding in G has its base term equal to its identity instantiation.

Definition 12.1. *We say that a lifting environment* I *is correct relative to an elaboration environment* G *if, for every lifting* $\forall\overline{\alpha}(x\langle\overline{\omega}, s\rangle \rightsquigarrow y\ \overline{\alpha} : (\omega_i)^i = A) \in I$ *there is a matching binding* $x\langle(\beta_i)^i, _, _\rangle : _ = a \rightsquigarrow _ \in G$, *and for all indices* p *and* $\gamma \in \mathcal{G}_p[\overline{\alpha}]$, *we have* $(\gamma_1(a[\beta_i \leftarrow (\omega_i)^-]^i), \gamma_2(A)) \in \mathcal{V}_p[\omega]_\gamma$. ♢

Then, we have the following correctness result for instantiation, stating that liftings are indeed lifted at the advertised ornament type:

Theorem 12.1 (Instantiation is correct)**.** *Consider* $\vdash G$, *and suppose* I *is correct relative to* G. *If* $G; I \vdash \mathsf{let}\ y\ \overline{\beta} = \mathsf{lifting}\ x\ (\omega_j)^j\ \mathsf{with}\ s, r \Rightarrow I'$, *then* I' *is also correct relative to* G.

Proof. We use here the notations from LIFTING in Figure 12.4.

Consider an index p and an environment $\gamma \in \mathcal{G}_p[\overline{\alpha}]$. We prove the correctness of the ornamentation by constructing a relational instantiation $\gamma' \in \mathcal{G}_p[((\beta_i)^i, S, R)^+]$. as follows:

- For all β_i, $\gamma'(\beta_i) = (\mathcal{V}_q[\omega_i]_\gamma)^q$.
- For all ornaments $(\varphi \mapsto (\delta, \mathsf{proj}, \mathsf{inj}) \lhd \hat{\zeta}\,(\omega_i)^i : \zeta\,(\tau_\ell)^\ell \Rightarrow \beta) \in S \in S$, we set γ'_1 and γ'_2 using the values and types in $\mathsf{IdInst}_{S,R}$ and $(s)^+_S$ respectively, and the relations as follows:
  - $\gamma'_R(\beta) = (\mathcal{V}_q[r(\varphi)]_\gamma)^q$;
  - $\gamma'_R(\delta) = (\lambda(v_1, v_2).\,\{(\mathsf{Unit}, w_2) \mid \vdash w_2 : \gamma'_2(\delta) \# v_2\})^q$
- For all patches e, $\gamma'(e) = (\mathsf{IdInst}_{S,R}(e), r(e))$.
- For all liftings y of $x\ (\omega_i)^i$, $\gamma'(y) = (x\ ((\omega_i)^-_S)^i, r(y))$.

This is a valid relational environment:

- Ornaments are related by Theorem 10.1.
- Patches are related: the type of a patch is always of the form $\Pi(\Gamma).[\delta \# u]$ for some term u. Consider u_1 and u_2 the two patches. Then, we apply some arguments described by Γ and evaluate the thunk, and compare the results in $\mathcal{E}_q[\delta \# u]_{\gamma''}$ for some γ''. The left-hand side reduces to Unit. Suppose the right-hand side reduces to a value v_2. Then, the relation for $\delta \# u$ relates Unit to any value, thus the relation holds.
- Liftings are related by the condition on I.

We have by hypothesis on G: $G, \overline{\alpha}, S, R \vdash^p a \rightsquigarrow A : \omega \Rightarrow \Delta$. Then, by Lemma 11.1, $\overline{\alpha}, S^+, R^+_S \vdash^p A : \omega^+_S \Rightarrow \Delta$. Then, by Theorem 8.1, we have $(\gamma'_1(A), \gamma'_2(A)) \in \mathcal{V}_p[\omega^+_S]_{\gamma'}$

We have $\omega^+_S = \omega[\varphi \leftarrow \beta_\varphi]$ where β_φ is the type variable associated to φ in S. γ'_R associates β_φ to $\mathcal{V}[s(\varphi)]_\gamma$. Moreover, γ' associates the $(\alpha_i)^i$ to the relation of the corresponding ω_i interpreted in γ. Thus, by substitution, $\mathcal{V}_p[\omega^+_S]_{\gamma'} = \mathcal{V}_p[\omega[(\alpha_i \leftarrow \omega_i)^i, s]]_{\gamma,\gamma'} = \mathcal{V}_p[\omega[(\alpha_i \leftarrow \omega_i)^i, s]]_\gamma$ (the last equality is because the variables in γ' do not appear in the type $\omega[(\alpha_i \leftarrow \omega_i)^i, s]$).

In the pair of related terms $(\gamma'_1(A), \gamma'_2(A))$, the left-hand term is equal to $A[\mathsf{IdInst}_{S,R}][\alpha_i \leftarrow (\omega_i)^-]^i$. By Theorem 11.1 applied to the elaboration A of a, we have $(\beta_i)^i \vdash A : (\omega)^-_S \simeq a : (\omega)^-_S$. Then:

$$\overline{\alpha} \vdash \gamma'_1(A) : (\omega)^-_S[\alpha_i \leftarrow (\omega_i)^-]^i \simeq a[\alpha_i \leftarrow (\omega_i)^-]^i : (\omega)^-_S[\alpha_i \leftarrow (\omega_i)^-]^i$$

The right-hand side term $\gamma'_2(A)$ is $A[(\alpha_i \leftarrow (\omega_i)^+)^i, (s)^+_{R[(\alpha_i \leftarrow \omega_i)^i]}, r^+]$. By Theorem 9.1:

$$\overline{\beta} \vdash \gamma'_2(A) : (\omega[s])^+ \simeq \mathsf{simplify}(A[(\alpha_i \leftarrow (\omega_i)^+)^i, (s)^+_{R[(\alpha_i \leftarrow \omega_i)^i]}, r^+]) : (\omega[s])^+$$

Finally, since the relation is stable by equality, and by composition with γ:

$$\begin{aligned}&(\gamma_1(a[\alpha_i \leftarrow (\omega_i)^-]^i), \gamma_2(\mathsf{simplify}(A[(\alpha_i \leftarrow (\omega_i)^+)^i, (s)^+_{R[(\alpha_i \leftarrow \omega_i)^i]}, r^+]))) \\ &\quad \in \mathcal{V}_p[\omega[(\alpha_i \leftarrow \omega_i)^i, s]]_\gamma\end{aligned}$$

□

In the case of strictly positive datatypes and first-order functions, this result can be translated to the *coherence* property of ornaments [Dagand and McBride, 2014]. We can define a *projection function* that projects the whole datatype at once (rather than incrementally as with the proj function), *e.g.* the length function for $\mathsf{natlist}\ \alpha$. Then, the relation expressed by $\mathsf{natlist}\ \alpha$ between a and A is simply $a = \mathsf{length}\ A$ (up to termination). Consider the statement that append is a lifting of add at $\mathsf{natlist}\ \alpha \to \mathsf{natlist}\ \alpha \to \mathsf{natlist}\ \alpha$. Following the definition of the relation, it is equivalent (again, up to termination) to the fact that for all a_1 and A_1 related by $\mathsf{natlist}$, and all a_2 and A_2 also related, the terms $\mathsf{add}\ a_1\ a_2$ and $\mathsf{append}\ A_1\ A_2$ are related. Since $a_1 = \mathsf{length}\ A_1$ and $a_2 = \mathsf{length}\ A_2$, this translates to the fact that for all A_1, A_2, $\mathsf{length}\ (\mathsf{append}\ A_1\ A_2) = \mathsf{add}\ (\mathsf{length}\ A_1)\ (\mathsf{length}\ A_2)$.

