



HAL
open science

Mezzo: a typed language for safe effectful concurrent programs

Jonathan Protzenko

► **To cite this version:**

Jonathan Protzenko. Mezzo: a typed language for safe effectful concurrent programs. Programming Languages [cs.PL]. Université Paris Diderot - Paris 7, 2014. English. NNT : . tel-01086106

HAL Id: tel-01086106

<https://inria.hal.science/tel-01086106v1>

Submitted on 22 Nov 2014

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.



Distributed under a Creative Commons Attribution - NonCommercial - ShareAlike 4.0 International License

THÈSE

présentée à
l'Université Paris Diderot

pour obtenir le titre de
Docteur
spécialité
Informatique

Mezzo

a typed language for safe effectful concurrent programs

soutenue par
JONATHAN PROTZENKO
le 29 septembre 2014

JURY

Président	M. Roberto Di Cosmo
Rapporteurs	M. Lars Birkedal M. Nikhil Swamy
Examineurs	M. Jean-Christophe Filliâtre M. David Pichardie M. Don Syme
Directeur	M. François Pottier

Abstract

The present dissertation argues that better programming languages can be designed *and* implemented, so as to provide greater safety and reliability for computer programs. I sustain my claims through the example of *Mezzo*, a programming language in the tradition of ML, which I co-designed and implemented. Programs written in *Mezzo* enjoy stronger properties than programs written in traditional ML languages: they are data-race free; state changes can be tracked by the type system; a central notion of ownership facilitates modular reasoning.

Mezzo is not the first attempt at designing a better programming language; hence, a first part strives to position *Mezzo* relative to other works in the literature. I present landmark results in the field, which served either as sources of inspiration or points of comparison. The subsequent part is about the *design* of the *Mezzo* language. Using a variety of examples, I illustrate the language features as well as the safety gains that one obtains by writing their programs in *Mezzo*. In a subsequent part, I *formalize* the semantics of the *Mezzo* language. *Mezzo* is not just a type system that lives on paper: the final part describes the *implementation* of a type-checker for *Mezzo*, by formalizing the algorithms that I designed and the various ways the type-checker ensures that a program is valid.

Résumé

Cette thèse décrit comment obtenir de plus fortes garanties de sûreté pour les programmes en utilisant *Mezzo*, un langage de programmation inspiré par ML, et muni d'un système de types novateur. Les programmes écrits en *Mezzo* bénéficient de plus fortes garanties, comparés à des programmes équivalents écrits dans un dialecte de ML: absence de séquencements critiques (« race conditions »), suivi des changements d'états au travers du système de types, et une notion de possession qui facilite le raisonnement modulaire et la compréhension des programmes.

Mezzo n'est pas la premier langage à s'attaquer à cet objectif louable : une première partie s'efforce donc de situer *Mezzo* dans son contexte, en présentant des travaux emblématiques de la recherche en langages de programmation, travaux qui ont constitué des sources d'inspiration ou ont servi de points de comparaison. Une seconde partie présente le langage. Tout d'abord, au travers d'une riche palette d'exemples, qui permettent d'illustrer les fonctionnalités du langage ainsi que les gains de sûreté qui en découlent. Puis, dans une partie suivante, de manière formelle, en détaillant les différentes règles qui gouvernent le système de types de *Mezzo*. *Mezzo* n'existe pas seulement sur le papier : une dernière partie décrit la manière dont le langage est implémenté, en formalisant les algorithmes utilisés dans le typeur et en détaillant les techniques utilisées pour déterminer la validité d'un programme.

Remerciements

On a parfois pu en douter, mais après quatre années d'un labeur qui fut, naturellement, fébrile et ininterrompu, cette thèse est finalement arrivée à son terme. Rédiger les remerciements au son de FIP, par une paisible après-midi du mois d'août, me permet ainsi de clore une époque heureuse à Rocquencourt. Une légère angoisse m'étreint, néanmoins, à la pensée de tous ceux qui liront ce bref texte, et qui frémiraient de n'y pas voir leur nom : je leur présente mes excuses d'avance pour ces regrettables oublis.

Mes remerciements vont en tout premier lieu à mon directeur de thèse, qui m'a accompagné tout au long de ces années. Sa disponibilité et sa patience, alliées à une connaissance encyclopédique, et une compréhension toujours fine des problèmes en jeu, m'ont permis de bénéficier d'un encadrement scientifiquement irréprochable.

Mes remerciements les plus chaleureux vont également à mes rapporteurs, Lars Birkedal et Nikhil Swamy, qui ont tous deux accepté de lire plus de deux cents pages d'un texte aride durant leurs vacances d'été. Je ne peux que m'incliner devant un tel dévouement. Je suis également heureux de compter parmi mon jury des chercheurs que je côtoie et que j'apprécie depuis plusieurs années ; c'est une réelle joie de soutenir devant un tel public.

L'équipe Gallium est également pour beaucoup dans ce dénouement heureux : les discussions autour d'un café, qu'il ait été produit par l'ancienne ou la nouvelle machine, ont toujours été hautement enrichissantes, qu'il s'agisse de systèmes de types, de catégories, de tanks, ou de voile. Un environnement d'excellence, et pas seulement scientifique, donc.

Mezzo a été l'occasion de collaborer avec d'autres membres. L'arrivée des stagiaires Henri, Armaël ou Cyprien a été particulièrement motivante pour mon propre travail ; la collaboration avec les post-doctorants Thibaut et Pierre-Évariste a été également hautement fructueuse. Quand la saturation s'est fait proche, la collaboration avec Thomas et Gabriel sur Articheck a été très agréable : mes félicitations à ces deux très bons éléments.

Au moment de franchir la porte du bâtiment 14, je me rappelle mes premiers mois où, jeune étudiant fraîchement débarqué, mes connaissances en système de types se révélèrent vacillantes. Nicolas et Benoît m'ont, à cette époque, consacré du temps et de l'énergie pour combler mes lacunes, tout comme, plus tard à d'autres occasions, Gabriel, Didier ou Xavier. Cette thèse leur doit également.

La liste des membres de Gallium que j'ai croisés est trop longue pour écrire une note personnelle à chacun : je terminerai simplement en saluant le stoïcisme de Jacques-Henri qui a bravement enduré la playlist éclectique de FIP dans le bureau.

Mon séjour à l'INRIA ne s'est, fort heureusement, pas limité au bâtiment 14 : je suis reconnaissant à tous les collègues et amis avec qui j'ai pu discuter autour d'un café. Par ordre croissant de bâtiments : Thierry, Pauline, Sylvain (bâtiment 8), toute l'équipe RAP du bâtiment 9, qui m'a toujours accueilli dans un environnement sportif, lui aussi, d'excellence : Mathieu, Emanuele, Renaud, Sarah, Henning, Cédric, et surtout Philippe ; Victorien (bâtiment 15) ; Elisa et Stéphanie (bâtiment 16) ; Stéphane (bâtiment 25). Et mes excuses à tous ceux que j'oublie.

Beaucoup me connaissent aussi comme un sportif régulier et le plus gros contributeur de messages inutiles à la mailing-list `inria-sport-team-roc` : un grand merci à tous ceux qui m'ont motivé pour devenir un nageur et un coureur. Une pensée particulière va à Philippe, qui mérite le titre de « directeur sportif » de ma thèse, dont les conseils sportifs ou humains ont été toujours précieux. J'espère me montrer digne de ses enseignements en courant, après deux semi-marathons, un marathon à Seattle. La section course à pied a été ma porte d'entrée dans le club des sportifs : merci à Alexandre et Benoît pour m'avoir poussé initialement, et à tous les autres qui sont venus courir un jour ou l'autre, la liste est trop longue pour être recopiée ici. Mention spéciale à Thierry, Pauline et Thomas, pour avoir couru un ou plusieurs semi-marathons avec moi. La section natation n'est pas en reste : Alexandre, Mathieu, Emanuele, Renaud et Cédric en ont été les dignes membres, avec, plus occasionnellement, la

participation de Thierry, Sarah, Pauline ou Henning. Un seul nom manque cruellement à l'appel, celui de Philippe Robert. J'espère qu'un thésard plus doué saura le motiver pour aller fendre les flots de la piscine du Chesnay de son crawl puissant.

Un grand plaisir au cours de ces années à Rocquencourt a été d'organiser le séminaire des doctorants, et d'aller traquer des victimes dans les divers bâtiments du centre. D'abord avec Emanuele et Mathieu, puis avec Elisa. Une pensée émue pour tous ceux qui, sous l'effet de la surprise, se sont vus répondre « oui » pour donner un exposé, et tous mes vœux de succès aux repreneurs, Pauline, Jacques-Henri et Matteo.

Sur une note plus personnelle, dans le flot de souvenirs de ces années de thèse, ressortent des soirées, des week-ends ou des après-midis, en compagnie d'amis proches. Sans eux, ces années auraient assurément été plus austères et étriquées. Je ne crois pas nécessaire de m'étendre sur le soutien et l'amitié qu'ils m'ont témoignés. Je peux, en revanche, mentionner Benoît, qui n'a jamais manqué une occasion de partager un picon-bière bien mérité après une rude journée à l'INRIA ; Alexandre, pour les bières autour de ses souvenirs en Alpha-Jet, les journées à la bibliothèque, le canapé-lit aux États-Unis « de l'Amérique » ou (bientôt) à Paris, ou encore pour avoir enduré la revue de presse du matin pendant deux années et demi ; Emanuele et Yasmine, pour l'hospitalité rue d'Aligre, les balades à Paris, le hipster-style, et pour m'avoir fait découvrir la gastronomie et la culture italiennes ; Mathieu et Jessica, qui m'ont supporté en vacances en Grèce, et dont le barbecue m'a réconcilié avec la banlieue ouest ; Nicolas, pour les discussions jusqu'à tard sur le sens de l'existence ; Sarah ou Henning, pour les pique-niques, les sorties jusqu'à trop tard ; Thomas et Pauline, pour les dîners et les bières autour d'idées de start-up ; Anne-Gaëlle et Marie pour les soirées à la colocation et les bières après ; Stéphanie, pour toutes les excursions en Normandie et ailleurs ; Ornella, Pauline, Dimitri, qui étaient loin mais toujours au bout du téléphone ; et tous ceux que j'oublie...

Stéphane et Chrysanthi reçoivent une section à part ; peut-être parce que j'ai passé plus de soirées qu'ailleurs Boulevard de Port-Royal ; peut-être encore parce que Google Now! a fini par me proposer les temps de trajet jusqu'à chez eux (l'apprentissage automatique, c'est merveilleux) ; peut-être enfin parce qu'ils m'ont supporté pendant trop de week-ends. Merci pour tout.

Je termine par ma famille, pour la présence et le soutien. Et le chat, aussi, qui remercie tous ceux et celles qui l'ont gardée pendant mes trop nombreuses absences. Meow. 🐾

Conventions

The present manuscript uses the following conventions.

Interactive sessions In some situations, I show a code sample along with the response of the associated compiler. Such passages are denoted by a keyboard symbol in the margin.

Reference to the code I will sometimes, when describing an algorithm formally, point to the corresponding location in the source code of Mezzo that implements the algorithm. The line numbers refer to the source code at the time of this writing, that is, revision 58f9a30bd.

Substitution I believe I have seen all possible variants of the substitution notation in the literature: $[x'/x]P$, $[x/x']P$, $P[x'/x]$ and $P[x/x']$. In this document, I use: $[x'/x]P$ as “substitute x' for x in P ”.

Digressions In some technical passages, I sometimes reflect on alternative design choices, or explain a point that may seem minor compared to the rest of the discussion.



This paragraph is a digression.

Pronouns *We* denotes my advisor François Pottier and I, and is used whenever talking about joint work, such as the design of Mezzo. *I* denotes the present author, and is used whenever talking about work that is in major part mine, such as the implementation. *They* is used as a gender-neutral singular pronoun. While this usage seems to be debatable [Wik14b], the Chicago Manual of Style seems to side with the present author in that it is perfectly acceptable.



Contents

I	Introduction	1
1	About type systems	3
1.1	Languages and type systems	4
1.2	What is the purpose of a type system?	5
1.3	The purpose of building a <i>better</i> type system	7
1.4	Mezzo: a better typed language for ML	8
1.5	Playing with Mezzo	11
1.6	About the proof of soundness	11
2	A brief survey of advanced type systems	13
2.1	Early, seminal works	13
2.2	Close sources of inspiration	16
2.3	Other related works	21
II	A taste of Mezzo	23
3	A Mezzo tutorial	25
3.1	Three small examples	25
3.2	Lists	31
3.3	Breaking out: arbitrary aliasing of mutable data structures	41
4	Bits from the standard library	49
4.1	Nesting	49
4.2	Adoption/abandon as a library	54
4.3	One-shot functions	56
4.4	Rich booleans	56
4.5	Locks and conditions, POSIX-style	59
4.6	Landin's knot (recursion via the mutable store)	63
4.7	Other interesting bits	64
5	Writing programs in Mezzo	65
5.1	Ownership and function signatures	65
5.2	Higher-order effects and crafting signatures	68
5.3	Reifying coercions	70
5.4	Object-oriented programming	72
III	Mezzo, formalized	77
6	A Mezzo reference	79

6.1	Differences with the informal presentation	79
6.2	Meta-variables	80
6.3	Types and kinds	80
6.4	Type definitions	81
6.5	Modes and facts	82
6.6	Programs	83
6.7	Module layer	85
7	Translating Mezzo to Simple Mezzo	87
7.1	Examples	88
7.2	Kind-checking	90
7.3	Translation	95
8	Type-checking Simple Mezzo	99
8.1	Memory model	99
8.2	Subsumption rules	101
8.3	Type-checking rules	103
8.4	The duplicable and exclusive modes	108
8.5	Facts	112
8.6	Variance	116
8.7	Signature ascription	117
8.8	Differences between Simple Mezzo and Core Mezzo	118
8.9	Reflecting on the design of Mezzo	120
IV	Implementing Mezzo	125
9	Normalizing permissions	129
9.1	Notations	129
9.2	Requirements for a good representation	129
9.3	Prefixes	130
9.4	Normal form for a permission	132
9.5	Treatment of equations	135
9.6	Data structures of the type-checker	138
9.7	A glance at the implementation	138
10	A type-checking algorithm	143
10.1	Typing rules <i>vs.</i> algorithm	145
10.2	Transformation into A-normal form	145
10.3	Helper operations and notations	146
10.4	Addition	146
10.5	Type-checking algorithm	147
10.6	About type inference	149
10.7	Non-determinism in the type-checker	150
10.8	Propagating the expected type	151
10.9	A glance at the implementation	151
11	The subtraction operation	153
11.1	Overview of subtraction	153
11.2	Subtraction examples	154
11.3	The subtraction operation	156
11.4	Implementing subtraction	163
11.5	A complete example	168
11.6	Relating subtraction to other works	169
11.7	A glance at the implementation	170

11.8	Future work	174
12	The merge operation	175
12.1	Illustrating the merge problem	175
12.2	Formalizing the merge operation	179
12.3	An algorithmic specification for merging	180
12.4	Implementation	187
12.5	Relation with other works	191
12.6	A glance at the implementation	191
12.7	Future work	193
V	Conclusion	195
13	Summary; perspectives	197
13.1	A brief history of <i>Mezzo</i>	197
13.2	Looking back on the design of <i>Mezzo</i>	198
13.3	Type system <i>vs.</i> program logic	198
13.4	Perspectives	199
	List of Figures	201
	Bibliography	205

Part I

Introduction

1	About type systems	
1.1	Languages and type systems	4
1.2	What is the purpose of a type system?	5
1.3	The purpose of building a <i>better</i> type system	7
1.4	Mezzo: a better typed language for ML	8
1.5	Playing with Mezzo	11
1.6	About the proof of soundness	11
2	A brief survey of advanced type systems	
2.1	Early, seminal works	13
2.2	Close sources of inspiration	16
2.3	Other related works	21

*Madame, je ne suis pas un grand homme, ce n'est pas la
peine de venir chercher sur moi des matériaux pour écrire
des thèses...*

Lu Xun, *La Mauvaise Herbe* (traduction Pierre Ryckmans)

The goal of this part is twofold.

The first chapter is a gentle introduction to the problem at stake. I seek to help the reader become acquainted with the landscape of programming language research: I present the challenges associated with designing a new language (there are many!) and the kind of guarantees one seeks to obtain by using a type system. The discussion is illustrated using code samples written in various languages.

The second chapter attempts to give a tour of related work, so as to better highlight the big research topics of the past few years. This chapter hopefully can be read by a student who wishes to get a (brief) overview of the narrow domain which I was preoccupied with during the past four years. This chapter is thus fairly high-level; later chapters feature, whenever applicable, specific comparisons with similar works in the literature.

1. About type systems

Ever since the field of computer science was pioneered, people have been trying to interact with machines. The internet folklore is rich with stories about how real programmers would directly poke into the front panels of computers and fix them over the phone [Pos83].

“ Jim was able to repair the damage over the phone, getting the user to toggle in disk I/O instructions at the front panel, repairing system tables in hex, reading register contents back over the phone. The moral of this story: while a Real Programmer usually includes a keypunch and line printer in his toolkit, he can get along with just a front panel and a telephone in emergencies.

Figure 1.1: A story about a real programmer

But for all the manliness that comes out of being a “real programmer”, most people agree nowadays that using a programming language is certainly a much more convenient way to talk to a computer, rather than punch cards or front panels.

Indeed, just like people (supposedly) talk to each other in English, French or some other natural language, programmers talk to the computer using a programming language. And, just like natural languages, the first computer languages were rather primitive. Figure 1.2 shows a sample program written for the lunar landing module of the Apollo 11 mission; it is a wonder people actually landed on the moon.

```
TC      BANKCALL      # TEMPORARY, I HOPE HOPE HOPE
CADR    STOPRATE      # TEMPORARY, I HOPE HOPE HOPE
TC      DOWNFLAG      # PERMIT X-AXIS OVERRIDE
ADRES   XOVINFLG
TC      DOWNFLAG
ADRES   REDFLAG
TCF     VERTGUID

STARTP67 TC      NEWMODEX      # NO HARM IN "STARTING" P67 OVER AND OVER
DEC     67             # SO NO NEED FOR A FASTCHNG AND NO NEED
CAF     ZERO          # TO SEE IF ALREADY IN P67.
TS      RODCOUNT
CAF     TEN
TCF     VRTSTART
```

Figure 1.2: Assembly listing for the Apollo Guidance Computer – lunar landing module

Scientists thus strived to design better programming languages. The flurry of programming languages that have been invented since the dawn of computing is a testimony to the countless hours programming language

(PL) researchers have devoted to this noble endeavour. Wikipedia [Wik14a] lists more than six hundred “non-esoteric” programming languages.

Some even seemed to think that finding the “right” programming language was within reach.



At the present time I think we are on the verge of discovering at last what programming languages should really be like. I look forward to seeing many responsible experiments with language design during the next few years; and my dream is that by 1984 we will see a consensus developing for a really good programming language (or, more likely, a coherent family of languages). (...) At present we are far from that goal, yet there are indications that such a language is very slowly taking shape.

Figure 1.3: Donald Knuth on programming languages [Knu74]

History, quite unfortunately, tends to prove the author of the quote above wrong. Not only have many new languages been created since 1974, but also present languages are far from satisfactory. The quest for a better programming language thus goes on.

1.1 Languages and type systems

Early on, programming languages such as Algol 60 and Fortran were equipped with basic type systems. Algol, for instance, would classify program values according to their types. There were only a few types, such as REAL, INTEGER, BOOL and ARRAY.

Consider a program which tries to access an array. If the program is given, say, an integer instead of an array, the program is going to read memory it was not intended to read. This can cause the program to crash or, worse, to run into completely random, erratic behavior, with worrying consequences (Figure 1.4).



An unreliable programming language generating unreliable programs constitutes a far greater risk to our environment and to our society than unsafe cars, toxic pesticides, or accidents at nuclear power stations. Be vigilant to reduce that risk, not to increase it.

Figure 1.4: Tony Hoare on programming languages [Hoa81]

The point of having a type system is to eliminate entirely this family of errors by ensuring, say, that programs which expect arrays only ever receive arrays as inputs.

Nowadays, the guarantees offered by type systems vary, but the original idea remains the same: rule out *some* run-time errors by making sure the program manipulates values with the right *types*, thus guaranteeing some degree of *safety*.

In the case of Algol and Fortran, type-checking would be performed by the compiler, so as to rule out errors *in advance*. This is not always the case. Indeed, a common way to classify modern type systems is to separate them into dynamic type systems and static ones.

Dynamic type systems check *on the fly* that values have the correct type. Here is an ill-typed fragment of Javascript, a rather popular programming language. We try to perform a function call; the expression we call is not a function but a string, meaning that the call cannot take place. In PL lingo, the program cannot reduce and execution is stuck.



```
"lambda"();
```

```
TypeError: "lambda" is not a function
```

The error will not pop up until we *actually* try to run the program. That is, the programmer is free to write this script and load it into a web page. The error will not appear until execution reaches this piece of code.

Static type systems, conversely, attempt to detect errors *before* the program is actually run. Here is the same example, this time fed into an OCaml compiler.

```
"lambda"();;
```

```
Error: This expression has type string
      This is not a function; it cannot be applied.
```

The compiler will not even attempt to run the program above; the type system detected *in advance* that this program would cause an error at execution and consequently refuses to even compile it.

Both typing disciplines have their respective merits, and people have argued endlessly about which of the two is the better one, using constructive arguments such as “I need a PhD in type systems to understand an OCaml error message, types kill my productivity” or “How can these people program without a typing discipline, they’re wasting their time on trivial mistakes they could catch easily, they are *morally* wrong”.

Amusingly, this dispute is not new and already made headlines thirty years ago. (Figure 1.5).

“The division of programming languages into two species, typed and untyped, has engendered a long and rather acrimonious debate. One side claims that untyped languages preclude compile-time error checking and are succinct to the point of unintelligibility, while the other side claims that typed languages preclude a variety of powerful programming techniques and are verbose to the point of unintelligibility.

Figure 1.5: John Reynolds on typed *vs.* untyped languages [Rey85]

It turns out these two options are not mutually exclusive: mainstream languages such as C++ and Java, while being statically typed, offer mechanisms (`dynamic_cast` and `instanceof` respectively) for dynamically examining the type of an object.

In spite of having happily written thousands of lines of Javascript, I remain at the time of this writing a member of Gallium, a team where OCaml originated! I will therefore from now on focus on mostly-static type systems, and argue that the quest for a safer programming language mandates the use of a better static type system.

1.2 What is the purpose of a type system?

As stated earlier, the purpose of a type system is to rule out *certain* errors. Here is a series of examples, written in various languages, that illustrate the kind of guarantees that typical, real-world type systems offer (or fail to offer).

Rejecting programs

As our first example, let us consider the following Java program, which is accepted by any conforming Java compiler.


```
public class Test {
    static void printHello(String what) {
        System.out.println("Hello, "+what+"!");
    }
    public static void main(String[] args) {
        printHello("thesis");
    }
}
```



```
jonathan@ramona:/tmp $ javac Test.java && java Test
Hello, thesis!
```

The program is valid: it does not run into a memory error at execution-time. This property is guaranteed by the type system: this is known as *type soundness*.

Here is a modified version that is rejected by the type system.




```
public class Test {
    static void printHello(String what) {
        System.out.println("Hello, "+what+"!");
    }
    public static void main(String[] args) {
        printHello(2014);
    }
}
```

```
Test.java:7: error: method printHello in class Test cannot be
applied to given types;
    printHello(2);
    ^
    required: String
    found: int
```

The type system is right in rejecting this example: we do not know what the behavior of the resulting program at execution-time would be. The type system of Java provides *type safety*: well-typed Java programs do not run into memory errors at execution.

Some more advanced type systems provide more guarantees than memory safety. Some research languages such as F*, Coq or Agda [SCF⁺11, Theo06, Noro09] are equipped with type systems that can prove *functional correctness*, that is, that the program does what it is intended to do.

Conversely, some weaker type systems do not guarantee memory safety. Here is the example of a C++ program that, although accepted by clang++, runs into a memory error at run-time.



```
int main() {
    int* i = new int;
    delete i; delete i;
}
```

```
protzenk@sauternes:/tmp $ clang++ test2.c && ./a.out
*** Error in './a.out': double free or corruption (fasttop):
0x00000000183a010 ***
```

Here, the type system was not smart enough to figure out something was wrong with this program. Conceptually speaking, the variable *i* went from “valid pointer to an integer” to “invalid pointer to an integer”. Yet, in the eyes of the type-checker, the variable *i* keeps the *int** type throughout the whole code sample, which is the only requirement for the calls to *delete* to be legitimate.

Incidentally, we are lucky the program crashed: it may be the case that the program returns an incorrect result without notifying the user an error happened. We have no guarantees about a buggy program; anything may happen.

This example illustrates a first important question about any type system: what are the kind of guarantees that the type system offers? Worded differently, *what is the class of errors that a type system promises to rule out statically?*

Accepting programs

A way to look at a type system is whether it *rejects* or not an incorrect program. Another way to look at it is to see whether it *accepts* or not a correct program. I have not defined what it means for a program to be “correct” or

“incorrect” yet; let us just assume that a program that terminates by producing the intended result for all inputs is correct, and that any other program is incorrect.

As it turns out, type systems are inherently imperfect. Just like they may accept buggy programs, they may fail to accept correct programs. Here is an example, written in Haskell. The `x` variable is defined to be either a string or an integer; the program then prints `x`. The program may run into an error if `x` happens to be an integer and we try to print it as a string; the condition, however, is always true, meaning that `x` is always a string, and that the program never runs into an error at execution. This program, quite unfortunately, is rejected by the type system: we say that the type system is *incomplete*.

```
Prelude> let x = if True then "hello" else 4 in putStrLn x
```

```
<interactive>:7:31:
  No instance for (Num [Char]) arising from the literal '4'
  Possible fix: add an instance declaration for (Num [Char])
```

A type system would have to perform remarkably sophisticated reasoning to figure out that such a program will not run into an error. Indeed, the boolean expression can be arbitrarily complex; whether this program runs into a run-time error may not even be *decidable*!

This highlights another important feature of type systems: they are fundamentally limited, in that they will fail to accept correct programs. Indeed, the more programs one wishes to accept, the more complex the resulting type system ends up.

This is an important issue when designing a new type system: one is always faced with a compromise between:

- the class of errors one wishes to *rule out*,
- the class of programs one wishes to *accept*,
- the complexity of the underlying theory.

One may find the last point remarkably restrictive. However, trying to keep the type system simple is not necessarily a bad thing. First, type systems have been successful partly because they enjoy a number of properties, such as being decidable. A more sophisticated type system risks losing these properties. Second, accepting more exotic programs is not necessarily a good thing either; the Haskell program above, for instance, makes little sense. Rejecting a program and telling the user to better structure their code is a good property of a type-checker.

1.3 The purpose of building a *better* type system

We have seen a few type systems in action; each one of them provides a different set of guarantees (rules out certain classes of errors), and has its own expressive power (accepts certain classes of programs). If one takes an existing type system, and pushes it further, what would this “better” type system look like?

The present dissertation is about the design and implementation of Mezzo, a new programming language that takes the type system of ML further. Figure 1.6 illustrates what one can expect of Mezzo compared to ML. Each potato denotes the set of programs accepted by the type system.

Incorrect programs The type system of Mezzo provides strictly stronger guarantees than that of ML: any wrong program rejected by ML will also be rejected by Mezzo (●). Mezzo will also reject programs that ML accepts: the new type system rules out more incorrect programs (●). Mezzo, however, is not complete, meaning some incorrect programs are still accepted (●).

There is one color missing out of the 8 possible ones: no incorrect program that was previously rejected by ML becomes accepted by Mezzo.

Correct programs The sets of programs accepted by Mezzo and ML are incomparable.

Some correct programs that were *previously rejected* will become *accepted* (●): the new type system is more sophisticated and can therefore “understand” programs that were rejected before.

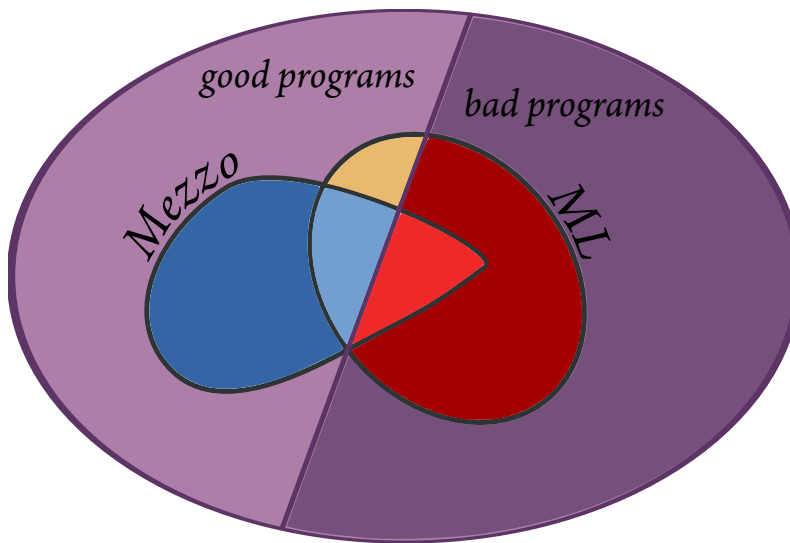


Figure 1.6: Mezzo vs. ML: potato, potato

Some correct programs that were *previously accepted* will become *rejected* ●: the new type system imposes a stronger discipline. Programs that previously relied on implicit invariants will now see the type-checker trying to verify these implicit invariants. In some cases, the invariants will be too complex for the type-checker to understand, meaning that the program will have to be rewritten in a different style.

Some correct programs that were *previously accepted* will remain *accepted* ●: the type system will still cover previously-working programs.

Some correct programs that were *previously rejected* will remain *rejected* ●: the type system will still fail to account for correct programs.

1.4 Mezzo: a better typed language for ML

The contributions of Mezzo are along two main axes. First, Mezzo offers a sophisticated type system that rules out more programming mistakes, hence providing greater confidence in programs written using the language. Second, Mezzo also offers the programmer a carefully crafted *surface language*, which provides natural, familiar constructs while hiding the complexity of the underlying theory.

The run-time model of Mezzo is the same as OCaml; therefore, this dissertation focuses solely on the design and implementation of the language and the corresponding type-system. A regrettable restriction is that the OCaml run-time system is (currently) not concurrent; any concurrent Mezzo program will thus be executed, sadly, in a sequential manner.