In order to prove that, when the patches terminate, the lifted term terminates whenever the base term terminates, we need to use the relation the other way around, with the base term on the right-hand side and the lifted type on the left-hand side: for each ornament, we define a relation $\mathcal{V}_q[\chi^{-1}\ (\omega_i)^i]_\gamma$ using the same definition with the sides reversed. Then, a patch is in the interpretation of its type if it always terminates, and the final relation then guarantees that the lifted term terminates more often than the base term. Since the base term also terminates more often than the lifted term, they terminate on exactly the same inputs.

Part IV

Implementing lifting

Chapter 13

Implementation

Our implementation works on a small subset of OCaml containing only the constructs of core ML with datatypes, but without objects, GADTs, modules, *etc.* (we explore possible extensions to the language in Section 15.4). It follows the structure outlined in Chapter 3. In this chapter, we examine some interesting parts of the implementation: how inference is used to perform generalization (§13.1); how we process the lifting directives given by the user and infer the missing parts (§13.2), and how we annotate terms to recover more readable terms after *e*ML simplification (§13.3).

13.1 Generalization by inference

The first step for lifting a term is to find a generalized term and the constraints on its instantiations. The rules for generalization we presented in Chapter 11 are presented as checking rules: they assume given a set of datatype ornaments and check that to each datatype constructor and pattern matching we assign an ornament variable of the right specification. In practice, we do not want the user to have to specify the generalization themself: instead, we should compute the most generic term that can be obtained from the base declaration.

For instance, consider again the example of addition of natural numbers:

```
let rec add = fun m n → match m with
    | Z → n
    | S m' → S (add m' n)
```

As we explained previously, it is possible to use two distinct ornaments of natural numbers, one transforming the first input, and the other transforming the second input and the output. Thus, the most generic lifting has the following signature, using the notation established in Chapter 11:

$$\mathsf{add}\langle(\varphi \mapsto _ \lhd \widehat{\mathsf{nat}}\ \varphi : \mathsf{nat} \Rightarrow _), (\psi \mapsto _ \lhd \widehat{\mathsf{nat}}\ \psi : \mathsf{nat} \Rightarrow _), _\rangle : \varphi \to \psi \to \psi$$

Reusing ML inference

We implement this by building upon ML type inference [Milner, 1978] to also extract information about ornaments. The intuition behind this process is that when two variables have the same type, it can be because their types have actually been unified during inference, or simply because we constructed the same

type expression twice, but unrelatedly. We can observe both in the example of add: while the branch Z → n actually unifies the type of n and the return type, the type of m and the return type are only accidentally the same: we match on n with constructors of nat, thus m must have type nat, and we construct the return value using the constructor S, which also requires the return type to be nat.

This can be observed by adding a *phantom* type variable to the type nat: the type 'a nat_with_tag is the same as nat, but the extra type variable ensures that two unrelated instances of nat_with_tag are *labeled* by different variables:

```
type 'a nat_with_tag = Z_with_tag | S_with_tag of 'a nat
let rec add_with_tag = fun m n → match m with
    | Z_with_tag → n
    | S_with_tag m' → S_with_tag (add_with_tag m' n)
 val add_with_tag : 'a nat_with_tag → 'b nat_with_tag → 'b nat_with_tag
```

Here, we took the arbitrary choice of using the same tag for a natural number and its recursive argument. This is not necessary: indeed, we have seen some examples of ornaments (*e.g.* expr_value in Section 2.3) where the recursive structure of type does not match the original type. During ornamentation, we use the skeleton to express the fact that arguments of constructors may change types. We can similarly use the skeleton for ornament inference. This requires extending the inference to support recursive types. In OCaml, this can be enabled by passing the option −rectypes [Leroy et al., 2020]. The type ('s1, 'tag) nat_skel_tag represents the skeleton of nat, where 's1 is the type of the argument of the constructor S, and 'tag the tag distinguishing separate ornamentations. We name the constructors Z_skel_tag and S_skel_tag in the code. In mathematical notation, we will name the type $\widetilde{\mathsf{nat}}$ and its constructors $\breve{\mathsf{Z}}$ and $\breve{\mathsf{S}}$.

```
type ('s1, 'tag) nat_skel_tag = Z_skel_tag | S_skel_tag of 's1
let rec add_st = fun m n → match m with
  | Z_skel_tag → n
  | S_skel_tag m' → S_skel_tag (add_st m' n)
 val add_st :
  (('a, 'phi) nat_skel_tag as 'a) → (('b, 'psi) nat_skel_tag as 'b) → 'b
```

The type signature at the bottom can be read in the following way: let φ be an ornament with destination type α and skeleton $\widehat{\mathsf{nat}}\ \alpha$, and ψ an ornament with destination type β and skeleton $\widehat{\mathsf{nat}}\ \beta$. Then, we can give add the ornament type $\varphi \to \psi \to \psi$.

Translating to ornamentation constraints

We want to prove that this reinterpretation of the ornamentation constraints translates exactly the constraints in the generalization judgment.

Consider first the case of pattern matching match x with $(d_i(\tau_j)^j(y_{ik})^{k\in K_i} \to b_i)^{i\in I}$ on a type $\zeta\,(\tau_j)^j$ (we assume for simplicity that we are matching on a variable). Let us examine the corresponding elaboration rule, simplified to only

handle matching on local variables:

$$\textsc{E-Match}\quad \frac{\begin{array}{l}(x:\varphi)\in\Gamma \qquad (\varphi\mapsto(\delta,\mathsf{inj},\mathsf{proj})\lhd\hat{\zeta}\,(\omega_{ik})^{i\in I,k\in K_i}:\zeta\,(\tau_j)^j\Rightarrow_)\in\Gamma \\ \left(\begin{array}{l}\Gamma,(x_{ik}:\omega_{ik})^{k\in I_k}, \\ (x\simeq\hat{d}_i((\omega_{\ell k})^+_S)^{\ell\in I,k\in K_\ell}(y_{ik})^{k\in K_i}:\hat{\zeta}\,((\omega_{\ell k})^+_S)^{\ell\in I,k\in K_\ell})\# \\ \quad\vdash^p b_i\rightsquigarrow B_i:\omega\Rightarrow\Delta_i\end{array}\right)_i\end{array}}{\begin{array}{l}\Gamma\vdash^p\mathsf{match}\ x\ \mathsf{with}\ (d_i(\tau_j)^j(y_{ik})^{k\in K_i}\to b_i)^{i\in I} \\ \quad\rightsquigarrow\mathsf{match}\ x\ \mathsf{with}\ (\hat{d}_i((\omega_{ik})^+_S)^{\ell\in I,\ell\in K_\ell}(y_{ik})^{k\in K_i}\to B_i)^{i\in I}:\omega \\ \quad\Rightarrow(\mathsf{match}\ x\ \mathsf{with}\ \hat{d}_i((\omega_{\ell k})^+_S)^{\ell\in I,k\in K_\ell}(x_{ik})^{k\in K_i}\to\Delta_i)^i\end{array}}$$

We suppose that the datatype ζ has constructors $(d_i : \forall(\alpha_j)^j\ (\sigma_{ik})^{k\in I_k} \to \zeta\,(\alpha_j)^j)^{i\in I}$. Its skeleton with tags is the type $\check{\zeta}$ with parameters $((\beta_{ik})^{i\in I,k\in K_i},\varphi)$ and constructors $(\check{d}_i : \forall((\beta_{ik})^{i\in I,k\in K_i},\varphi)\,(\beta_{ik})^{k\in I_k}\to\check{\zeta}\,((\beta_{ik})^{i\in I,k\in K_i},\varphi))^{i\in I}$. We write $\check{b}_i$ the terms obtained by applying the transformation to use the skeleton with tags to the $(b_i)^i$. Then, for the whole term the transformation gives:

$$\mathsf{match}\ x\ \mathsf{with}\ (\check{d}_i((\check{\tau}_{\ell k})^{\ell\in I,k\in K_\ell},\varphi)(x_{ik})^{k\in K_i}\to\check{b}_i)^i$$

where φ is the inferred ornament for x (it will always be a variable because it cannot be unified with anything else than a variable). This expression only generates a constraint that x has type $\check{\zeta}\,((\check{\tau}_{\ell k})^{\ell\in I,k\in K_\ell},\varphi)$ where φ is some ornament variable and the $\check{\tau}_{ik}$ are the types expected by the $\check{b}_i$, and a constraint that all the $\check{b}_i$ return the same type, which are exactly the constraints expressed by E-Match.