Mezzo ships with an interpreter and a compiler. The compiler works by translating a Mezzo program into an equivalent OCaml program; the resulting OCaml program can then be compiled by the regular OCaml compiler.

Mezzo takes its inspiration from ML: the language has first-class functions, data types, pattern-matching, polymorphism, type inference. The type system of Mezzo, however, goes further than that of a regular variant of ML, as it features more sophisticated reasoning. Therefore (§1.3), some programs that were previously rejected will now be accepted (the type system is smarter) and some programs that were previously accepted will be rejected (they will be deemed *unsafe* per the standards of the new language).

The distinctive feature of Mezzo is that it fundamentally reasons about *ownership* of data structures. This in turn allows the user to reason about *aliasing* as well as *state*. Mezzo is made up of two layers. The core, static layer combines several type-theoretic mechanisms to create an expressive, albeit somewhat restrictive static discipline. To evade the restrictions of the static discipline, Mezzo complements the static layer with a dynamic layer that uses run-time tests to regain static type information.

Mezzo has been mechanically proved sound [BPP14a]. Mezzo enjoys the standard soundness result, which is that “well-typed programs do not go wrong”. Mezzo, however, also enjoys a stronger result, which is that well-

typed programs have no data races.

Modular reasoning

As I argued earlier, gaining confidence in the programs one writes is fundamental. One achieves greater confidence by *reasoning* about whether the program achieves the intended purpose.

Reasoning about programs with mutable state is hard, yet important: a purely functional approach is not always practical. (Even using *monads*, one can only separate effectful code from non-effectful code, thus offering no way to reason in a fine-grained manner about mutations.) Indeed, in the general case, whenever one inspects a piece of code that performs mutations, one needs to think about the *effects* the snippet may perform: is this piece of code going to increment a global counter? Is this snippet going to modify the mutable objects I pass it? If I send a reference to a mutable object on another thread, am I allowed to further modify the object?

Having untracked effects prevents *modular* reasoning: one cannot reason about a piece of code in isolation, and one has to think about whatever effects the rest of the program may perform. The programmer thus has to reason about the entire program, which is often beyond the cognitive power of normal humans.

Sadly, mainstream programming languages do not provide facilities for reasoning about programs. Right now, the programmer is left with documentation. Documentation is usually text, which states (imprecisely) what guarantees the function *expects* (the function’s pre-condition), as well as what guarantees the function *provides* (the function’s post-condition). This text is not machine-checked, meaning that one relies on the good will and skill of the programmer to enforce proper usage of the code.

Documentation can also take the form of conventions: the Google C++ style guide [goo], for instance, mandates that reference arguments be labeled with `const`, meaning that a function that takes a `const T&` cannot modify it, while a function that takes a `T*` will, by convention, modify it. These code conventions, when combined with extra “lint” tools, provide a limited set of guarantees.

I argue that these checks should be performed by the compiler; worded differently, the reasoning that yields stronger confidence in programs should be part of the core language, and not be a feature provided by external tools. The purpose of this thesis is thus to present the design of a language where modular reasoning about effects is made possible, and incorporated in the language from the ground up.

Towards a notion of ownership

If we want to reason in a modular fashion about effects, we need to describe in a rigorous manner the pre- and post-conditions of functions. In the presence of global state, we will thus mention whatever effects the function may perform: a function which increments a global counter will now advertise that it may modify this global, shared object.

I argue that ownership is an intuitive and natural notion that allows one to reason efficiently about their programs. In *Mezzo*, at any program point, there is a notion of what “we” own and what “others” may own. Here, “we” should be understood as “the currently executing piece of code”, while “others” may refer to other running threads, or the calling context. Ownership may be understood in various ways; in *Mezzo*, whatever is mutable has a unique owner. This enables reasoning such as “I own (uniquely) this mutable reference, therefore others cannot modify it”. If a function demands ownership of the reference, I know that it may modify it. Conversely, if a function does not demand ownership of the reference, I can assume that the reference will remain unchanged throughout the function call.

Ruling out mistakes

Reasoning about ownership yields significant benefits. A major one is in concurrent settings, where one needs to make sure that concurrent accesses to a shared mutable piece of data are protected. By incorporating reasoning about ownership directly within the type-system, we guarantee that *Mezzo* programs are data-race free.

Another class of bugs that *Mezzo* rules out entirely is representation exposure. Consider the code Figure 1.7. The code implements a `Set` interface. The purpose of the interface is to prevent the client from breaking the internal invariants of `Set`, which are as follows.

- A `Set` has a fixed maximum size `maxSize` initially set when the object is created.
- There are, at any time, `curSize` elements in the `Set`; besides, `curSize` is at most `maxSize`.

```
import java.util.*;

public class Set<T> {
    final int maxSize;
    int curSize;
    LinkedList<T> elements;

    Set(int maxSize) {
        this.maxSize = maxSize;
        this.curSize = 0;
        elements = new LinkedList<T>();
    }

    boolean add(T elt) {
        if (curSize == maxSize)
            return false;

        elements.push(elt);
        curSize++;
        return true;
    }

    List<T> asList() {
        return elements;
    }
}
```

Figure 1.7: A buggy Java program

- The Set is implemented using a list of elements; at any time, the length of elements is equal to curSize.

Quite unfortunately, the under-caffeinated developer who wrote this code implemented an `asList` method. The Set class happens, internally, to also use a list for representing the set of elements. This means that by returning a pointer to the internal list, the `asList` function allows its caller to gain access to the *same* list which is used internally. The caller may thus modify it and break the invariants. By stating that a piece of mutable data has a unique owner, Mezzo prevents this sort of bugs: the list is owned either by the Set instance, by the caller of the `asList` method, but not by both at the same time. An ownership transfer takes place when entering and returning from the function call: invariants are checked across function calls.

Enabling new programming patterns

By ensuring that pieces of mutable data have a unique owner, Mezzo can track mutations in a fine-grained manner. Type-changing updates, for instance, are possible in Mezzo. I may freely change the type of a reference; the type-checker is happy with it, since both the compiler and I know that other parts of the program are not aware of the existence of the reference, for I am its unique owner.

The programmer is thus free to perform more powerful mutations than what is traditionally possible in ML. This enables new programming patterns, such as progressive initialization, or memory re-use.

As a type system

We chose to seek greater confidence through the use of a type system. Several approaches have been explored. Some works, for instance, embed already type-checked, existing programs in a program logic. We made a design decision, and chose not to aim for full program correctness, but rather a stronger notion of correctness through the use of a type system.

In our eyes, the advantages are many. Type-checking is pervasive throughout all stages of the development cycle; it helps the programmer shape their code rather than retrofit a program proof on top of already-written code; it helps the compiler generate better code; we envisioned, for the long term, a two-stage process where users would type-check their code in a powerful type system, thus yielding enough knowledge and structure to perform a subsequent proof of the program more easily.

1.5 Playing with Mezzo

Online

The homepage of Mezzo is at <http://protz.github.io/mezzo>. The curious reader may want to play with the web-based version of Mezzo at <http://gallium.inria.fr/~protzenk/mezzo-web/>. The sources for Mezzo are available at <https://github.com/protz/mezzo>; the web frontend is in the `web/` subdirectory and can be setup locally.

On your computer

Mezzo can also be used in a more traditional way. Using OPAM [opa], one merely needs to type `opam install mezzo`. This installs a Mezzo binary in the path called `mezzo`. By default, calling `mezzo foo.mz` will type-check the file `foo.mz`. Adding a `-i` argument will interpret the file. The adventurous reader who wishes to compile Mezzo code can use the sample project at <https://github.com/protz/mezzo-sample-project/>.

1.6 About the proof of soundness

The present dissertation does not cover the proof of soundness for Mezzo, which was carried out by F. Pottier *et al.* It does seem tempting to include the formalization of Mezzo as an appendix, so as to make this document a complete Mezzo reference. The metatheory, however, is complex, and requires quite a bit of an explanation (about 30 pages of a fairly dense journal paper). It thus seems hard to include the formalization “as is”. The interested reader will most certainly want to read the submitted journal paper [BPP14a]; the first half of the paper contains material that has been updated and included in the present dissertation (Chapter 3, Chapter 7).

2. A brief survey of advanced type systems

I mentioned earlier (§1.4) that Mezzo allows one to reason about state, aliasing and ownership. Many languages have been designed over the past decades, aiming at a better static control of mutable state. These keywords thus surely ring a bell for any reader familiar with the field.

The present chapter attempts to do a quick panorama of the field. I highlight early, seminal works that had a significant influence on later works (§2.1), and that articulated key concepts. Then, I detail a few sources of inspiration for Mezzo (§2.2). These comparisons feature forward pointers to specific parts of the dissertation, should the reader wish to come back to this chapter later. Finally, I also mention similar efforts in other domains, that however did not constitute sources of inspiration for Mezzo (§2.3). These efforts may relate to another programming paradigm, or may be too low-level for us to take inspiration from; they help, nonetheless, locate Mezzo within the larger field of partial static verification of programs.

This chapter remains very general; later chapters (Chapter 8, Chapter 11, Chapter 12), when tackling specific issues, provide their own sections (§8.9, §11.6, §12.5) devoted to the comparison and the articulation of links with other works.

A history of linear type systems, regions and capabilities is available online [Poto7], which features many more works, although not necessarily related to Mezzo. The historical timeline has been reproduced in Figure 2.1.

2.1 Early, seminal works

Effect systems

In the late 80s, Gifford and Jouvelot [GJSO92, GJLS87] worked on FX, a language where *effects* are tracked within the type system. The system has inference and polymorphism for both types and effects [JG91]. Effects are coarse: a function, for instance, may be annotated with `pure` if it performs no side-effects, or with the `read` or `write` keyword.

The system provides type-checking rules built on an algebra of effects. A typing judgement, in their system, takes the form of $E : T ! F$, meaning that type-checking E yields type T with effect F . The typing rule for application, shown below, states that the effect of evaluating a function call is the greatest effect of evaluating the function and of evaluating the argument.

$$\frac{A \vdash E_0 : (\text{proc } F T_0 T) ! F_0 \quad A \vdash E_1 : T_0 ! F_1}{A \vdash (E_0 E_1) : T ! \max(F_0, F_1)}$$

This early strand of work seems to introduce first the key idea that *effects can be tracked within the type system*. This idea influenced a whole series of subsequent languages.

While FX does indeed provide stronger static properties and facilitates modular reasoning, the authors mostly sell this new type system as a way for the compiler to perform automatic optimizations, such as parallelism or mem-

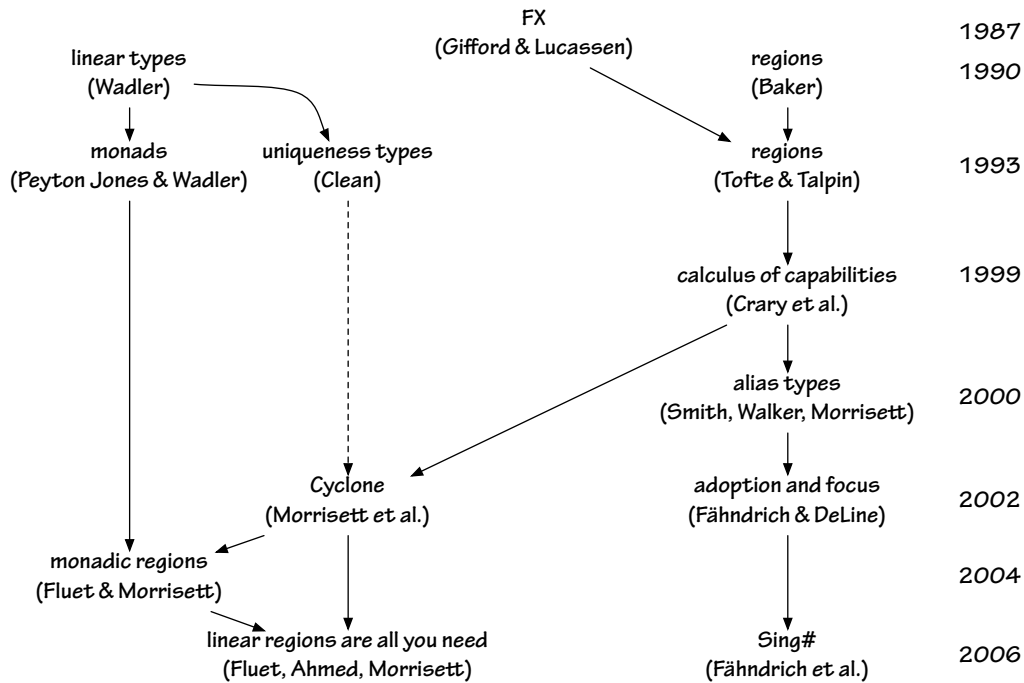


Figure 2.1: A history of linear type systems, regions and capabilities (F. Pottier)

oization. (As a historical side note, they also present FX as a way to solve the interaction between polymorphism and side-effects – the value restriction had not been discovered back then.)

Linear types

Philip Wadler, around the same time, notes that the concepts of linear logic [Gir87], where a hypothesis may only be used once, can naturally be extended to a programming language, where some types may be made *linear* [Wad90]. Variables with a linear type must be used exactly once. This introduces the idea that the objects that one manipulates may be considered as *resources*: possessing a file handle means, for instance, that there is an underlying system resource that exists once and that needs to be destroyed exactly once.

The main selling point that Wadler brings forward is efficiency: having linear variables leads to a more efficient implementation strategy. This may sound surprising, as one may expect nowadays to reason about linearity for safety reasons. The paper, however, is from the early 90's and the context is that of purely functional languages, which suffered from performance issues.

Linear values cannot be duplicated, nor discarded. The former implies that some operations, such as writing into an array, can now be implemented destructively instead of performing copies. Indeed, as long as the array is linear, there is only one reference to it at any program point. Hence, it is safe to modify the same array all throughout the program rather than allocate copies of it. The latter point implies that the destruction of linear objects is explicit (e.g. by calling a special *destroy* function), which leads to an efficient implementation technique: allocations and deallocations of linear objects do not require the use of a garbage-collector since they are known in advance.

Wadler accurately notes that a system where everything is linear would be too restrictive. He thus divides his types in two categories: linear types, and nonlinear types. Data types come in two flavors: λK (linear) and K (nonlinear). A lambda-abstraction may be annotated with λ to create a linear arrow. Nonlinear closures cannot capture linear variables.

The typing rules of his system reflect this fact (Figure 2.2). For instance, a variable may only be typed in the singleton context, meaning that it has to be used exactly once. Naturally, the rule for application splits the environment to make sure we don't use a variable twice.

The way these restrictions are lifted for nonlinear variables is by means of two rules, called LLC-KILL and LLC-COPY.

$$\begin{array}{c}
 \text{LLC-VAR} \\
 \frac{}{x : t \vdash x : t}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{LLC-APP} \\
 \frac{\Gamma_1 \vdash M : t \multimap u \quad \Gamma_2 \vdash N : t}{\Gamma_1, \Gamma_2 \vdash MN : u}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{LLC-KILLIFDUP} \\
 \frac{\Gamma \vdash M : u}{\Gamma, x : t \vdash M : u}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{LLC-COPYIFDUP} \\
 \frac{\Gamma, x : t, x : t \vdash M : u}{\Gamma, x : t \vdash M : u}
 \end{array}$$

Figure 2.2: Sample typing rules for linear lambda calculus

Interestingly enough, Wadler notes that the user may want to temporarily “freeze” a mutable variable into a read-only version. This allows one to temporarily duplicate, say, an array: indeed, it is safe for multiple people to just *read* at the same time. This is done using a syntactic scoping criterion and a special `let!` $(x) y = e1$ in $e2$ construct, where x is temporarily made nonlinear in $e1$ via the alias y . The fact that y is scoped only in $e1$ means that no nonlinear alias to x can escape. This construct would fail to work if there were any sort of global mutable state, such as ML references, since one could naturally write y into the global state.

Interestingly, the construct above forces an evaluation order, which deserves to be noted since we are in the context of a lazy, purely functional language. Indeed, the parts that access the array as read-only (that is, $e1$) have to be evaluated *before* it may be mutated (in $e2$).

Wadler also notes that this would probably play badly with a type inference algorithm in the style of Damas-Milner.

Temporarily relaxing aliasing restrictions has been a pervasive problem in the literature back from the early 90's until today. Each of the various strands of works features its own mechanism for doing so. While Wadler uses a syntactic criterion (just like Cyclone, which I mention in §2.3), Crary *et al.* [CWM99], for instance, use *bounded quantification*, that is, a type-theoretic mechanism. In their calculus of capabilities, $\{\rho^1\} \leq \{\rho^+\}$, meaning that the capability for owning the region ρ *uniquely* (which allows destroying) can be weakened into a weaker capability for just *accessing* the region. By writing a function with bounded quantification $\forall(\rho \leq \rho_1^+)$, the body of the function may freely alias the capability since it is not assumed to be unique. The caller, however, may own a unique capability that, by virtue of subtyping, can also serve as a capability for calling the function. The caller, by unification, regains its unique capability after calling the function. The mechanism is sound because capabilities cannot be stored. The design choice made in Mezzo was to use runtime tests to relax aliasing restrictions.

Regions

A last strand of work that turned out to be remarkably influential is *regions*. Baker [Bak90] notes in 1990 that the US Department of Defense declined to add a Garbage Collector (GC) to its official language Ada83, because of performance concerns. He thus tackles the issues of static analyses that could possibly help handle the allocation and deallocation of objects better.

Noting the link between instantiating a polymorphic type scheme and inlining the said function, Baker pushes the correspondence further and notes that variables which have been assigned different types are necessarily distinct (I should probably add that this fact is not true in the presence of type abstraction). The static typing information can thus provide information about aliasing at run-time.

A key example in the paper is that of the `append` function, which usually has type `list 'a -> list 'a -> list 'a`. Baker then adds an extra type variable to the `list` type, which acts as an “instance identifier” for the list. This “instance identifier” is not a memory location: all cells from the same list have the same identifier and cells from distinct lists have distinct identifiers. Type-checking the body of `append` again, with the usual unification algorithm, we get a new type `list ('l1, 'a) -> list ('l2, 'a) -> list ('l2, 'a)`. This gives what we would now call *alias information*: the return list *cannot* contain elements from the first argument. Since these are type variables, in the case the two arguments belong to the same list, the type variables `'l1` and `'l2` get unified at call-site.

The main application in the eyes of Baker is memory collection performance: when returning from a function, all the intermediate values that live in instances distinct from the return value can be safely discarded; when updating an array that is distinct from all others, it may be updated in place (this was also a concern of Wadler).

The paper is interesting in several regards. First, it introduces the idea that the type system can track *aliasing*; also, it introduces the idea that some memory nodes may live in the same conceptual region of the heap (e.g. be in the same list) and alias each other unbeknownst to the type-checker.

In 1994, Tofte and Talpin [TT94] push the idea further by incorporating the idea of regions in a dialect of ML. They introduce two key mechanisms: the `letregion ρ in e_1` construct, which introduces a well-scoped *lexical region*, and the `e_1 at ρ` annotation, which instructs in which region a *value* should live. These two constructs are *inserted automatically* by their inference procedure.

Function types are annotated with read and write effects (which they call *get* and *put*) over the various regions. An effect system similar to FX allows to unify conflicting effects when unifying function types. The soundness result depends on being able to track properly the effects performed by a function: indeed, a closure may not only perform effects over the regions of its arguments, but may also capture regions!

Their work addresses two concerns with Baker’s paper. First, they prove safety. Second, they deal with polymorphism by making sure that region variables correspond to Λ -abstractions which can be instantiated just like in ML. In particular, they use *polymorphic recursion* to make sure that within a recursive function, recursive calls can instantiate onto different regions, hence providing good results.

Indeed, their concern remains the same as Baker’s: provide efficient memory management by using region annotations to figure out that a whole set of objects went out of scope, hence becoming candidates for deallocation. This approach has been leveraged by MLKit [BRIT93].

2.2 Close sources of inspiration

Alias types

The authors of “Alias Types” [SWMoo] set themselves in the context of a low-level language, possibly a typed assembly language. Things are quite different: memory locations (or registers) are mutated in place, so as to reuse space. A single location will thus have several distinct types throughout its lifetime. Besides, compilation generates code that uses patterns such as progressive initialization, where the fields of an object are initialized one after another. The object hence goes through a variety of states (types) before it is ready.

Having type-changing updates mandates the use of linear types; otherwise, the system would be unsound. This is at odds with another constraint that the authors have, namely that they *need* to alias objects: a pointer may be copied into a register, thus creating another alias to the object originally pointed to.

The authors introduce a new type called $\text{ptr}(\ell)$, which is that of a pointer to memory location ℓ . They represent the mutable store through constraints of the form $\ell \mapsto \langle \dots \rangle$, where \mapsto stands for “points to” and $\langle \dots \rangle$ stands for a memory block. A store is made up of several constraints separated by \oplus .

Here is a fragment of code that describes a memory block at ℓ_1 of size two, whose two components point to the same memory location ℓ_2 , where an integer is located.

$$\ell_1 \mapsto \langle \text{ptr}(\ell_2), \text{ptr}(\ell_2) \rangle \oplus \ell_2 \mapsto \langle \text{int} \rangle$$

In a fashion similar to Wadler, the authors provide two flavors of constraints: linear and nonlinear.

The authors also add two important mechanisms: location polymorphism, in order to write generic functions that can operate on any memory location, and store polymorphism, so as to make sure that a function can touch a portion of the store and leave the rest untouched.

The key insight of the paper is that the $\text{ptr}(\ell)$ type is duplicable; that is, it conveys no ownership information. The ownership information is stored in the $\ell \mapsto \dots$ constraint. One is thus free to alias an object; the type system merely records that there are pointers to this object, and keeps the ownership information in one central place. This makes *strong updates* (type-changing updates) possible, since there only exists one copy of the “real type” of an object.

In an extension of their original paper [WMoo], the authors introduce extra type-theoretic mechanisms to take into account recursive data structures such as trees and lists. They add tagged unions (using integer singleton types), recursive types (using a μ combinator), existential packing, and more importantly, what they call *encapsulation*. Encapsulation allows one to package a pointer (the $\text{ptr}(\ell)$ type we saw earlier) along with a constraint

(of the form $\ell \mapsto t$), meaning that a value now embeds ownership of another fragment of the heap. They write $(\text{ptr}(\ell) \mid \{\ell \mapsto t\})$. These ingredients are enough to define the type of lists:

$$\mu\text{list}. \langle \hat{0} \rangle \cup \exists(\ell : \text{loc} \mid \{\ell \mapsto \text{list}\}). \langle \hat{1}, \text{ptr}(\ell) \rangle$$

The definition above describes a type that is either the constant 0 (this would be the Nil case in an ML-like setting), or a block tagged with the constant 1 (this would be Cons). The latter case is wrapped in an existential quantifier that refers to a *certain location ℓ along with* a linear constraint for that *same location*. The linear constraint merely says that the tail of a cons cell is another list.

What this means, in essence, is that once a run-time test is performed, the type-checker can discriminate the union and, if we are in the Cons case, unpack the existential and obtain a linear constraint for a Cons cell whose tail is itself a list.

Naturally, the devil’s in the details; I leave out of the picture complex aspects of this work, such as subtyping. Nonetheless, these simple ingredients are remarkably powerful. The authors manage to express patterns such as destination-passing style, or in-place tree traversal using link reversal.

Many of these ideas are to be found (although in a somewhat different form) in Mezzo; it is not a coincidence that two of the first few examples that we wrote were a destination-passing style `List.map` and an in-place tree traversal. The description of the heap in terms of locations and points-to constraints is very similar to the “expanded”, structural permission conjunctions we describe in §3.2: the $\text{ptr}(\ell)$ is analogous to our singleton types and $\ell \mapsto \dots$ is reminiscent of $\ell @ \dots$. The encoding of data types remains, however, slightly low-level and is closer to what is modeled in the proof of soundness of Mezzo. The type-checking of Mezzo, however, remains slightly higher-level and uses data types as a primitive construct (§8.3).

Alias Types had a somewhat lasting influence. Their novel mechanism for handling the ever-pervasive problem of aliasing was re-used for Typed Assembly Language [MWCG99], the flow-sensitive analysis of CQUAL [AFKT03], as well as Vault (§2.3).

L³: a Linear Language with Locations

The authors of L³ [AFMo7] try to re-interpret the alias types language within the formalism of Linear Logic.

Interestingly, they note that a linear reference (in the style of Wadler) is simply the combination of a pointer to a memory location along with a capacity for that location, thus tying Alias Types and Linear Types together.

$$\text{lref } \tau \equiv \exists \ell (\text{ptr}(\ell) \mid \{\ell \mapsto \tau\})$$

They build a core language where everything is linear and every well-typed program terminates, then extend it with duplicable (also known as “nonlinear”, “unrestricted”, “frozen”, or “type-invariant”) references. Typically, one wishes to *freeze* a linear reference into a non-linear one which no longer supports strong updates.

Rather than implement a (necessarily restrictive) swap operation for unrestricted references that ensures that the type remains the same through the update, they offer a construct that thaws a frozen reference into a linear one. In order to make this sound, the user has to justify that no other thawed reference exists; they use a run-time, linear token that is threaded through the entire program to keep track of which references have been thawed, along with their original type. This permits a re-freeze operation to take place as well.

The authors do not commit to a specific implementation of this linear token; rather, they leave it up to the actual implementation to choose a particular mechanism for proving at run-time that the value currently has no other alias. Some mechanisms for doing actually already exist: the *locks* mechanism is one such example.

Threading a run-time token to ensure unique ownership using run-time tests is an idea that Mezzo reuses, through the adoption/abandon mechanism §3.3.

Separation logic

I now leave the strand of work covered by Figure 2.1 and turn to separation logic, which is perhaps the biggest source of inspiration for Mezzo. Indeed, I often sum up Mezzo as “separation logic turned into a type system”.

Separation logic is not a type system but a program logic, that is, it allows one to reason on programs. Essentially, it is a remarkably convenient framework for manipulating *symbolic heap predicates*, thus performing symbolic interpretation of (possibly concurrent) programs. Separation logic forms the basis for program verification tools such as Ynot [CMM⁺09].

The original paper by John Reynolds [Rey02] appeared in 2002. Since then, more than a hundred papers have been written, all of which use a slightly different presentation of separation logic. Several common traits emerge [Jen13].

Symbolic heap predicates are written using an *assertion language*. The language usually contains memory blocks, points-to assertions, and features the separating $*$ connective. The $*$ connective is key: it denotes two physically disjoint heap fragments. We see below why this is crucial.

Here is an example of an assertion, where $\text{tree}(p)$ is a built-in predicate that describes a heap fragment that forms a tree, whose root is p .

$$t \mapsto \langle l, r \rangle * \text{tree}(l) * \text{tree}(r)$$

The *specification language* describes how a statement transforms a symbolic heap into another. The specification is typically made up of Hoare triples, whose pre- and post- condition are written using the assertion language. Here are some key rules of separation logic (Figure 2.3). Mezzo possesses similar subsumption rules (FRAME, TRANSITIVE in §8.2).

$$\begin{array}{ccc} \text{SL-FRAME} & \text{SL-PAR} & \text{SL-CONSEQUENCE} \\ \frac{\{P\} c \{Q\}}{\{P * R\} c \{Q * R\}} & \frac{\{P\} c \{Q\} \quad \{P'\} c' \{Q'\}}{\{P * P'\} c \parallel c' \{Q * Q'\}} & \frac{P \Vdash P' \quad \{P'\} c \{Q'\} \quad Q' \Vdash Q}{\{P\} c \{Q\}} \end{array}$$

Figure 2.3: The key rules of separation logic

The most important rule is perhaps SL-FRAME. In an informal manner, if a statement c requires a heap predicate P and turns it into Q , then it leaves the rest R of the heap untouched. Indeed, the $*$ connective *bakes in* the fact that its two sides live in distinct fragments of the heap. One could sum this up as “everyone minds their own business”.

Other important rules of separation logic include the consequence rule, which allows for a natural *narrative* style of proof, where the current assertion is weakened using an *entailment relation* \Vdash as we go through the program. This is the core idea of Hoare logic, on which separation logic is based.

Separation logic is also remarkably well-suited for reasoning about concurrent programs. If c and c' run in parallel, SL-PAR tells us that as long as they operate on physically disjoint heap fragments, they will not interfere.

These rules are quite general. Here is an example of an axiom for assignment. The way this rule is written is somehow reminiscent of a weakest precondition calculus.

$$\text{SL-ASSIGN} \\ \{[E/x]\Pi\} x := E \{ \Pi \}$$

Reading backwards, the axiom states that in order to prove that the statement “ $x := E$ ” gives the heap predicate Π , then it suffices to show $[E/x]\Pi$, that is, Π where all occurrences of x have been replaced with E .

Let us illustrate these rules using a classic example, which is that of the `dispose` function. This function frees a heap-allocated tree, and is used in the Smallfoot papers [BCO05b]).

```
void dispose(tree* p) {
  if (p != NULL) {
    dispose(p->left) || dispose(p->right);
    free(p);
  }
}
```

Two recursive calls are required, so as to free the left and right sub-trees, before freeing the node itself. The recursive calls are performed in parallel.

Without separation logic, the pre- and post-conditions for `dispose` need to explicitly assert that the function only frees elements reachable via `p` and leaves other parts of the heap untouched. That is, we need to state non-interference properties about the function, to be able to reason in a modular manner.

With separation logic, the specification for this function admits a natural formulation.

$$\{\text{tree}(p)\} \text{dispose}(p) \{\text{empty}\}$$

Thanks to the frame rule, the fact that `dispose` only operates on the fragment of the heap covered by the $\text{tree}(p)$ predicate comes for free.

Here is how one would prove that the function matches its specification, using a combination of SL-FRAME, SL-PAR and SL-CONSEQUENCE. The Hoare triples are shown in math font, and denote the assertions that are valid at each program point.

```
void dispose(tree* p) {
  {tree(p)}
  if (p != NULL) {
    {p ↦ ⟨l; r⟩ * tree(l) * tree(r)}
    dispose(p->left) || dispose(p->right);
    {p ↦ ⟨l; r⟩}
    free(p);
    {emp}
  }
}
```

The purpose of the present section is not to cover separation logic exhaustively; a book would be the right format for that. Rather, let me try to highlight a few key points about separation logic that are reminiscent, or in contrast with problems that need to be solved in Mezzo.