Similarly, construction in the skeleton with tags expresses the constraints required by the rule E-Con used for elaborating data constructors.

We also want to show that the term rewritten to use the skeleton with tags always admits a typing: this is enough to show that elaboration is always possible. Inference will then find the most generic term. We can first apply an extension operation to a term: we replace all types by their skeleton and all constructors by the corresponding constructor in the skeleton. For example, any instance of nat becomes the cyclic type $\mu\alpha.\,\widehat{\mathsf{nat}}\ \alpha$ (where the notation $\mu\alpha.\,F(\alpha)$ denotes the recursive type that is the fixed point of F). Similarly, any instance of $\mathsf{S}\ a$ becomes $\hat{\mathsf{S}}\ (\mu\alpha.\,\widehat{\mathsf{nat}}\ \alpha)\ a$. This preserves the typing constraints, since the transformation transforms datatypes in a uniform way, and all constructors are transformed in a compatible way. We then switch to the skeleton with tags by adding type variables to all skeleton types such that two types have the same variable if and only if they have exactly the same parameters. This transformation preserves all equalities between types, and preserves the property that one ornament variable is always linked to exactly one skeleton. Thus, the resulting term is well-typed.

Implementation

The implementation is not based on this transformation: instead, we directly generate the constraints that would be generated with this transformation. We use the Inferno library [Pottier, 2014a,b] to express the constraints in applicative style and solve them.

To reconstruct the final constraints in S, we keep a mapping from ornament variables φ, *i.e.* the variables used to tag skeleton types, to the corresponding skeleton type. When interpreting the solution of the constraints as an ornament type, we ignore the skeleton and only keep the ornament variable.

When generalizing a term, we obtain two kinds of free variables:

- We may get usual type variables α. These variables are copied in the environment $\overline{\alpha}$ containing the type variables in scope of the generalized type.

- We may also quantify over ornament type variables φ. In this case, we look up the corresponding skeleton in the mapping from ornament variables to skeleton, and add it in S with the constraints that its skeleton be the one found in the mapping.

We also need to handle liftings and patches. This is done by keeping a global list R of patches that are required by the type. For each location needing a patch, we take the expected type from the ornament specification, we collect the type of local variables and add a patch taking the local variables as arguments and returning something of the expected type to R. For each location where a global definition is used, we need to choose the ornaments that will be used to instantiate the ornaments in its environment S': we create a fresh ornament variable for each ornament in S' and let them unify with the other ornament variables in the term. Once inference is done, we recover their value and add to the patch environment for the current term R a request for a lifting of the global definition at the ornament specification determined during inference.

We thus obtain a generic term and an ornament specification that we add to the global environment. We immediately add the identity instantiation of this generic term as an already-done lifting and proceed to the next definition.

13.2 Strategies for instantiation

When describing the instantiation of a generic to obtain a lifting (Chapter 12), we assumed that a value was given for all the required ornaments, liftings and patches. In practice, we allow the user to give less information and try to infer the missing information using various strategies. Our strategies consider ornaments and liftings on the one hand, and patches on the other hand, in two separate phases.

The instantiation of ornaments and lifting uses unification (again, based on the type inference engine) to ensure that the provided ornaments and liftings are compatible with the requirements of the generic term and with each other. When lifting a definition of x, we first look up the generic term for x and encode the constraints in the ornament requirements S and the liftings requirements of R into type inference constraints. We then consider some information given by the user:

- The user may specify an ornament specification for the term. This ornament specification may be partial (*i.e.* contain wildcards _ to indicate unconstrained parts). We unify this specification with the specification of the generic term to infer the values of some variables.

- The user may specify the ornaments to be used at some construction and pattern matching locations. We add constraints specifying that the ornament at these locations must be instantiated with the specified ornament.
- The user may specify some liftings to be used at some program points. The choice of the lifting gives us additional constraints on the ornament variables in the generic term.

Once these explicit indications are processed, we apply multiple strategies:

- The user may similarly specify a list of ornaments to be used: for each program point where we still need a datatype ornament, we try each of the specified ornaments in order. This is useful for a refactoring: the user may specify that the ornament describing the refactoring is always to be used. This is expressed with a declaration of the form: **ornament** $* \leftarrow \mathsf{orn}_1, \mathsf{orn}_2, \mathsf{orn}_3$.
- The user may specify a list of liftings to be used in priority order, but not attached to a particular location in the program. Each of these liftings is a lifting of some term y: for each occurrence of y still to be lifted, we try each of these liftings in order and commit to the first matching lifting. This in turn may instantiate some ornament variables. This can be written **lifting** $* \leftarrow \mathsf{f}_1,\ \mathsf{f}_2,\ \mathsf{f}_3$.

The order of application of these strategies is important: one of them might decide on an ornament and prevent the other from applying. For this reason, these clauses are always applied in the order given by the user: they may choose to prioritize one strategy over the other. We do not warn the user if one of these strategies does not apply because some other strategy already instantiated the term (this is in contrast with the hints applied to specific program locations, where we interpret as requirements for the lifted term).

Finally, we propose two other optional mechanisms. The first one exploits unique liftings: after learning of some ornaments, we may end up in a situation where only one known lifting can be used in some location. In this case, the user may elect to have the unique remaining lifting automatically used. This avoids specifying some liftings, assuming that the user already defined all the liftings they may want to use.

The other mechanism allows for *auto-lifting*: if no lifting is available for a location (*i.e.* applying any existing lifting leads to a set of unsatisfiable constraints), the user may ask us to automatically generate a new lifting to fill the spot. This is useful when no patches are needed, for example in the case of refactorings: this way, a user can ask for a refactoring of one function and all functions it depends on will have refactorings automatically generated.

We have much fewer possibilities for patches, because we are not treating patches as typed: instead, we assume that they are well-typed and check the types after reduction to eML and simplification to an ML term. We still perform an approximation of type-directed patch inference: during simplification, if we are matching on an unspecified patch with a pattern matching having a single branch, we assume that the patch must start with this constructor.

Finally, when simplifying the term obtained after lifting and instantiation, some program locations may disappear, for example if we simply remove a constructor and the associated branches: for this reason, we proceed with instantiation even if some ornaments, liftings, or patches are missing. If simplification

eliminates the missing program locations, we accept the lifting. Otherwise, we pretty-print the term with markers indicating where a lifting is missing, what is the name of the location and what kind of object is expected to instantiate this location. For example, the fully generalized version of add is printed as:

```
let rec add_gen m n = match (orn−match #4) m with
    | Z_skel → n
    | S_skel m' → (orn−cons #1) (S_skel (add_gen m' n)) #2
```

The user can deduce that #1 should be instantiated with an ornament of the type nat (since it should transform S_skel the skeleton of S), that the same should be true for #4, and that #2 should be a patch of the type required by the ornament given for #1.

13.3 Refolding terms after the transformation

The transformation described in Chapter 9 to convert *e*ML terms to ML is rather heavy-handed: notably, it separates out possibly effectful computations and removes other let bindings. Moreover, before generalizing the term, we had to transform deep pattern matching into shallow pattern matching. The simplified term might end up rather unreadable. For this reason, after simplifying, we apply a *resugaring* transformation to generate a term closer to the original term.