Spatial assertions and logical assertions Some authors find it desirable to split heap predicates into spatial parts and logical parts [BCO05b, CDOY09]. Logical parts contain equality and disequality predicates. Spatial parts contain points-to assertions (written $x \mapsto \cdot$) separated by the $*$ connective. The right-hand-side of points-to assertions is made up of memory blocks.

Mezzo bundles everything together as permissions. While equations get a special treatment (via EQUALS-FOR-EQUALS) in the subsumption relation as well as in the implementation, they are presented as regular permissions of the form $x @ = y$ using the singleton type mechanism.

The entailment problem When checking a function *definition*, one starts with Π_{pre} , the function pre-condition. This is the assertion (also known as “heap predicate”) that holds when one enters the function body. One then steps through the function definition, applying rules from the specification to update the *current assertion*, until one reaches the end of the function body, obtaining a heap predicate Π . The function is annotated with a post-condition Π_{post} .

One then needs to answer the question: does Π entail Π_{post} ? This is usually written as $\Pi \Vdash \Pi_{\text{post}}$.

Here is an example of two entailment problems: the first assertion entails $\text{tree}(t)$, but the second one does not, for the $\text{tree}(t)$ predicate denotes a tree whose two sub-trees are physically distinct.

$$\begin{aligned} t \mapsto \langle l; r \rangle * \text{tree}(l) * \text{tree}(r) &\quad \Vdash \text{tree}(t) \\ t \mapsto \langle l; r \rangle * \text{tree}(l) * l = r &\quad \not\Vdash t \mapsto \langle l; r \rangle * \text{tree}(l) * \text{tree}(r) \end{aligned}$$

Solving the entailment problem depends very much on the specification language chosen, as well as the assertion language that one manipulates. One can design systems where entailment is decidable [BCO04].

Entailment is also crucial when stepping through the program (SL-CONSEQUENCE); one needs to transform the current assertion into the one needed for the next program instruction. This is done via the entailment relation.

In Mezzo, the entailment problem is called *subtraction* and is the topic of Chapter 11.

The frame inference problem The declarative SL-FRAME rule from Figure 2.3 does not provide a way to *compute* the actual fragment R that is left untouched. In other words, how does one separate a heap H into a P part (the “smallest” one required to call a function) and the R part (the leftover)?

The specifics of frame inference will, again, depend on the particulars of the system. The Smallfoot system [BCO05b], which I mentioned earlier, offers an incomplete frame inference procedure.

In Mezzo, the subtraction operation also performs frame inference.

Built-in vs. user-defined predicates Many early works feature built-in predicates for the spatial part of assertions. Typical spatial predicates include $\text{list}(x)$, $\text{listseg}(x)$, $\text{tree}(x)$. This somehow restrictive state of things was lifted later on [NDQCo7] by allowing assertions to mention user-defined predicates.

Allowing user-defined predicates is not a trivial change: as the previous Alias Types examples showed (§2.2), unfolding an inductive predicate amounts to unpacking an existential. This means that the entailment and frame inference problems need to deal with quantifiers, something that was not covered in earlier works.

Permissions in object-oriented languages

The object-oriented community also tried to build upon existing type systems to provide stronger static guarantees and rule out more bugs. In particular, a salient issue in an object-oriented setting is object protocols, also known as “typestate” in the literature [SY86].

Objects go through different *states* during their lifetime. The typical example is that of a file descriptor that goes from ready to closed. This is purely conceptual, though: the type system of, say, Java, will not enforce these conventions, and the user must rely on documentation and manual code review to verify what are these implicit conventions and whether they are enforced.

An early strand of work is Fahndrich and DeLine’s Vault [DFo1] and Fugue [DFo4] languages. The former provides reasoning about typestate for a low-level language, while the latter tackles high-level, object oriented languages. The main restriction, though, is that Fugue provides limited guarantees in the presence of aliasing.

Indeed, typestate and alias control are two closely related problems. If one piece of code moves a file descriptor from ready to closed, then we must make sure we are aware of all the aliases to the object. Failing to track an alias means that somewhere, an outdated, inconsistent vision of the world (“the object is still ready”) exists, which can lead to crashes.

Plural A first line of work that tries to talk about typestate in the presence of aliasing is Plural [BA07, BBA11]. The idea is to take a well-typed Java program, annotate it with extra state specifications, and run an external tool, Plural, to check that the protocols are well-enforced.

At each program point, a set of *permissions* describe the kind of ownership *I* have; these permissions also describe the kind of ownership *others* have. Here, “*I*” and “*others*” should be understood as “the currently executing piece of code” and “other pieces of code that may hold aliases to my object”. There are as many as five (Figure 2.4) possible permissions for an object, but only some variants can co-exist at the same time.

Access through other permissions	Current permission has ...	
	Read-write access	Read-only access
None	unique	–
Read-only	full	immutable
Read-write	share	pure

Figure 2.4: The zoology of permissions in Plural

An example of permission is $\text{full}(\text{this})$, which indicates that a method requires read-write access to the receiver, and demands that other parts of the code only hold read-only aliases to the receiver. Permissions may be refined with an *in* clause, which indicates ownership of the object in one particular state. A permission describing a file descriptor that is ready and shared in a read-write fashion with others would thus be $\text{share}(\text{fd})$ in ready.

These permissions are user-facing permissions; internally, Plural uses more complex permissions.

Specification of the various states is done using state diagrams. This means that there is a hierarchy of states that is formed by a tree. The children of a state are understood to be *refinements* of that state.

Method specifications are embedded in a decidable fragment of linear logic. A classic example is the iterator, where the `hasNext` method either returns `true`, indicating that the iterator has more elements available, or returns `false`, indicating that the iterator has come to an end. The method specification for `hasNext` is:

$$\begin{aligned} \text{pure}(\text{this}) \quad & \multimap (\text{result} = \text{true} \otimes \text{pure}(\text{this}) \text{ in available}) \\ & \oplus (\text{result} = \text{false} \otimes \text{pure}(\text{this}) \text{ in end}) \end{aligned}$$

Permissions can be split when one desires to create aliases: an object is initially held as unique, but this permission can be, for instance, split into a combination of a full and a pure permission. One naturally wishes to recombine

these two permissions to obtain the stronger, original one. This is done using fractional permissions [Boy03].

$$\text{pure}(\text{this}, 1) \equiv \text{full}(\text{this}, \frac{1}{2}) \otimes \text{pure}(\text{this}, \frac{1}{2})$$

Some operations, such as refining into a more precise state, do not demand full ownership of the object. Some other operations, such as moving an object into a coarser state, will require a unique permission to ensure consistency with other aliases.

Plural is a static analysis, as the permissions that are manipulated exist purely at compile-time, not at run-time.

The Plural analysis does not have false negatives, that is, it will not flag correct programs for errors; it fails, however, to catch all protocol violations, meaning that some incorrect programs will still be accepted by the tool.

Mezzo features a much simpler permission model, and does not include the sophistications of a language such as Plural. Mezzo can express state change, albeit in a slightly different fashion. While in Plural the central concept is objects and classes, in Mezzo, the central feature of the language would certainly be data types. In the tail-recursive concatenation example (§3.2), cells change state and go from uninitialized Cells to well-formed Cons blocks; this would be an example of state change in Mezzo.

Plaid Plaid [SNS⁺₁₁, Ald10], in contrast to Plural, is a brand new language. In Plaid, states are first-class objects that can be composed, re-used, and that can inherit from one another. Concretely, the programmer, instead of defining classes just like in Java, defines states. Similarly, instead of extending a parent class, the programmer refines a parent state. This amounts to defining a state chart via the various inheritance relationships.

A consequence of this choice is that, unlike Plural, states have a run-time representation: states are object that exist at run-time; one can determine the actual state of an object by performing a run-time test. A drawback is that whenever one performs a state change, a run-time operation has to take place to actually reflect the state change on the object that lives in memory.

2.3 Other related works

This final section presents some works that, even though they didn't serve as direct sources of inspiration, share the same aspirations or goals as Mezzo. There are too many to list them all; I present a (necessarily) incomplete selection.

Ownership types

The object-oriented community has been aware for a long time that undesired aliasing can cause representation exposure, failures to track invariants and prevent reasoning in a modular fashion.

Ownership Types ([CPN98, CÖSW13], as well as a long series of papers in-between these two) offer a key idea, which is to structure the heap according to an ownership hierarchy.

An object owns a context, which may be understood, in the words of separation logic, as a heaplet. The object may allocate other objects in that context. Only the owner *or* the inhabitants of the context may refer to objects living in that context. This means that outside references are disallowed: the only way to refer to an object is to go through the object hierarchy and follow a path that traverses all of the object's owners.

Contexts appear as part of types. This means that, by looking at the types of two objects, if the contexts are distinct, then the two objects live in separate parts of the heap, hence cannot be aliases.

There is an important point that the paper makes: possessing a reference to an object and owning it are two separate concepts. This is something that also appears in Alias Types (the $\text{ptr}(\ell)$ type), as well as in Mezzo.

Cyclone

Cyclone [JMG⁺₀₂, GMJ⁺₀₂, SHM⁺₀₆] is a type-safe dialect of C dedicated to low-level programming. The goal of Cyclone is to have, just like Tofte and Talpin, predictable memory deallocation using regions. They note, like so many others, that the last-in first-out discipline of regions induced by the lexical `letregion` construct is too restrictive. This is particularly true of long-lived applications: imagine an application with an event loop, for instance. If the region is scoped within the loop, variables cannot persist across iterations. If the region's scope encompasses the loop, variables are kept allocated until the end of execution, hence creating an unbounded memory leak.

Cyclone thus incorporates a remarkable variety of mechanisms to handle allocation and deallocation: a static region mechanism, a conservative garbage collector, unique pointers, reference-counted pointers, dynamically-sized regions, LIFO arenas... The authors mention at numerous times that formalizing the entire system is too complex, and focus instead on restrictions which they sometimes dub “core Cyclone”.

The concept of static region is used widely: a stack frame is a static region, the heap is also another static region. Using a type-and-effect system allows tracking which regions are still alive. The system naturally features region polymorphism.

Cyclone, having unique pointers, also encounters the need to temporarily alias a uniquely-owned object. I mentioned already that Wadler bumped into the issue early on; Cray *et al.* in the Calculus of Capabilities [CWM99] use subtyping. Here, the authors of Cyclone use a `let alias` construct that is similar in spirit to Wadler’s mechanism and relies on syntactic scope to guarantee that aliases do not escape. This mechanism is also implemented using a locally-scoped region.

Rust

Rust [The14] is a new systems-oriented programming language designed by Mozilla. Rust’s stance is to incorporate high-level constructs from modern programming languages into a language that still allows the user to retain control over low-level features, such as manual memory management.

Compared to Mezzo, Rust thus needs to expose much more information to the programmer: where does a variable live (stack or heap), how the lifetime of a variable is managed, whether an object is boxed or not, whether a closure lives on the stack or on the heap, etc. Exposing these concerns is necessary: for instance, objects, just like in C++, have destructors.

The technical ingredients that Rust uses are: a discipline of unique pointers, which can be temporarily *borrowed*, along with a mutability analysis that determines which operations are legal or not.

Rust also has a notion of ownership: unique ownership of data structures is recursive. Borrowing an object grants non-unique ownership of it. Rust is equipped with a mechanism of *traits* which can act either as performing a dynamic dispatch or a static dispatch.

At the time of this writing, it seems that Rust is leaning towards a stronger emphasis on *ownership* rather than *mutability*.

An important difference is that Rust currently does not have a formal model, even though some attempts have been made [Mat14].

Vault

Vault [DFo1] is another attempt at bridging the gap between high-level reasoning and low-level software. The idiom used in Vault is that of a *type key*: the type of each object mentions its key. The compiler keeps track at compile-time of which keys are available; each key is a token that models a distinct resource. All keys are linear. Functions can demand that a key be present; functions may also consume keys; the pre-condition and post-conditions are checked at compile-time. Keys may be in a particular *state*.

Having linear keys naturally allows the compiler to properly track ownership and state change via keys. It incurs the usual restrictions on aliasing, though. Vault thus uses a region abstraction: creating a new region allocates a new key, and the types of objects within the region mention the same key. Thus, whenever the key for the region is available, one can use the objects within the region. Conversely, once the key is gone, one can safely deallocate the entire region. The ownership of a key thus grants ownership of all objects within the region. The mechanism remains purely static.

The authors also track local aliasing relationship using singleton types, just like in Mezzo. Interestingly, they mention the issue of computing type agreement at join points, which is the topic of Chapter 12.

Part II

A taste of Mezzo

3	A Mezzo tutorial	
3.1	Three small examples	25
3.2	Lists	31
3.3	Breaking out: arbitrary aliasing of mutable data structures	41
4	Bits from the standard library	
4.1	Nesting	49
4.2	Adoption/abandon as a library	54
4.3	One-shot functions	56
4.4	Rich booleans	56
4.5	Locks and conditions, POSIX-style	59
4.6	Landin's knot (recursion via the mutable store)	63
4.7	Other interesting bits	64
5	Writing programs in Mezzo	
5.1	Ownership and function signatures	65
5.2	Higher-order effects and crafting signatures	68
5.3	Reifying coercions	70
5.4	Object-oriented programming	72

This part is about Mezzo, as a language: I wish to illustrate the language in various respects. The first chapter tries to give the reader a sense of what the language *feels like*. There are many examples that illustrate the way the language works; how programs are type-checked; the kind of guarantees our novel type system provides. This chapter is completely informal, yet, the user should feel fairly familiar with Mezzo after reading it. Hopefully, the reader should also appreciate the design effort that was spent on creating a user-facing language, rather than just a core, low-level calculus.

The subsequent chapter takes a few pieces of code from the Mezzo standard library and comments them, highlighting the strengths or limitations of the language. Some more arcane features of the language are showcased, and a few algorithms are fleshed out.

The last chapter reflects on the art of writing Mezzo programs and tries to identify a few common patterns, as well as guidelines for writing programs.

3. A Mezzo tutorial

This chapter, just like Chapter 4 and Chapter 5, uses the *surface language*, that is, the language that the programmer manipulates. The surface language provides several syntactic and semantic facilities that make programming in Mezzo easier. Hopefully, this chapter will feel natural and the reader, and the extent of these facilities will only become apparent in Part III, where I explain all the implicit mechanisms and translate them away in a simpler, more regular core calculus.

3.1 Three small examples

Let me begin with three small examples that should, hopefully, give the reader a good preview of Mezzo. These examples are small, self-contained, and serve as a good basis for explaining our typing discipline. Later sections feature more full-fledged examples.

Write-once references

A write-once reference is a memory cell that can be assigned at most once and cannot be read before it has been initialized.

A client of the module Figure 3.1 shows some client code that manipulates a write-once reference. The Mezzo type system guarantees that the user must call `set` before using `get`, and can call `set` at most once.

```
1 open woref
2
3 val _ : (int, int) =
4   let r = new () in
5     set (r, 3);
6     (get r, get r)
```

Figure 3.1: Using a write-once reference.

At line 4, we create a write-once reference by calling `woref::new`. (Thanks to the declaration `open woref`, one can refer to this function by the unqualified name `new`.) The local variable `r` denotes the address of this reference. In the eyes of the type-checker, this gives rise to a *permission*, written `r @ writable`. This permission has a double reading: it describes the layout of memory (i.e., “the variable `r` denotes the address of an uninitialized memory cell”) and grants *exclusive* write access to this memory cell. That is, the type constructor `writable` denotes a uniquely-owned writable reference, and the permission `r @ writable` is a unique token that one must possess in order to write `r`. (We explain later on how the system, via a lookup of the definition of `writable`, knows that `r @ writable` is exclusive.)