For re-sharing let declarations, we insert *markers* in the term, one at the site of the original declaration and corresponding markers at every place where the definition has been expanded. When resugaring the term, we look for the markers of declarations: under the declaration, we look up all markers of *uses* of this declaration. If there are no uses left, we do not attempt to restore the definition. Otherwise, we check that the usages would still be well-scoped at the declaration site, compare them for syntactic equality, and re-insert a declaration for each family of equal expressions at use site. Along with the declaration marker, we store the original name of the declaration, so that we can restore it (management of names is detailed later in this section).

For pattern matching, we look at chains of pattern matching in the final term and try, as much as possible, to gather them in one pattern matching. We then compare branches: if some branches have the same code, we use an alternative pattern to regroup them. We do not add back wildcard patterns to match all remaining cases, even if they were present in the original term. We also do not preserve the order of the patterns: this could be done by adding a mark on the branches indicating which appeared first in the source term. For example, consider the input term:

```
let rec eval e = match e with
  | App (u, v) →
      begin match eval u with
        | Some (Abs f) → eval (f v)
        | _ → None
      end
  | _ → Some(e)
```

If we ask for its identity lifting, we obtain the following output term. The order of the branches has been lost and the wildcard pattern has been replaced by a pattern listing all alternatives, but the branches that were merged are still merged:

```
let rec eval_id e = match e with
  | (Abs _ | Const _) → Some e
  | App(u, v) →
    begin match eval_id u with
      | Some (Abs f) → eval_id (f v)
      | ((Some (App(_, _)) | Some (Const _)) | None) →  None
    end
```

Finally, we insert a specific mark on let bindings introduced to lift effectful computations. When we encounter such a binding, we try to insert the computation as close to its use-site in the term without reordering function applications. For example, if f1 and f2 are two functions, consider the following term and its identity lifting:

```
let h t = (f1 t, f2 t)
let h' = lifting h with ornament ∗ ← @id
```

Since constructor arguments are evaluated from right to left, during its elaboration, it is transformed to:

```
let h' t =
  let x = f2 t in
  let y = f1 t in
  (x, y)
```

The resugaring transform will then fold the two function calls back into the construction:

```
let h' t = (f1 t, f2 t)
```

Suppose now that we apply an ornament that reverses the members of a tuple:

```
type ornament ('a, 'b) rev : ('a ∗ 'b) ⇒ ('b ∗ 'a) with
    | (x,y) ⇒ (y,x)
let h'' = lifting h with ornament ∗ ← @id
```

We cannot refold both applications into the tuple construction, because they would run in the reverse order. Instead, we obtain the following term: the second applications could be folded into the construction, then the first one is blocked because if it were lifted in its position it would execute before the second one.

```
let h'' t =
  let x = f2 t in
  (x, f1 t)
```

All these transformations operate on abstract names, represented by unique integers. To generate readable code at the end, we store alongside these names the name they had in the original program. This might be a set of names, as some variables can end up identified during the transformation (for example if we match twice on the same value). Names can come both from the base term and from the pattern in the ornament definition. We then look at the names that must be available in each scope to select a name formed by one of the original names followed by a numeric prefix. If possible, we prefer shadowing variables to using the same variable with a prefix.

Taken together, these transformations generate reasonable-looking codes on our (admittedly simple) examples. To use ornamentation to transform codebases and continue working on the transformed codebase, we would have to also

be able to preserve comments. This is usually done by attaching them to nodes of the abstract syntax tree. We do output pretty-printed term, but we expect that the user would prefer running a code formatter configured with the recommended style for their project as a post-processing phase before committing the code to source control.

Chapter 14

Unfolding of definitions

The presentation of lifting given before has one major restriction: we cannot change the recursive structure of a function. This is because recursive calls refer to the function being defined as a local variable, so they can only refer to the function being currently lifted.

14.1 Unfolding the recursive structure

This sometimes prevents us from writing interesting liftings of functions. For example, consider the map function on lists:

```
type 'a list = Nil | Cons of 'a * 'a list
let rec map f l = match l with
  | Nil → Nil
  | Cons(x, xs) → Cons(f x, map f xs)
```

We can define a type of chunked lists, where each constructor contains two elements. We define the ornament using the low-level syntax for ornaments, and use an auxiliary ornament to transform lists of odd length into an element and a list of even length.

```
type 'a chunked_list = Nil2 | Cons2 of 'a * 'a * 'a chunked_list
type ornament 'a chunk_list : 'a list ⇒ 'a chunked_list with
  | Nil ⇒ Nil2
  | Cons(x1, (x2, xs) : 'a odd_list) ⇒ Cons2(x1, x2, xs)
and 'a odd_list : 'a list ⇒ 'a * 'a chunked_list with
  | Nil ⇒ ∼  (* empty lists are not odd *)
  | Cons(x1, xs) ⇒ (x1, xs) when xs : 'a chunk_list
```

The syntax (x2, xs) : 'a odd_list is used to require this argument to be lifted with the ornament 'a odd_list and then matching on the result of the lifting. This ornament relates, *e.g.*, the list Cons $(1, \mathsf{Cons}\ (2, \mathsf{Nil}))$ to Cons2 $(1, 2, \mathsf{Nil2})$.

Given such an ornament, we cannot lift the function map: we quickly realize that the type of the recursive call must be the same as the type of map because it is defined as a fixed point, while the ornament requires the recursive call to be ornamented with odd_list.

Our solution is to optionally consider recursive calls as calling global variables: then, a recursive call can be ornamented by any lifting of the function, not

necessarily the lifting we are currently defining. In the case of map, we simultaneously define two mutually recursive liftings of the single-recursive function map:

```
(* Run with option --unfold *)
let chunk_map = lifting map : _ → _ chunk_list → _ chunk_list
and chunk_map_odd = lifting map : _ → _ odd_list → _ odd_list
```

We obtain the following result:

```
let rec chunk_map f l = match l with
    | Nil2 → Nil2
    | Cons2(x, x2, xs) →
      begin match chunk_map_odd f (x2, xs) with
        | (x2, xs) → Cons2(f x, x2, xs)
      end
 and chunk_map_odd f l = match l with
    | (x, xs) → (f x, chunk_map f xs)
```

With autolifting, we do not even need to request a lifting for odd lists:

```
(* Run with --unfold --autolift *)
let chunk_map = lifting map : _ → _ chunk_list → _ chunk_list
```

We are not limiting the number of generated liftings. In some cases this process could generate liftings recursively, never stopping.

14.2 Unfolding for specialization

We can use unfolding to specialize a program to a more restricted version of a datatype. Continuing with our example of lists and the map functions, we might want to ornament lists to *e.g.* three-element tuples representing coordinates:

```
type 'a vec3 = Vec3 of 'a * 'a * 'a
type ornament 'a list3 : 'a list ⇒ 'a vec3 with
  | Nil ⇒ ∼
  | Cons(x, (y, z) : 'a list2 ) ⇒ Vec3(x, y, z)
and 'a list2 : 'a list ⇒ 'a * 'a with
  | Nil ⇒ ∼
  | Cons(y, z : 'a list1 ) ⇒ (y, z)
and 'a list1 : 'a list ⇒ 'a with
  | Nil ⇒ ∼
  | Cons(z, () : 'a list0 ) ⇒ z
and 'a list0 : 'a list ⇒ () with
  | Nil ⇒ ()
  | Cons(x, xs) ⇒ ∼
```

The last definition list0 is useful because we cannot simply ignore the tail of the list, but we can ignore a value of type unit by matching it. With this definition, we can ask for a lifting of map operating only on three-elements vectors. This has two advantages: first, it is obvious from the type that this function takes a three-element vector and returns a three-element vector. The second advantage is that a three-element vector has a much more compact memory representation, avoiding indirection, and the functions operating on them do not need pattern matching. We can again require an automatic lifting:

```
(* Requires --unfold --autolift *)
let map_vec3 = lifting map : ('a → 'b) → 'a list3 → 'b list3
```

We obtain the following result:

```
let rec map_vec3 f x = match x with
    | Vec3(x, y, z) →
      begin match map_auto1 f (y, z) with
        | (y, z) → Vec3(f x, y, z)
      end
 and map_auto1 f x = match x with
    | (x, xs) → (f x, map_auto2 f xs)
 and map_auto2 f x = match map_auto3 f () with
    | () → f x
 and map_auto3 f x = match x with
    | () → ()
```

This definition is made harder to read by the many intermediate function calls and pattern matchings. If our objective is to generate human-readable code, we can inline some definitions as a post-processing phase. We provide an option for this, that attempts to inline all functions left from unfolding (but leaves all definitions that were explicitly specified by the programmer). We obtain the following inline code:

```
let rec map_tuple f x = match x with
    | Vec3(x, y, z) → Vec3(f x, f y, f z)
```

14.3 Generic programming with unfolding

The fact that unfolding allows for more specialization could also be used for generic programming. Suppose that we represent the structure of containers with the following type, equipped with a map function:

```
type 'a gen = Pair of 'a gen ∗ 'a gen | Value of 'a | Unit
let rec gen_map f x = match x with
  | Pair(u,v) → Pair(gen_map f u, gen_map f v)
  | Value x → Value (f x)
  | Unit → Unit
```

We can express lists as an ornament of this type:

```
type 'a list = Nil | Cons of 'a ∗ 'a list

type ornament 'a gen_list : 'a gen ⇒ 'a list with
  | Unit ⇒ Nil
  | Pair(x, xs) ⇒ Cons(x, xs) when x : 'a gen_elem, xs : 'a gen_list
  | Value x ⇒ ∼
and 'a gen_elem : 'a gen ⇒ 'a with
  | Value x ⇒ x
  | Unit ⇒ ∼
  | Pair(x, y) ⇒ ∼
```

Then, we can ask for a lifting of the map function to operate on lists and recover the map function on lists:

```
let map_list = lifting gen_map : ('a → 'b) → 'a gen_list → 'b gen_list
```

```
let rec map_list f x = match x with
    | Nil → Nil
    | Cons(u, v) → Cons(gen_map_auto1 f u, map_list f v)
 and gen_map_auto1 f x = f x
```

If we ask for the generated definitions to be inlined, we obtain the following code:

```
let rec map_list f x = match x with
    | Nil → Nil
    | Cons(u, v) → Cons(f u, map_list f v)
```

Similarly, we can express an ornament from the generic type to binary trees, and obtain a map function by lifting:

```
type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree

type ornament 'a gen_tree : 'a gen ⇒ 'a tree with
  | Unit ⇒ Leaf
  | Pair(v : 'a gen_value, (l, r) : 'a gen_treepair) ⇒ Node(v,l,r)
  | Value x ⇒ ∼
and 'a gen_value : 'a gen ⇒ 'a with
  | Value x ⇒ x
  | Unit ⇒ ∼
  | Pair(x,y) ⇒ ∼
and 'a gen_treepair : 'a gen ⇒ 'a tree * 'a tree with
  | Unit ⇒ ∼
  | Pair(l : 'a gen_tree, r : 'a gen_tree) ⇒ (l, r)
  | Value x ⇒ ∼

let map_tree = lifting gen_map : ('a → 'b) → 'a gen_tree → 'b gen_tree
```

This produces the following code:

```
let rec map_tree f x = match x with
    | Leaf → Leaf
    | Node(u, l, r) →
      begin match gen_map_auto2 f (l, r) with
        | (l, r) → Node(gen_map_auto3 f u, l, r)
      end
 and gen_map_auto2 f x = match x with
    | (u, v) → (map_tree f u, map_tree f v)
 and gen_map_auto3 f x = f x
```

With unfolding, this simplifies to:

```
let rec map_tree f x = match x with
    | Leaf → Leaf
    | Node(u, l, r) → Node(map_tree f u, f l, f r)
```

We can write other operations on the generic representation and lift then to trees and lists, for example a fold operation:

```
let rec gen_fold z f x = match x with
  | Pair(u, v) → f (gen_fold z f u) (gen_fold z f v)
  | Value x → x
  | Unit → z

let fold_list = lifting gen_fold : _ → _ → _ gen_list → _
```

The following code is produced:

```
let rec fold_list z f x = match x with
    | Nil → z
```

```
    | Cons(u, v) →
      f (gen_fold_auto z f u) (fold_list z f v)
and gen_fold_auto z f x = x
```

This can be simplified by inlining to:

```
let rec fold_list z f x = match x with
    | Nil → z
    | Cons(u, v) → f u (fold_list z f v)
```

Similarly, if we ask for a lifting to trees, we obtain the following code after inlining:

```
let rec fold_tree z f x = match x with
    | Leaf → z
    | Node(x, l, r) →
      f x (f (fold_tree z f l) (fold_tree z f r))
```

Chapter 15

Extensions and future work

In this manuscript, we presented a theoretical framework for ornamentation, and we illustrated its usefulness with a proof of concept implementation of ornamentation. This presentation is purposefully restricted to focus on the core ideas behind the transformation. The results obtained on this restricted field are very encouraging and invite us to consider how this refactoring technique could be scaled to a wider family of cases. The theoretical framework has been explored in a core subset of ML, and on a restricted class of transformations: there is some work remaining to adapt it to handle more language features. The implementation we propose is only a proof of concept, with only the minimal features needed to demonstrate ornamentation. It only operates on a restricted subset of OCaml. Operating on this subset is already useful, but there is a lot to do to scale ornamentation to larger examples using more language features. Although they work remarkably well, our strategies to output readable code are also an interesting area for further improvements. This remaining work is extensive, and involve both interesting research topics and extensive design and engineering work, but our results indicate that scaling up ornamentation should be very rewarding. We detail some of these challenges in this chapter.

15.1 Handling effects

OCaml programs very often feature effects, in the form of non-termination, exceptions, references, and interaction with external systems. Our presentation of ornamentation only handles non-termination. In practice, programmers will want to ornament effectful programs, and thus require an ornamentation procedure that preserves a wider range of effects.

In fact, the ornamentation procedure has been carefully designed so as to preserve effects, assuming only function calls can carry side effects. This can be intuitively understood by the following reasoning: first, two instantiations of a generic term perform the same side effects (when called with related arguments) up to the side effects performed in patches: all functions are called in the same order in the same branches and with related arguments. The side effects performed by patches are interleaved between the side effects performed by the base term. Then, the term is transformed both on the left and the right-hand sides, preserving the full term equality. But full term equality also preserves

effects: it precisely keeps the order of function calls and their arguments stay equal. Thus, effects should be preserved throughout the ornamentation process.

Our implementation does respect the ordering imposed by full term equality, and thus does preserve the existence and the ordering of side-effects. We are still missing a presentation of effects in mML and results asserting that they are preserved by both the logical relation and the program transformations. We would also like to explicitly handle control effects such as exceptions and continuations.

15.2 Generalizing the ornamentation relations

We limited the relations we consider on datatypes to only adding and reorganizing information. This is obviously not the only kind of transformation on datatypes that a programmer might want to perform. Baudin and Rémy [2018] extend ornaments to also allow for disornamentation, *i.e.* removing information from a datatype. They build on the theory described in this manuscript to allow writing relations such as rev_add_loc below, removing location information from an abstract syntax tree, and lifting functions along these relations.

```
type relation rev_add_loc : (expr' ∗ location) ⇒ expr with
  | (Abs' f, _) ⇒ Abs f when f : (rev_add_loc → rev_add_loc)
  | (App' (u, v), _) ⇒ App (u, v)  when u v : rev_add_loc
  | (Const' i, _) ⇒ Const i
```

This relation is not an ornament, since it removes information that cannot be recovered: indeed, it is a *disornament* as its inverse transformation would be an ornament. Their system actually generalizes ornamentation: it allows writing both ornaments, disornaments, and compositions of ornaments and disornaments in a uniform syntax.