Permissions are tokens that exist at type-checking time only. Many permissions have the form `x @ t`, where `x` is a program variable and `t` is a type. At a program point where such a permission is available, we say informally

```

1  data mutable writable =
2    Writable { contents: () }
3
4  data frozen a =
5    Frozen  { contents: (a | duplicable a) }
6
7  val new () : writable =
8    Writable { contents = () }
9
10 val set [a] (consumes r: writable, x: a | duplicable a)
11     : (| r @ frozen a) =
12     r.contents <- x;
13     tag of r <- Frozen
14
15 val get [a] (r: frozen a) : a =
16     r.contents

```

Figure 3.2: Implementation of write-once references

that “ x has type t (now)”. Type-checking in Mezzo is flow-sensitive: at each program point, there is a *current permission*, which represents our knowledge of the program state at this point, and our rights to alter this state. The current permission is typically a conjunction of several permissions. The conjunction of two permissions p and q is written $p * q$.

Permissions replace traditional type assumptions. A permission $r @ \text{writable}$ superficially looks like a type assumption $r : \text{writable}$. However, a type assumption would be valid everywhere in the scope of r , whereas a permission should be thought of as a token: it can be passed from caller to callee, returned from callee to caller, passed from one thread to another, etc. If one gives up this token (say, by assigning the reference), then, even though r is still in scope, one can no longer write it.

At line 5, we exercise our right to call the `set` function, and write the value 3 to the reference r . In the eyes of the type-checker, this call *consumes* the token $r @ \text{writable}$, and instead produces another permission, $r @ \text{frozen int}$. This means that any further assignment is impossible: the `set` function requires $r @ \text{writable}$, which we no longer have. Thus, the reference has been rendered immutable. This also means that the `get` function, which requires the permission $r @ \text{frozen int}$, can now be called. Thus, the type system enforces the desired usage protocol.

The permissions $r @ \text{writable}$ and $r @ \text{frozen int}$ are different in one important way. The former denotes a uniquely-owned, writable heap fragment. It is *affine*: once it has been consumed by a call to `set`, it is gone forever. The latter denotes an immutable heap fragment. It is safe to share it: this permission is *duplicable*. If one can get ahold of such a permission, then one can keep it forever (i.e., as long as r is in scope) and pass copies of it to other parts of the program, if desired. Such a permission behaves very much like a traditional type assumption $r : \text{frozen int}$.

At line 6, we build a pair of the results of two calls to `get`. There is an implicit sequence: the dynamic semantics of Mezzo prescribes left-to-right evaluation order. The second call is type-checked using whatever permissions remain after the first call. Here, the duplicable permission $r @ \text{frozen int}$ is implicitly copied, so as to justify the two calls to `get`.

We now explain how this module is implemented. Its code appears in Figure 3.2.

To be or not to be duplicable The type `writable` (line 1) describes a mutable heap-allocated block. Such a block contains a `tag` field (which must contain the tag `Writable`, as no other data constructors are defined for this type) and a regular field, called `contents`, which has unit type. The function `new` (line 7) allocates a fresh memory block of type `writable` and initializes its `contents` field with the unit value. A call to this function, such as `let r = woref::new() in ...`, produces a new permission $r @ \text{writable}$.

The definition of `writable` contains the keyword `mutable`. This causes the type-checker to regard the type `writable`, and every permission of the form $r @ \text{writable}$, as affine (i.e., non-duplicable). This ensures that $r @$

`writable` represents exclusive access to the memory block at address x . If one attempts to duplicate this permission (for instance, by writing down the static assertion `assert (r @ writable * r @ writable)`, or by attempting to call `set (r, ...)` twice), the type-checker rejects the program.

The parameterized data type `frozen a` (line 4) describes an immutable heap-allocated block. Such a block contains a tag field (which must contain the tag `Frozen`) and a regular field, also called `contents`¹, which has type `(a | duplicable a)`.

Before going further, we must say a little more about the syntax of types. The type `(a | duplicable a)` is of the form `t | p`: indeed, `a` is a type, while `duplicable a` is a permission². This means that the value stored at runtime in the `contents` field has type `a`, and is logically accompanied by a proof that the type `a` is duplicable.

More generally, a type of the form `t | p` can be thought of as a pair; yet, because permissions do not exist at runtime, a value of type `t | p` and a value of type `t` have the same runtime representation. We write `(| p)` for `(() | p)`, where `()` is the unit type. Another syntactic convention is that, by default (that is, unless the `consumes` keyword is used), a permission that appears in the domain of a function type is implicitly repeated in the codomain.

Why do we impose the constraint `duplicable a` as part of the definition of the type `frozen a`? The reason is, a write-once reference is typically intended to be shared after it has been initialized. (If one did not wish to share it, then one could use a standard read/write, uniquely-owned reference.) Thus, its content is meant to be accessed by multiple readers. This is permitted by the type system only if the type `a` is duplicable. Technically, the constraint `duplicable a` could be imposed either when the write-once reference is initialized, or when it is read. We choose the former approach because it is simpler to explain. The latter would work just as well, and would offer a little extra flexibility.

The definition of `frozen` does not contain the keyword `mutable`, so a block of type `frozen a` is immutable. Thus, it is safe to share read access to such a block. Furthermore, because we have imposed the constraint `duplicable a`, it is also safe to share the data structure of type `a` whose address is stored in the `contents` field. In other words, by inspection of the definition, the type-checker recognizes that the type `frozen a` is duplicable. This means that a write-once reference can be shared after it has been initialized.

Changing states: strong updates The use of the `consumes` keyword in the type of `set` (line 10) means that the caller of `set` must give up the permission `r @ writable`. In exchange, the caller receives a new permission for `r`, namely `r @ frozen a` (line 11). One may say informally that the type of `r` changes from “uninitialized” to “initialized and frozen”.

The code of `set` is in two lines. First, the value x is written to the field `r.contents` (line 12). After this update, the memory block is described by the permission `r @ Writable { contents: a }`.

Then, the tag of `r` is changed from `Writable` to `Frozen`: this is a *tag update* (line 13). This particular tag update instruction is ghost code: it has no runtime effect, because both `Writable` and `Frozen` are represented at runtime as the tag `o`. This pseudo-instruction is just a way of telling the type-checker that our view of the memory block `r` changes. After the tag update instruction, this block is described by the permission `r @ Frozen { contents: a }`.

This permission can be combined with the permission `duplicable a` (which exists at this point, because `set` requires this permission from its caller) so as to yield `r @ Frozen { contents: (a | duplicable a) }`. This is the right-hand side of the definition of the type `frozen a`. By folding it, one obtains `r @ frozen a`. Thus, the permissions available at the end of the function `set` match what has been advertised in the header (line 11).

In general, the tag update instruction allows changing the type of a memory block to a completely unrelated type, with two restrictions: (i) the block must initially be mutable; and (ii) the old and new types must have the same number of fields. This instruction is compiled down to either a single write to the tag field, or nothing at all, as is the case above.

An interface for `woref` Mezzo currently offers a simple notion of module, or unit. Each module has an implementation file (whose extension is `.mz`) and an interface file (whose extension is `.mzi`). This system supports

¹Contrary to OCaml, Mezzo allows two user-defined types to have a field by the same name.

²The user is encouraged to think of `duplicable a` as a regular permission; the syntax is crafted so that no difference exists between `t | p` and `t | duplicable a`. Both the formalization and the implementation, however, distinguish the two concepts. Part III clarifies this.

```

1 | abstract writable
2 | abstract frozen a
3 | fact duplicable (frozen a)
4 | val new: () -> writable
5 | val set: [a] (consumes r: writable, x: a | duplicable a)
6 |     -> (| r @ frozen a)
7 | val get: [a] frozen a -> a

```

Figure 3.3: Interface of write-once references

```

1 | val spawn: [p: perm] (
2 |   f: (| consumes p) -> ()
3 | | consumes p
4 | ) -> ()

```

Figure 3.4: Signature of the thread module

type abstraction as well as separate type-checking and compilation. It is inspired by OCaml and by its predecessor Caml-Light.

The interface of the module `woref` is shown in Figure 3.3³.

The type `writable` is made abstract (line 1) so as to ensure that `set` is the only action that can be performed with an uninitialized reference. If the concrete definition of `writable` was exposed, it would be possible to read and write such a reference directly, without going through the functions offered by the module `woref`.

The type `frozen` is also made abstract (line 2). One could expose its definition without endangering the intended usage protocol. Nevertheless, it is good practice to hide the details of its implementation; this may facilitate future evolutions.

The fact that `frozen a` is a `duplicable` type is published (line 3). In the absence of this declaration, `frozen a` would by default be regarded affine, so that sharing access to an initialized write-once reference would not be permitted. This fact declaration is implicitly universally quantified in the type variable `a`. One can think of it as a universally quantified permission, `[a] duplicable (frozen a)`, that is declared to exist at the top level. This permission is itself `duplicable`, hence exists everywhere and forever.

Indeed, lacking any specific annotation, abstract types are understood to be affine. The fact mechanism is similar to *variance* in ML signatures: type parameters are by default considered invariant, which is the less precise information we can have (type `'a t`). The user may provide variance annotations to *refine* the variance information (type `'a map+`). (Both facts and variance annotations form a lattice – this is developed in §8.6).

The remaining lines in Figure 3.3 declare the types of the functions `new`, `get`, and `set`, without exposing their implementation. In the type of `set`, the first argument `r` must be named (line 5), because we wish to refer to it in the result type (line 6). In a function header or in a function type, the name introduction form `r: t` binds the variable `r` and at the same time requests the permission `r @ t`. In contrast, in the permission `r @ t`, the variable `r` occurs free. The second argument of `set`, `x`, need not be named; we name it anyway (line 5), for the sake of symmetry.

A race

We now consider the tiny program in Figure 3.5. This code exhibits a data race, hence is incorrect, and is rejected by the type system. Indeed, the function `f` increments the global reference `r`. The main program spawns two threads that call `f`. There is a data race: both threads may attempt to modify `r` at the same time. Let us explain how the type-checker determines that this program must be rejected.

³ The fact that we require the elements stored in write-once references to be `duplicable` makes crafting a signature for the module easier. Lacking the `duplicable` hypothesis, writing a proper signature for the module would require quite some expertise. The issue is discussed at length in §5.1.

```

1  open thread
2
3  val r = newref 0
4  val f (| r @ ref int) : () =
5      r := !r + 1
6  val () =
7      spawn f; spawn f

```

Figure 3.5: Ill-typed code.

The signature of `spawn` is shown in Figure 3.4. The function consumes a permission p , which it then passes to a function that takes no argument, returns no value, and requires p to execute.

The racy program is shown in Figure 3.5. At line 3, we allocate a (classic) reference r , thus obtaining a new permission $r @ \text{ref int}$. A write-once reference would not suffice: we want to write several times into the reference.

The function f at line 4 takes no argument and returns no result. Its type is not just $() \rightarrow ()$, though. Because f needs access to r , it must explicitly request the permission $r @ \text{ref int}$ and return it. (The fact that this permission is available at the definition site of f is not good enough: a closure cannot capture a non-duplicable permission.⁴) This is declared by the type annotation. Thus, at line 6, in conjunction with $r @ \text{ref int}$, we have a new permission, $f @ (| r @ \text{ref int}) \rightarrow ()$. This means that f is a function of no argument and no result (at runtime), which (at type-checking time) requires and returns the permission $r @ \text{ref int}$.

By our syntactic conventions for function types, $f @ (| r @ \text{ref int}) \rightarrow ()$ means that f requires and returns the permission $r @ \text{ref int}$. When one wishes to indicate that a function requires some permission but does not return it, one must precede that permission with the keyword `consumes`.

This is the first time this situation arises: previously, permissions were either always consumed, or duplicable, such as in the `set` function.

On line 7, there is a sequencing construct. The second call to `spawn` is type-checked using the permissions that are left over after the first `spawn`. A call `spawn f` requires two permissions: a permission to invoke the function f , and $r @ \text{ref int}$, which f itself requires. It does *not* return these permissions: they are transferred to the spawned thread. Thus, in line 7, between the two `spawn`s, we no longer have a permission for r . (We still have $f @ (| r @ \text{ref int}) \rightarrow ()$, as it is duplicable.) Therefore, the second `spawn` is ill-typed. The racy program of Figure 3.5 is rejected.

This behavior is to be contrasted with that of the earlier example. In Figure 3.1, the permission $r @ \text{frozen int}$, which `get` requires, is duplicable. We can therefore obtain two copies of it and justify two successive calls to `get`.

A fix for the race

In order to fix the racy program, one must introduce enough synchronization so as to eliminate the race. A common way of doing so is to introduce a lock and place all accesses to r within critical sections. In Mezzo, this can be done, and causes the type-checker to recognize that the code is now data-race free.

Figure 3.6 shows a corrected version of Figure 3.5, where a lock has been introduced to fix the program. In fact, this common pattern can be implemented as a polymorphic, higher-order function, `hide` (Figure 3.7), which we explain below.

In Figure 3.7, the polymorphic, higher-order function `hide` takes a function f of type $(\text{consumes } a \mid s) \rightarrow (b \mid s)$, which means that f needs access to a piece of state represented by the permission s . `hide` requires s , and consumes it. It returns a function of type $(\text{consumes } a) \rightarrow b$, which does not require s , hence can be invoked by multiple threads concurrently. The type variables a and b have kind `type`. The square brackets denote universal quantification.

⁴The fact that Mezzo only offers duplicable closures may seem surprising. However, affine closures can be defined as a library (§4.3). Moreover, this keeps the system simple and saves the need for introducing various kinds of arrows depending on whether the underlying closure is duplicable or not.


```
1  open thread
2
3  val r = newref 0
4  val l: lock::lock (r @ ref int) = lock::new ()
5  val f () : () =
6    lock::acquire l;
7    r := !r + 1;
8    lock::release l
9  val () =
10   spawn f; spawn f
```

Figure 3.6: Fixing the racy program, using a lock

```
1  open lock
2
3  val hide [a, b, s : perm] (
4    f : (consumes a | s) -> b |
5    consumes s
6  ) : (consumes a) -> b =
7    let l : lock s = new () in
8    fun (consumes x : a) : b =
9      acquire l;
10     let y = f x in
11     release l;
12     y
```

Figure 3.7: Hiding internal state via a lock, generic version

Locks are discussed in thorough detail in §4.5 and Figure 4.8. Let us, for the time being, explain this code snippet succinctly.

In Figure 3.7, `f` is a parameter of `hide`. It has a visible side effect: it requires and returns a permission `s`. When `hide` is invoked, it creates a new lock `l`, whose role is to govern access to `s`. At the beginning of line 7, we have two permissions, namely `s` and `f @ (consumes a | s) -> b`. At the end of line 7, after the call to `lock::new`, we have given up `s`, which has been consumed by the call, and we have obtained `l @ lock s`. The lock is created in the “released” state, and the permission `s` can now be thought of as owned by the lock.

At line 8, we construct an anonymous function. This function does not request any permission for `f` or `l` from its caller: according to its header, the only permission that it requires is `x @ a`. Nevertheless, the permissions `f @ (consumes a | s) -> (b | s)` and `l @ lock s` are available in the body of this anonymous function, because they are duplicable, and a closure is allowed to capture a duplicable permission.