Disornamentation can be used to remove data that is no longer useful, or to provide a simplified view into a program manipulating different kinds of data at once to focus on only understanding and editing one aspect of the program. For example, the disornament above can be used to erase location information in a compiler manipulating a toy language: the programmer would then be able to see and edit code that does not manipulate the locations. After performing the desired changes, the programmer may want to port their change back to the original version of the compiler. One way to do this is to generate during the disornamentation a set of patches explaining how to reornament the program to manipulate locations again: these patches can then be applied when possible, leaving the programmer with only a few locations to fill in after reornamentation.

The example of reornamentaion is very interesting in itself, since it may offer a model to abstract away some irrelevant details when manipulating complex programs. It also shows that our framework is not exclusively limited to ornamentation and can handle at least some other transformations: we can handle disornamentation and mixed ornamentation solely by adding an argument to the function defining projection of a type before pattern matching, and propagating the corresponding requests for patches. Ornamentation and disornamentation can be freely mixed, and all the theory we developed then immediately extends to this case. Further exploration is needed to determine what other interesting transformations could be encoded, or what obstacles exist to encoding them.

15.3 Improving the patching language

The patching language we propose is very limited. We identify two main issues with it. The first one is that patches are tied to program locations expressed as integers. These locations are not at all robust to modifications of the source term, even modifications that should not change the meaning of the patches. To avoid this issue, Baudin and Rémy [2018] propose to match on the syntax of the term to identify where patches should be inserted. For example, consider the incomplete version of append obtained from the addition function on natural numbers:

```
let rec append_incomplete m n = match m with
  | Nil → n
  | Cons(_, m') → Cons(#3, append_incomplete m' n)
```

There is a single hole #3, which may be selected with any of the following patterns:

- #[...]
- Cons(#[...], _)
- **match** _ **with** | Cons(_, _) → Cons(#[...], _)

The code pattern can be used to capture the variables used in the patch, instead of relying on the names in the original term. Here, the hole should be filled with the first argument of the constructor Cons that comes from the destruction of m. Then, the patch could be expressed as follows, with the patch inserted inside the square brackets appearing in the pattern:

```
patch match _ with | Cons(a, _) → Cons(#[a], _)
```

The other issue is that most patches need to be written manually. It would be interesting to explore some strategies to generate patches, for example by exploiting typing information to deduce that only one value of the desired type could be constructed in the context [Scherer and Rémy, 2015], or by exploiting strategies such as copying the latest value of the appropriate type introduced in the context. Such a strategy could be enough to propagate source locations in an abstract syntax tree.

15.4 Scaling up the prototype

Our small prototype only works on small subset of OCaml and on small examples. Porting it to work on the full OCaml language would be a large amount of work, and would come with significant challenges. Ornament inference is based on type inference: the simplest way to implement ornament inference for OCaml might be to base it on type inference. Most alterations to the type-checker should be local to code handling data types and type constructors. Then, lifting would be based on a suitably-annotated version of the typed abstract syntax tree. Some language constructions would require special care:

Modularity Large programs tend to be split into different modules: some of these modules might be created by the programmer, others might be part of a library. In both cases, the lifting should be done following the structure of the existing modules: one module can be lifted to its lifted version, producing both the lifted code and a lifting interface similar to the module interface, but describing the relation between two versions of the module. This lifting interface could then be used as an input to the lifting process for modules depending on the first one. One can imagine that library maintainers wanting to update the interface of their library could provide such a file, allowing all clients of the library to semi-automatically lift their programs to use the newest version of the library.

The OCaml module system has rich abstraction constructs (such as functors and first-class modules). These would need special care. Module used as arguments of functors often provide an abstract datatype and operations on this datatype: we can replace them with a module providing an ornament of the original datatype and implementations of these operations related to the implementations on the base type.

GADTs Programming with generalized algebraic datatypes (GADT) requires writing multiple definitions of the same type holding different invariants. GADT definitions that only add constraints could be considered ornaments of regular types, which was one of the main motivations for introducing ornaments in the first place [Dagand and McBride, 2014]. It would then be useful to automatically derive, whenever possible, copies of the functions on the original type that preserve the new invariants. Extending our results to the case of GADTs is certainly useful but still challenging future work. GADTs integrate both existential types and type equalities: we would have to add both into eML and mML. On the practical side, we would have to consider how a definition of an ornament from a GADT might be expressed, and ensure that the procedure for simplifying from eML terms to terms that type in ML is compatible with GADTs. It might also be necessary to add type annotations inside the lifted term to make it type: we would have to provide a mechanism to insert them.

Extensible variants, objects, polymorphic variants, and exceptions OCaml provides some other datatype-like constructs, including objects, extensible variants and polymorphic variants. It would be interesting to accept them as both source and destination types for ornaments. For polymorphic and extensible variant, this could be expressed with a default case that transforms all constructors that were not mentioned using the identity ornament. Exceptions are represented by a global extensible variant. Ornamenting the exception type means, in essence, changing the global exception type used by some functions: then, functions using this new exception type would be incompatible with functions using the normal exception type. This is acceptable for full program refactoring, but it might be possible to find a finer strategy (for example, terms with different ornaments of the exception type become compatible again once all ornamented exception constructors are captured).

Part V

Conclusion

Chapter 16

Related works

16.1 Ornaments

The notion of ornament has been introduced by McBride [2010] in context of dependently-typed programming language. Here, ornaments are described in a framework where a universe of datatypes [Chapman et al., 2010] is encoded in the Agda programming language, allowing descriptions of indexed datatypes to be manipulated as first-class objects. Building on this description, ornaments can be described as patches over existing datatypes, adding fields and enriching their indexing. Because they respect the indexing structure of the base datatypes, ornaments describe an *algebra* allowing for example to forget the elements of a list to get a natural number. This ornamental algebra can be used to index lists, building vectors as an *algebraic ornaments* of lists.

Working in a dependently-typed programming language and over a universe of datatypes allows the flexibility to describe ornaments as first-class objects: this allows the programmer to re-use the concept of ornaments, adapting it to their specific use. On the other hand, the encoding induces a lot of syntactic overhead and makes it impossible to work on programs that operate on the native datatypes. We instead chose to have ornaments operate as a second-class construct in a tool external to the language and to the compiler: this allows ornamentation on practical programs within the constraints of ML. We lose the ability to describe ornaments as diffs: instead, we define ornaments as relations between two pre-existing types. We could recover this ability by adding a facility to generate an ornamented type and a relation by describing what data should be added on what constructors.

Finally, the presentation of ornaments in this article, and in subsequent work, is very much rooted in the presentation of datatypes as fixed points of functors: ornaments must respect the recursive structure of datatypes. In ML, the recursive structure of a datatype is less obvious: for example, when considering two mutually recursive datatypes, the recursion steps could be inserted only when coming back to the first type, or between each of the types. Using the notion of skeleton and recursive types during ornament inference, we do not have to decide on a recursive structure for each type; the way the type is to be interpreted is instead implicitly described by the user when giving the recursive structure of the ornamentation relation.

The idea of lifting functions across ornaments was introduced by Dagand and McBride [2014], along with the coherence property explaining the meaning of lifting. Again, this is done by defining a universe of functions, and a notion of functional ornament following the original signature. A notion of patch is then defined, using algebraic ornaments to have a convenient structure isomorphic to a function along with its coherence proof. By writing a term of this type, the programmer builds a correct by construction lifting. The authors then propose following the structure of the base term to build the patch: a user can build a patch reproducing the recursive structure of the base term, and then choose to lift the constructors already present. They propose an interactive mechanism to define such liftings. This idea formed the base of our exploration of semi-automatic ornamentation in ML.

In this context, Ko and Gibbons [2016] explore the use of ornaments, still using universes of datatypes, to define data structures (*e.g.* binomial trees) as liftings of representations of binary numbers. They also define a notion of ornamentation for higher-order functions by considering an ornamentation relation between functions that is not based on projection.