The fact that `l @ lock s` is duplicable is a key point. Quite obviously, this enables multiple threads to compete for the lock. More subtly, this allows the lock to become hidden in a closure, as illustrated by this example. Let us emphasize that `s` itself is typically *not* duplicable (if it were, we would not need a lock in the first place).

The anonymous function at line 8 does not require or return `s`. Yet, it needs `s` in order to invoke `f`. It obtains `s` by acquiring the lock, and gives it up by releasing the lock. Thus, `s` is available only to a thread that has entered the critical section. The side effect is now hidden, in the sense that the anonymous function has type `(consumes a) -> b`, which does not mention `s`.

It is easy to correct the code in Figure 3.5 by inserting the redefinition `val f = hide f` before line 6. This call consumes `r @ ref int` and produces `f @ () -> ()`, so the two `spawn` instructions are now type-correct. Indeed, the modified code is race-free.

3.2 Lists

The example of write-once references has allowed us to discuss a number of concepts, including affine versus duplicable permissions, mutable versus immutable memory blocks, and strong updates. References are, however, trivial data structures, in the sense that their exact shape is statically known. We now turn to lists. Lists are data structures of statically unknown length, which means that many functions on lists must be recursive. Lists are representative of the more general case of tree-structured data.

The algebraic data type of lists, `list a`, is defined in a standard way (Figure 3.8). This definition does not use the keyword `mutable`. These are standard immutable lists, that is, lists with an immutable spine. The list elements may be mutable or immutable, depending on how the type parameter `a` is instantiated.

Concatenation

Our first example of an operation on lists is concatenation. There are several ways of implementing list concatenation in `Mezzo`. We begin with the function `append`, also shown in Figure 3.8, which is the most natural definition.

The type of `append` (line 5) states that this function takes two arguments `xs` and `ys`, together with the permissions `xs @ list a` and `ys @ list a`, and produces a result, say `zs`, together with the permission `zs @ list a`. The `consumes` keyword indicates that the permissions `xs @ list a` and `ys @ list a` are not returned: the caller must give them up. Before discussing the implications of this fact, let us first explain how `append` is type-checked.

At the beginning of line 6, the permission `xs @ list a` guarantees that `xs` is the address of a list, i.e., a memory block whose `tag` field contains either `Nil` or `Cons`. This justifies the `match` construct: it is safe to read `xs`'s `tag` and to perform case analysis.

Upon entry in the first branch, at the beginning of line 8, the permission `xs @ list a` has been replaced with `xs @ Nil`. We refer to the latter as a *structural permission*. It is more precise than the former; it tells us not only that `xs` is a list, but also that its `tag` must be `Nil`. This knowledge, it turns out, is not needed here: `xs @ Nil` is not exploited when type-checking this branch. On line 8, we return the value `ys`. The permission `ys @ list a` is used to justify that this result has type `list a`, as advertised in the function header. This consumes `ys @ list a`, which is an affine permission.

Upon entry in the second branch, at the beginning of line 10, our knowledge about `xs` also increases. The permission `xs @ list a` is replaced with `xs @ Cons { head: a; tail: list a }`. Like `xs @ Nil`, this is a structural permission. It is obtained by looking up the definition of the data type `list a` and specializing it for the tag `Cons`.

The pattern `Cons { head; tail }` on line 9 involves a pun: it is syntactic sugar for `Cons { head = head; tail = tail }`, which binds the variables `head` and `tail` to the contents of the fields `xs.head` and `xs.tail`, respectively. Thus, we now have two names, `head` and `tail`, to refer to the values stored in these fields. This allows the type-checker to expand the structural permission above into a conjunction of three atomic permissions:

```
xs @ Cons { head: =head; tail: =tail } *
head @ a *
tail @ list a
```

The first conjunct describes just the memory block at address `xs`. It indicates that this block has tag `Cons`, that its `head` field contains the value `head`, and that its `tail` field contains the value `tail`. The types `=head` and `=tail` are *singleton types* [SWMoo]: each of them is inhabited by just one value. This first conjunct can also be written `xs @ Cons { head = head; tail = tail }`. In the following, this permission is not used, so we do not repeat it, even though it remains available until the end.

The second conjunct describes just the first element of the list, that is, the value `head`. It guarantees that this value has type `a`, so to speak, or more precisely, that we have permission to use it at type `a`. The last conjunct describes just the value `tail`, and means that we have permission to use this value as a list of elements of type `a`.

In order to type-check the code on line 10, the type-checker automatically expands it into the following form, where every intermediate result is named:

```
110 let ws = append (tail, ys) in
111 let zs = Cons { head = head; tail = ws } in
112 zs
```

```
1 data list a =
2   | Nil
3   | Cons { head: a; tail: list a }
4
5 val rec append [a] (consumes (xs: list a, ys: list a)) : list a =
6   match xs with
7   | Nil ->
8     ys
9   | Cons { head; tail } ->
10     Cons { head; tail = append (tail, ys) }
11 end
```

Figure 3.8: Definition of lists and list concatenation

```
1 data mutable cell a =
2   Dummy | Cell { head: a; tail: () }
3
4 val rec appendAux [a] (consumes (
5   dst: Cell { head: a; tail: () },
6   xs: list a,
7   ys: list a
8 )) : (| dst @ list a) =
9   match xs with
10  | Nil ->
11    dst.tail <- ys;
12    tag of dst <- Cons
13  | Cons ->
14    let dst' = Cell { head = xs.head; tail = () } in
15    dst.tail <- dst';
16    tag of dst <- Cons;
17    appendAux (dst', xs.tail, ys)
18  end
19
20 val append [a] (consumes (xs: list a, ys: list a)) : list a =
21   match xs with
22   | Nil ->
23     ys
24   | Cons ->
25     let dst = Cell { head = xs.head; tail = () } in
26     appendAux (dst, xs.tail, ys);
27     dst
28  end
```

Figure 3.9: List concatenation in tail-recursive style

The call `append (tail, ys)` on line 110 requires and consumes the permissions `tail @ list a` and `ys @ list a`. It produces the permission `ws @ list a`. Thus, after this call, at the beginning of line 111, the current permission is:

```
head @ a *
ws @ list a
```

The permission `head @ a`, which was not needed by the call `append (tail, ys)`, has been implicitly preserved. In the terminology of separation logic, it has been “framed out” during the call.

The memory allocation expression `Cons { head = head; tail = ws }` on line 111 requires no permission at all, and produces a structural permission that describes the newly-allocated block in an exact manner. Thus, after this allocation, at the beginning of line 112, the current permission is:

```
head @ a *
ws @ list a *
zs @ Cons { head = head; tail = ws }
```

At this point, since `append` is supposed to return a list, the type-checker must verify that `zs` is a valid list. It does this in two steps. First, the three permissions above can be conflated into one composite permission:

```
zs @ Cons { head: a; tail: list a }
```

This step involves a loss of information, as the type-checker forgets that `zs.head` is `head` and that `zs.tail` is `ws`. Next, the type-checker recognizes the definition of the data type `list`, and folds it:

```
zs @ list a
```

This step also involves a loss of information, as the type-checker forgets that `zs` is a `Cons` cell. Nevertheless, we obtain the desired result: `zs` is a valid list. So, `append` is well-typed.

When is a list duplicable? It is natural to ask: what is the status of the permission `xs @ list t`, where `t` is a type? Is it duplicable or affine?

Since the list spine is immutable, it is certainly safe to share (read) access to the spine. What about the list elements, though? If the type `t` is duplicable, then it is safe to share access to them, which means that it is safe to share the list as a whole. Conversely, if the type `t` is not duplicable, then `list t` must not be duplicable either. In short, the *fact* that describes lists is:

```
fact duplicable a => duplicable (list a)
```

This fact is inferred by the type-checker by inspection of the definition of the type `list`. If one wished to export `list` as an abstract data type, this fact could be explicitly written down by the programmer in the interface of the `list` module.

By exploiting this fact, the type-checker can determine, for instance, that `list int` is duplicable, because the primitive type `int` of machine integers is duplicable; and that `list (ref int)` is not duplicable, because the type `ref t` is affine, regardless of its parameter `t`.

A type variable `a` is regarded as affine, unless the permission `duplicable a` happens to be available at this program point. In the definition of `append` (Figure 3.8), no assumption is made about `a`, so the types `a` and `list a` are considered affine.

To consume, or not to consume Why must `append` consume the permissions `xs @ list a` and `ys @ list a`? Could it, for instance, *not* consume the latter?

In order to answer this question, let us attempt to change the type of `append` to `[a] (consumes xs: list a, ys: list a) -> list a`, where the `consumes` keyword bears on `xs` only. Recall that, by convention, the absence of the `consumes` keyword means that a permission is requested and returned. In other words, the above type is in fact syntactic sugar for the following, more verbose type:

```
[a] (consumes xs: list a, consumes ys: list a)
  -> (list a | ys @ list a)
```

It is not difficult to understand why `append` does *not* have this type. At line 8, where `ys` is returned, one would need two copies of the permission `ys @ list a`: one copy to justify that the result of `append` has type `list a`, and one copy to justify that the argument `ys` still has type `list a` after the call. Because the type `list a` is affine, the type-checker rejects the definition of `append` when annotated in this way.

A similar (if slightly more complicated) analysis shows that the `consumes` annotation on `xs` is also required.

These results make intuitive sense. The list `append (xs, ys)` *shares* its elements with the lists `xs` and `ys`. When the user writes `let zs = append (xs, ys) in ...`, they cannot expect to use `xs`, `ys` *and* `zs` as if they were lists with disjoint sets of elements. If the permission `xs @ list (ref int) * ys @ list (ref int)` exists before the call, then, after the call, this permission is gone, and `zs @ list (ref int)` is available instead. The integer references are now accessible through `zs`, but are no longer accessible through `xs` or `ys`.

The reader may be worried that this discipline is overly restrictive when the user wishes to concatenate lists of duplicable elements. What if, for instance, the permission prior to the call is `xs @ list int * ys @ list int`? There is no danger in sharing an integer value: the type `int` is duplicable. It would be a shame to lose the permissions `xs @ list int` and `ys @ list int`. Fortunately, these permissions are duplicable. So, even though `append` requests them and does not return them, the caller is allowed to copy each of them, pass one copy to `append`, and keep the other copy for itself. The type-checker performs this operation implicitly and automatically. As a result, after the call, the current permission is `xs @ list int * ys @ list int * zs @ list int`: all three lists can be used at will.

Technically, this phenomenon may be summed up as follows. In a context where the type `t` is known to be duplicable, the function types `(consumes t) -> u` and `t -> u` are equivalent, that is, subtypes of one another. It would be premature to prove this claim at this point; let us simply say that one direction is obvious, while the other direction follows from the frame rule and the duplication rule.

As a corollary, the universal type `[a] (consumes (list a, list a)) -> list a`, which is the type of `append` in Figure 3.8, is strictly more general than the type `[a] (list a, list a | duplicable a) -> list a`, where the `consumes` keyword has been removed, but the type `a` of the list elements is required to be duplicable. In short, this explains why `append` effectively does *not* consume its arguments when they have duplicable type.

List concatenation in tail-recursive style

The `append` function that we have discussed so far is a direct translation into Mezzo of the standard definition of list concatenation in ML. It has one major drawback: it is not tail-recursive, which means that it needs a linear amount of space on the stack, and may well run out of space if the operating system places a low limit on the size of the stack.

One can work around this problem by performing concatenation in two passes: that is, in OCaml, by composing `List.rev` and `List.rev_append`. Performing concatenation in one pass and in constant stack space requires breaking the ML type discipline. The authors of the OCaml library “Batteries included” [Bat14] have chosen to do so: they implement concatenation (and other operations on lists) by using an unsafe type cast.

We now show how to write and type-check a tail-recursive version of `append` in Mezzo. The code appears in Figure 3.9. It is written in *destination-passing style* [Lar89], and has constant space overhead. Roughly speaking, the list `xs` is traversed and copied on the fly. When the end of `xs` is reached, the last cell of the copy is made to point to `ys`. We emphasize that, even though mutation is used internally, the goal is to concatenate two immutable lists so as to obtain an immutable list.

Why is this code not well-typed in ML? There are two (related) reasons. One reason is that the code allocates a fresh list cell and initializes its `head` field, but does not immediately initialize its `tail` field. Instead, it makes a recursive call and delegates the task of initializing the `tail` field to the callee. Thus, the type system must allow the gradual initialization of an immutable data structure. The other reason is that, while concatenation is in progress, the partly constructed data structure is not yet a list: it is a list segment. Thus, the type system may have to offer support for reasoning about list segments.

A detailed look at the code The `append` function (line 20) is where concatenation begins. If `xs` is empty, then the concatenation of `xs` and `ys` is `ys` (line 23). Otherwise (line 25), `append` allocates an *unfinished, mutable* cell `dst`. This cell contains the first element of the final list, namely `xs.head`. It is not a valid list cell: its `tail` field contains

the unit value (). It is now up to `appendAux` to finish the work by constructing the concatenation of `xs.tail` and `ys` and by writing the address of that list into `dst.tail`. Once `appendAux` returns, `dst` has become a well-formed list (this is indicated by the postcondition `dst @ list a` on line 8) and is returned by `append`.

The function `appendAux` expects an unfinished, mutable cell `dst` and two lists `xs` and `ys`. Its purpose is to write the concatenation of `xs` and `ys` into `dst.tail`, at which point `dst` can be considered a well-formed list.

If `xs` is `Nil` (line 10), the address `ys` is written to the field `dst.tail` (line 11). Then, `dst`, a mutable block whose tag is `Cell`, is “frozen” by a tag update instruction (line 12) and becomes an immutable block, whose tag is `Cons`. As in §3.1, this instruction has no runtime effect, because these tags have the same runtime representation.

If `xs` is a `Cons` cell (line 13), we allocate a new destination cell `dst'` (line 14), let `dst.tail` point to it (line 15), freeze `dst` (line 16), and repeat the process via a tail-recursive call (line 17).

Look Ma, no segments Operations on (mutable or immutable) lists with constant space overhead are traditionally implemented in an iterative manner, using a `while` loop. For instance, Berdine *et al.*'s formulation of mutable list melding [BCO05a], which is proved correct in separation logic, has a complex loop invariant, involving two list segments, and requires an inductive proof that the concatenation of two list segments is a list segment. In contrast, in our tail-recursive approach, the “loop invariant” is the type of the recursive function `appendAux` (Figure 3.9). This type is quite natural and does not involve list segments.

How do we get away without list segments and without inductive reasoning? The trick is that, even though `appendAux` is tail-recursive, which means that no code is executed after the call by `appendAux` to itself, a *reasoning step* still takes place after the call.