Dagand [2017] presents a more general model of what ornamentation means: datatypes are represented as many-sorted signature and ornaments as cartesian morphisms on these signatures. This fits somewhat better the types available in ML, although our treatment of ornaments allows ornamentation below function arrows, even in negative position. From this categorical presentation, the author describes how a number of categorical concepts translate to operations on ornaments. Notably, vertical composition creates an ornament from two ornaments applied in sequence, and a categorical pullback would combine the data and constraints added by two ornaments and thus allow building types from base types and multiple ornamentations. These forms of composition could be integrated to our prototype implementation, and would be especially useful for GADTs, where one would want to combine multiple invariants into a single type.

On the practical side, Ringer et al. [2019] propose Devoid, an implementation of ornamentation as a plugin for Coq to allow for proof reuse. The plugin approach works directly on the Coq language and does not require an embedding of datatypes to allow their manipulation. Their work is concerned only with algebraic ornaments. They propose a way to automatically search for an ornament between two types, limiting the search by requiring that the types have the same constructors in the same order, with additional arguments accounting for the indexing. These ornaments can then be used to lift Coq terms, generating along the way a proof that the lifted terms and the base terms behave in the same way up to indexing. An automatic search for ornaments would also be interesting in ML, particularly for GADTs: when only adding constraints, without adding or reorganizing data, constructors are simply replaced with the matching constructor of the ornamented type. It would be good to ensure the programmer does not have to explicitly write such definitions.

16.2 Refactoring

The notion of refactoring has mainly been considered in context of object-oriented programming languages [Mens and Tourwe, 2004].

HaRe, the Haskell Refactorer [Li et al., 2005], is a refactoring tool for Haskell

that automates a variety of operations (adding arguments, renaming, moving definitions across modules, *etc.*). The tool also provides an API allowing the programmer to define their own transformations. More recently, the ROTOR tool [Rowe et al., 2019] has been developed to automatically perform renaming of top-level declarations in OCaml. Renaming a top-level declaration can induce many other renamings: functors take module of a given signature, and renaming an identifier in a module given as argument to a functor requires renaming the same identifier in the signature, and in all modules used in the same position. The transformation is based on a formal semantics of names in the module system. Cross-module lifting in OCaml would be affected by similar issues (indeed, lifting may already be used as a form of renaming), where we additionally would have to take ornament types into account. This shows that handling ornamentation across modules requires some pretty involved work, although one might be able to rely on the analysis underlying such a tool to propagate ornamentation constraints.

Robert [2018] considers the problem of refactoring proofs in dependently-types language, where the goal is to replace the original code by some code taking into account a modification. His approaches takes as input the original program and a *partially-refactored* program. From this partially-refactored program, a *diff* is inferred, representing the modifications applied by the programmer. This diff is then *repaired* by propagating changes from definitions to use-sites. The diff language considered in this work handles renaming of constants and variables, changing the number, order and type of function arguments, and changing the definition of inductive types, including addition and removal of constructors and arguments. The repair process can leave *holes* in the patches that should be filled by the programmer. This repair process is extended to richer languages by *embedding* them in the core language: the richer constructs are ignored during the transformation in the hope that they do not interfere with the repair that is in progress. This is unsafe, since there is no way to detect that an embedded construct should have been repaired, but still helps in automating part of a refactoring. The embedding from the surface language to the core language used in the transformation loses some information. To preserve as much as possible the surface syntax of the original term, the abstract syntax tree on which the transformation is performed is decorated with extra information indicating the original shape of the term. This information is attached to the abstract syntax tree and preserved in a systematic way during patching (following the "Trees that grow" technique, Najd and Peyton-Jones [2016]). This is similar to the mark we add in terms, although we do not take such a principled approach to attaching this information: we instead have a marking construct where the mark is taken from an extensible type. Finally, to preserve the layout of the code, the repaired code is pretty-printed by following simultaneously the original code and the new code.

PUMPKIN PATCH [Ringer et al., 2018] proposes a solution to the problem of patching *proof scripts* instead of terms. The idea is to examine existing adaptations to a change in a definition, and infer corresponding patches by looking at the downstream modifications. The patches are found in a type-directed way, by looking for term fragments that have the type of the desired transformation. The user can then reuse these patches to fix subsequent proofs.

16.3 Porting operations to similar datatypes

Type Theory in Color [Bernardy and Moulin, 2013] is another way to understand the link between a base type and a richer type. Some parts of a datatype can be tainted with a color modality: this allows tracing which parts of the result depend on the tainted values and which are independent from them. Terms operating on a colored type can then be erased to terms operating on the uncolored version, which would correspond to the base term. This is internalized in the type theory: in particular, equalities involving the erasure hold automatically. This is the inverse direction from ornaments, and thus more related to disornamentation (§15.2). Once the operations on the ornamented datatype are defined, the base functions are automatically derived, as well as a coherence property between the two implementations. Moreover, the range of transformations supported by type theory in color is more limited: it only allows field erasure, but not, for example, to rearrange a product of sums as a sum of products. Interestingly, their definition supports a notion of composition as there can be multiple colors than can be separately projected out.

Programming with GADTs may require defining one base structure and several structures with some additional invariants, along with new functions for each invariant. Ghostbuster [McDonell et al., 2016] proposes a *gradual* approach to porting functions to GADTs with richer invariants, by allowing as a temporary measure to write a function against the base structure and dynamically checking that it respects the invariant of the richer structure, until the appropriately typed function is written. While the theory of ornaments supports adding invariants, we do not yet support GADTs. Moreover, we propose ornaments as a way to generate new code to be integrated into the program, rather than to quickly prototype on a new datatype.

Dagand et al. [2018] also consider the problem of interacting between code written to operate on dependent types and code that operates on unrestricted *simple* types. They propose a solution based on inserting dynamic checks at appropriate boundaries. Notably, this allows the safe interaction of code extracted from a dependently-typed language and code that may generate invalid values because it operates on less-restricted types. Their framework is based on Galois connections between simple and dependent types, which offer a very flexible notion of connection.

Najd and Peyton-Jones [2016] also observe that one often needs many variants of a given data structure (typically an abstract syntax tree), and corresponding functions for each variant. They propose a programming idiom to solve this problem: they create an extensible version of the type, and use type families to determine from an extension name what information must be added to each constructor. In this approach, the type of the additional information only depends on the constructor, while our type-level pattern matching allows depending on the information stored in the already-present fields. Their approach uses only existing features of GHC, avoiding a separate pre-processing step and allowing one to write generic functions that operate on all decorations of a tree. On the other hand, the programmer must pay the runtime and readability cost of the encoding even when using only the undecorated tree. The encoding of extensible trees scales naturally to GADTs. Interestingly, this idiom and ornaments are largely orthogonal features with some common use case (factoring operations working on several variants of the same datatype) and might

hopefully benefit from one another.

Views, first proposed by Wadler [1986] and later reformulated by Okasaki [1998] have some resemblance with isomorphic ornaments. They allow several interchangeable representations for the same data, using isomorphisms to switch between views, but, again, the isomorphisms persist at runtime, which gives views applications that are quite different from those of ornamentation.

Chapter 17

Conclusion

In this work, we propose the notion of ornamentation as an interesting kind of refactoring, allowing for many useful examples that could free programmers from tedious and error-prone hand-refactoring. We use ornament types as specifications for these refactorings. The ornament types are a binary counterpart to usual types: they relate two different versions of a program, allowing us to reason about program transformations.

To implement this transformation and prove its correctness, we define a richer language mML and follow a process where from a base term we compute a generic term that can be instantiated with any ornamentation. We then use parametricity of mML to relate two instantiations of the generic term: one with identity ornaments, giving back the base term, and one with the user-specified ornaments. The instantiation with user-specified ornaments is then simplified and printed to the user.

We demonstrate a prototype implementation based on this process: we use it to illustrate how ornaments are useful on a variety of examples. One, maybe surprising, result is that, even though the transformation does a lot of modifications on the term, we manage, with a bit of marking, to recover a term close to the original term.

Many of the ideas exposed here can be reused for other transformations, as illustrated by the example of disornamentation. To allow the writing of the generic term, we construct a stack of languages over core ML: we add labels, then equalities to form eML, and meta-abstractions to form mML. The equalities and reasoning features of eML are closely related to the fact that, for ornamentation, the types of the patches depend on the shape of the term we are constructing. Other transformations might require adding some features in eML, or might not require equalities at all (that would be the case if we were only interested in pure refactorings that do not add or remove information). Labels are a very convenient way to transport information about the result of evaluating some language features that would be tricky to reason about (or that we are simply not interested in), such as side-effects: this should be useful when building other transformations. Similarly, the abstractions offered by mML are necessary as soon as the abstractions needed for the generic term cannot be expressed in ML, and they moreover ensure that traces of the encoding are eliminated from the term.

The type-like specifications we use to express the relation between the orig-

inal and the refactored terms can be extended to encode many other transformations, such as adding and removing arguments, or defunctionalization. The idea of inferring a most general transformed term that then needs to be instantiated allows incremental interaction with the transformation: a user can see a partial result and deduce from there what next steps they should take to obtain the desired result. On the other hand, it seems harder to scale this to offer a wide range of transformations at once: the term would quickly become too abstract and unintelligible. One may want instead to guide the generation of the abstract, generic term based on the (partial) specification provided by the user. The generic term plays another key role: by being an abstraction over both the base term and the ornamented term, it allows us to only prove that the basic building blocks of the ornamentation, *i.e.* the function describing datatype ornaments, are related at the correct specification, and to deduce by parametricity that the whole base term and ornamented terms are related. This reasoning principle should generalize to many transformations.

Providing programmers with powerful transformations they can apply as needed allows for faster development of higher-quality programs, and give them the ability to easily experiment with restructuring their code. This manuscript describes one possible transformation that is already useful by itself, but its building blocks are also hopefully reusable to construct other refactoring tools.

Bibliography

Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming*, pages 69–83. Springer, 2006.

Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23 (5):657–683, September 2001. ISSN 0164-0925. doi: 10.1145/504709.504712. URL `https://doi.org/10.1145/504709.504712`.

Lucas Baudin and Didier Rémy. Disornamentation. In *ML 2018 - ML Family Workshop*, St. Louis, Missouri, United States, September 2018. URL `https://hal.inria.fr/hal-02001629`.

Jean-Philippe Bernardy and Guilhem Moulin. Type-theory in color. In *International Conference on Functional Programming*, pages 61–72, 2013. doi: 10.1145/2500365.2500577.

Richard Bird and Lambert Meertens. Nested datatypes. In *International Conference on Mathematics of Program Construction*, pages 52–67. Springer, 1998.

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, page 3–14, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605587943. doi: 10.1145/1863543.1863547. URL `https://doi.org/10.1145/1863543.1863547`.

Pierre-Évariste Dagand. The essence of ornaments. *Journal of Functional Programming*, 27:e9, 2017. doi: 10.1017/S0956796816000356.

Pierre-Évariste Dagand and Conor McBride. A categorical treatment of ornaments. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 530–539, 2013. doi: 10.1109/LICS.2013.60. URL `http://dx.doi.org/10.1109/LICS.2013.60`.

Pierre-Évariste Dagand and Conor McBride. Transporting functions across ornaments. *J. Funct. Program.*, 24(2-3):316–383, 2014. doi: 10.1017/S0956796814000069. URL `http://dx.doi.org/10.1017/S0956796814000069`.

Pierre-Evariste Dagand, Nicolas Tabareau, and Éric Tanter. Foundations of Dependent Interoperability. *Journal of Functional Programming*, 28, March 2018. doi: 10.1017/S0956796818000011. URL `https://hal.inria.fr/hal-01629909`.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989. ISBN 0-521-37181-3.

Hsiang-Shang Ko. *Analysis and synthesis of inductive families*. Dphil dissertation, University of Oxford, 2014.

Hsiang-Shang Ko and Jeremy Gibbons. Programming with ornaments. *Journal of Functional Programming*, 27, 2016. doi: 10.1017/S0956796816000307.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 4.11: Documentation and user's manual. Technical report, Inria, August 2020. URL `https://caml.inria.fr/pub/docs/manual-ocaml/`.

Huiqing Li, Simon Thompson, and Claus Reinke. The haskell refactorer, hare, and its api. *Electronic Notes in Theoretical Computer Science*, 141 (4):29 – 34, 2005. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2005.02.053. URL `http://www.sciencedirect.com/science/article/pii/S157106610505173X`. Proceedings of the Fifth Workshop on Language Descriptions, Tools, and Applications (LDTA 2005).

Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, page 35–46, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580623. doi: 10.1145/1411304.1411311. URL `https://doi.org/10.1145/1411304.1411311`.

Conor McBride. Ornamental algebras, algebraic ornaments. *Journal of functional programming*, 47, 2010. URL `https://personal.cis.strath.ac.uk/conor.mcbride/pub/OAAO/LitOrn.pdf`.

Trevor L. McDonell, Timothy A. K. Zakian, Matteo Cimini, and Ryan R. Newton. Ghostbuster: A tool for simplifying and converting gadts. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 338–350, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951914. URL `http://doi.acm.org/10.1145/2951913.2951914`.

T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

Shayan Najd and Simon Peyton-Jones. Trees that grow. *JUCS*, 2016. URL `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/11/trees-that-grow-2.pdf`.

Chris Okasaki. Views for standard ml. In *In SIGPLAN Workshop on ML*, pages 14–23, 1998.

François Pottier. Hindley-Milner Elaboration in Applicative Style. In *ICFP 2014: 19th ACM SIGPLAN International Conference on Functional Programming*, Goteborg, Sweden, September 2014a. ACM. doi: 10.1145/2628136.2628145. URL `https://hal.inria.fr/hal-01081233`.

François Pottier. Inferno: a library for Hindley-Milner type inference and elaboration, February 2014b. URL `http://gallium.inria.fr/~fpottier/inferno/inferno.tar.gz`.

Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Adapting proof automation to adapt proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 115–129, 2018.

Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Ornaments for proof reuse in coq. In *10th International Conference on Interactive Theorem Proving*, 2019.

Valentin Robert. *Front-end tooling for building and maintaining dependently-typed functional programs*. Phd dissertation, University of California San Diego, 2018.

Reuben NS Rowe, Hugo Férée, Simon J Thompson, and Scott Owens. Characterising renaming within ocaml's module system: theory and implementation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 950–965, 2019.

Gabriel Scherer and Didier Rémy. Which simple types have a unique inhabitant? In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 243–255, 2015. doi: 10.1145/2784731.2784757. URL `http://gallium.inria.fr/~remy/focusing/`.

Masako Takahashi. Parallel reductions in λ-calculus. *Journal of Symbolic Computation*, 7(2):113 – 123, 1989. ISSN 0747-7171. doi: https://doi.org/10.1016/S0747-7171(89)80045-8. URL `http://www.sciencedirect.com/science/article/pii/S0747717189800458`.

Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. Let should not be generalized. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '10, page 39–50, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605588919. doi: 10.1145/1708016.1708023. URL `https://doi.org/10.1145/1708016.1708023`.

Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction, 1986.

Ambre Williams, Pierre-Évariste Dagand, and Didier Rémy. Ornaments in practice. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*, pages 15–24,

2014. doi: 10.1145/2633628.2633631. URL http://doi.acm.org/10.1145/2633628.2633631.

Andrew K Wright. Simple imperative polymorphism. *Lisp and symbolic computation*, 8(4):343–355, 1995.