Let us examine lines 14–17 in detail. Upon entering the `Cons` branch, at the start of line 14, the permission for `xs` is `xs @ Cons { head: a; tail: list a }`. As in the earlier version of `append` (Figure 3.8), the type-checker automatically expands it into a conjunction. Here, this requires introducing fresh internal names for the head and tail fields, because the programmer did not provide explicit names for these fields as part of the pattern on line 13. For clarity, we use the names `head` and `tail`. Thus, at the beginning of line 14, the current permission is:

```
dst @ Cell { head: a; tail: () } *
xs @ Cons { head = head; tail = tail } *
head @ a *
tail @ list a *
ys @ list a
```

On line 14, we read `xs.head`. According to the second permission above, this read is permitted, and produces a value whose type is the singleton type `=head`. In other words, it must produce the value `head`. Then, we allocate a new memory block, `dst'`. This yields one new permission, which comes in addition to those above:

```
dst' @ Cell { head = head; tail: () }
```

Although this does not play a key role here, it is worth noting that these permissions imply that the fields `xs.head` and `dst'.head` contain the same value, namely `head`. Besides, we have one (affine) permission for this value, `head @ a`. So, the type-checker “knows” that `xs.head` and `dst'.head` are interchangeable, and that either of them (but not both separately) can be used as a value of type `a`. Thanks to this precise knowledge, we do not need a “borrowing” convention [NBAB12] so as to decide which of `xs.head` or `dst'.head` has type `a`. The idea of recording must-alias information (i.e., equations) via structural permissions and singleton types is taken from Alias Types [SWMoo]. Separation logic [Reyo2] offers analogous expressiveness via points-to assertions and ordinary variables.

The assignment of line 15 and the tag update of line 16 are reflected by updating the structural permission that describes `dst`. Thus, at the beginning of line 17, just before the recursive call, the current permission is:

```
dst @ Cons { head: a; tail = dst' } *
xs @ Cons { head = head; tail = tail } *
head @ a *
tail @ list a *
ys @ list a *
dst' @ Cell { head = head; tail: () }
```

```

1  alias stack a =
2    ref (list a)
3
4  val new [a] (consumes xs: list a) : stack a =
5    newref xs
6
7  val push [a] (consumes xs: list a, s: stack a) : () =
8    s := append (xs, !s)
9
10 val rec work [a, p : perm] (
11   s: stack a,
12   f: (consumes a | s @ stack a * p) -> ()
13 | p) : () =
14   match !s with
15   | Cons { head; tail } ->
16     s := tail;
17     f head;
18     work (s, f)
19   | Nil ->
20     ()
21   end

```

Figure 3.10: A minimal implementation of stacks, with a higher-order iteration function

The call consumes the last four permissions and produces a new permission for `dst'`. Immediately, after the call, the current permission is thus:

```

dst @ Cons { head: a; tail = dst' } *
xs @ Cons { head = head; tail = tail } *
dst' @ list a

```

We have reached the end of the code. However, the type-checker still has to verify that the postcondition of `appendAux` is satisfied. By combining the first and last permissions above, it obtains `dst @ Cons { head: a; tail: list a }`. Then, it folds this permission into `dst @ list a`, thus proving that the postcondition is indeed satisfied: `dst` is now a valid list.

The fact that the structural permission `dst @ Cons { ... }` was framed out during the recursive call, as well as the folding steps that take place after the call, are the key technical mechanisms that obviate the need for list segments and inductive reasoning. In short, the code is tail-recursive, but the manner in which one reasons about it is recursive.

Minamide [Min98] proposes a notion of “data structure with a hole”, or in other words, a segment, and applies it to the problem of concatenating immutable lists. Walker and Morrisett [WM00] offer a tail-recursive version of mutable list concatenation in a low-level typed intermediate language, as opposed to a surface language. The manner in which they avoid reasoning about list segments is analogous to ours. There, because the code is formulated in continuation-passing style, the reasoning step that takes place “after the recursive call” amounts to composing the current continuation with a coercion. Maeda *et al.* [MSY11] study a slightly different approach, also in the setting of a typed intermediate language, where separating implication offers a way of defining list segments.

Our approach could be adapted to an iterative setting by adopting a new proof rule for `while` loops. This is noted independently by Charguéraud [Cha10, §3.3.2] and by Tuerk [Tue10].

A higher-order function

We briefly present a minimal implementation of stacks on top of linked lists. This allows us to show an example of a higher-order function, which is later re-used in the example of graphs and depth-first search (§3.3).

The implementation appears in Figure 3.10. A stack is defined as a mutable reference to a list of elements. The function `new` creates a new stack; the function `push` inserts a list of elements into an existing stack. The latter relies on the list concatenation function (§3.2). The higher-order function `work` abstracts a typical pattern of use of a stack as a work list: as long as the stack is non-empty, extract one element out of it, process this element (possibly causing new elements to be pushed onto the stack), and repeat. This is a loop, expressed as a tail-recursive function. The parameter `s` is the stack; the parameter `f` is a user-provided function that is in charge of processing one element. This function has access to the permission `s @ stack a`, which means that it is allowed to update the stack. The code is polymorphic in the type `a` of the elements. It is also polymorphic in a permission `p` that is threaded through the whole computation: if `f` requires and preserves `p`, then `work` also requires and preserves `p`. One can think of the conjunction `s @ stack a * p` as the loop invariant. The pattern of abstracting over a permission `p` is typical of higher-order functions.

Borrowing elements from containers

In *Mezzo*, a container naturally “owns” its elements, if they have affine type. A list is a typical example of this phenomenon. Indeed, in order to construct a permission of the form `xs @ list t`, one must provide a permission `x @ t` for every element `x` of the list `xs`.

If the type `t` is affine, then one must give up the permission `x @ t` when one inserts `x` into the list. Conversely, when one extracts an element `x` out of the list, one recovers the permission `x @ t`. Other container data structures, such as trees and hash tables, work in the same way.

If the type `t` is duplicable, then the permission `x @ t` does not have an ownership reading. One can duplicate this permission, give away one copy to the container when `x` is inserted into it, and keep one copy so that `x` can still be used independently of the container.

An ownership problem The fact that a container “owns” its elements seems fairly natural as long as one is solely interested in inserting and extracting elements. Yet, a difficulty arises if one wishes to *borrow* an element, that is, to obtain access to it and examine it, without taking it out of the container.

We illustrate this problem with the function `find`, which scans a list `xs` and returns the first element `x` (if there is one) that satisfies a user-provided predicate `pred`. Transliterating the type of this function from ML to *Mezzo*, one might hope that this function admits the following type:

```
val find: [a] (xs: list a, pred: a -> bool) -> option a
```

However, in *Mezzo*, `find` cannot have this type. There is an ownership problem: if a suitable element `x` is found and returned, then this element becomes reachable in two ways, namely through the list `xs` and through the value returned by `find`. Thus, somewhere in the code, the permission `x @ a` must be duplicated. In the absence of any assumption about the type `a`, this is not permitted.

One could assign `find` the following type, where the type parameter `a` is required to be duplicable:

```
val find: [a] (
  xs: list a,
  pred: a -> bool
| duplicable a
) -> option a
```

Naturally, this does not solve the problem. This means that `find` is supported only in the easy case where the elements are shareable. This is an important special case: we explain later on (§3.3) that, provided one is willing to perform dynamic ownership tests, one can always arrange to be in this special case. Nevertheless, it is desirable to offer a solution to the borrowing problem. In the following, we give an overview of two potential solutions, each of which has shortcomings.

A solution in continuation-passing style A simple approach is to give up control. Instead of asking `find` to return the desired element, we provide `find` with a function that describes what we want to do with this element. The signature of `find` thus becomes:


```
val find: [a] (
  xs: list a,
  pred: a -> bool,
  f: (x: a) -> ()
) -> ()
```

Recall that, in Mezzo, a function argument that is *not* annotated with the keyword `consumes` is preserved: that is, the function requires and returns a permission for this argument. Thus, this version of `find` preserves `xs @ list a`. The function `f`, which the user supplies, preserves `x @ a`, where `x` is some element of the list. That is, `f` is allowed to work with this element, but must eventually relinquish the permission to use this element. Note that `f` does *not* have access to the list: it does not receive the permission `xs @ list a`. If it did, the ownership problem would arise again!

It is indeed possible to write a version of `find` that admits the above type. One soon finds out, however, that this type is not expressive enough, as it does not provide any permission to `f` beside `x @ a`. This means that `f` cannot perform any side effect, except possibly on `x`. In order to relax this restriction, one must parameterize `find` over a permission `s`, which is transmitted to (and preserved by) `f`. This is a typical idiom for higher-order functions in Mezzo.

```
val find: [a, s: perm] (
  xs: list a,
  pred: a -> bool,
  f: (x: a | s) -> ()
  | s
) -> ()
```

This approach works, but is awkward for a variety of reasons. First, working in continuation-passing style is unnatural and rigid: elements must be borrowed from the container and returned to the container in a well-parenthesized manner. Second, it is verbose, especially in light of the fact that, in Mezzo, anonymous functions must be explicitly type-annotated. In fact, this version of `find` is just a restricted form of the general-purpose higher-order function for iterating on a list, `iter`. So, one may just as well provide `iter` and omit `find`.

A solution in direct style The root of the problem lies in the fact that the permissions `xs @ list a` and `x @ a` cannot coexist. Thus, the function `find`, if written in a standard style, must consume `xs @ list a` and produce `x @ a`. Of course, there must be a way for the user to signal that they are done working with `x`, at which point they would like to relinquish `x @ a` and recover `xs @ list a`.

Figure 3.11 shows a version of `find` that follows this idea. The function `find` requires the permission `xs @ list a`, which it consumes (line 9). If no suitable element exists in the list, then it returns a unit value, together with the permission `xs @ list a` (line 11). If one exists, then it returns a *focused element* (line 12). This alternative is expressed via the algebraic data type `either` (line 10), which is defined in the standard library as follows:

```
data either a b =
  | Left { contents: a }
  | Right { contents: b }
```

The notion of a focused element appears in our unpublished work on iterators, which pose a similar problem [GPP13]. A focused element (lines 6–7) is a pair of an element `x`, which has type `a`, and a function `w`, a “magic wand” that takes away `x @ a` and produces `xs @ list a` instead. The idea is, when the user is provided with a focused element `(x, w)`, they can work with `x` as long as they like; once they are done, they invoke the function `w`. This function in principle does nothing at runtime: by calling it, the user tells the type-checker that they are done with `x` and would now like to recover the permission to use the list `xs`.

Mezzo does not currently have magic wand as a primitive notion. Instead, we define a magic wand (lines 1–4) as a (runtime) function of no argument and no result, which consumes a permission `pre` and produces a permission `post`. Such a function typically has some internal state, which, conjoined with `pre`, gives rise to `post`. In our definition, this internal state is represented by the existentially quantified permission `ammo`. Within the existential quantifier, written `{ ammo: perm }`, is a package of a function that consumes `pre * ammo` and of one copy of `ammo`.

```

1  alias wand (pre: perm) (post: perm) =
2    {ammo: perm} (
3      (| consumes (pre * ammo)) -> (| post)
4      | ammo)
5
6  alias focused a (post: perm) =
7    (x: a, w: wand (x @ a) post)
8
9  val rec find [a] (consumes xs: list a, pred: a -> bool)
10 : either
11   (| xs @ list a)
12   (focused a (xs @ list a))
13 = match xs with
14 | Nil ->
15   left ()
16 | Cons { head; tail } ->
17   if pred head then begin
18     let w (| consumes (head @ a * tail @ list a))
19       : (| xs @ list a) = () in
20     right (head, w)
21   end
22 else
23   match find (tail, pred) with
24   | Left ->
25     left ()
26   | Right { contents = (x, w) } ->
27     let flex guess: perm in
28     let w' (| consumes (x @ a * head @ a * guess))
29       : (| xs @ list a) = w() in
30     right (x, w')
31   end
32 end

```

Figure 3.11: Borrowing an element from a container in direct style

Because `ammo` is affine, a magic wand can be used at most once: it is a one-shot function. The name “ammo” suggests the image of a gun that needs a special type of ammunition and is supplied with just one cartridge of that type.

Equipped with these (fairly elaborate, but re-usable) definitions, we may explain the definition of `find`.

At line 15, we have reached the end of the list. We return a `left` injection, applied to a unit value. The ownership of the (empty) list is returned to the caller.

At line 20, the element `head` is the one we are looking for. We return a `right` injection, applied to a pair of `head` and a suitable wand `w`. This pair forms a focused element. What is a suitable wand in this case? It should have type `wand (head @ a) (xs @ list a)`. The function `w` defined at lines 18–19 ostensibly has type `(| consumes (head @ a * tail @ list a)) -> (| xs @ list a)`. The type-checker is able to verify that the latter is a subtype of the former; this involves an existential quantifier introduction, taking `tail @ list a` as the witness for `ammo`. The type-checker must also verify that the definition of `w` matches its declared type. This is indeed the case because `w` has access not only to `head @ a * tail @ list a`, but also to the duplicable permission `xs @ Cons { head = head; tail = tail }`. (In Mezzo, a function has access to every *duplicable* permission that exists at its definition site.) By combining these three permissions, one obtains `xs @ list a`, as desired.

At line 25, the desired element has not been found further down: the recursive call to `find` returns `Left`. Even though the code is terse, the reasoning is non-trivial. As we are in the `Left` branch, we have `tail @ list a`. Furthermore, we still possess `head @ a` and `xs @ Cons { head = head; tail = tail }`, which were framed out during the call. The type-checker recombines these permissions and verifies that we have `xs @ list a`, as

demanded in this case by the postcondition of `find`.

At line 26 is the last and most intricate case. The desired element x has not been found further down. The recursive call returns the value x , the permission $x @ a$, and a wand w that we are supposed to use when we are done with x . This wand has type $\text{wand } (x @ a) (\text{tail} @ \text{list } a)$. Using it, we build a new wand w' , which has type $\text{wand } (x @ a) (xs @ \text{list } a)$.

For the benefit of the reader who would like to understand this code fragment in detail, let us say a little more. On line 26, the type-checker automatically expands the type of w , which is an abbreviation for an existential type, and unpacks this existential type. Thus, it views w as a function of type $(\text{! consumes } (x @ a * \text{ammo})) \rightarrow (\text{! tail} @ \text{list } a)$, where `ammo` is a fresh abstract permission; and it considers that one copy of `ammo` is available. At this point, `ammo` is anonymous: there is no way for the programmer to refer to it. Yet, this permission must be listed in the header of the function w' : because w' calls w , it needs the permission `ammo`. We solve this problem via the `let flex` construct on line 27. This construct introduces a flexible variable `guess`. When the type-checker examines the call $w()$ on line 29, it is able to guess that `guess` must be unified with `ammo`, as there is otherwise no way for this call to be well-typed⁵. Finally, the type-checker verifies that the definition of w' matches its explicitly-declared type (the reasoning is analogous to that of lines 18–19) and performs an existential type introduction, automatically picking `head @ a * guess` as the witness, in order to prove that w' has type $\text{wand } (x @ a) (xs @ \text{list } a)$, as desired. Thus, the pair (x, w') has type $\text{focused } a (xs @ \text{list } a)$, as desired. We return it, wrapped in a right injection

Limits of this approach The strength of this approach is that it allows the user to work in direct style, as opposed to continuation-passing style. The fact that Mezzo's type discipline is powerful enough to express the concepts of one-shot function, magic wand, focused element, and to explain what is going on in the `find` function, is good. Nevertheless, we are well aware that this solution is not fully satisfactory, and illustrates some of the limitations of Mezzo, as it stands today.

For one thing, the code is verbose, and requires non-trivial type annotations, in spite of the fact that the type-checker already performs quite a lot of work for us, including automatic elimination and introduction of existential quantifiers. The effort involved in writing this code is well beyond what most programmers would expect to spend.

A related issue is that the definition of `find` contains an eta-expansion that turns the w function into a chain of closures. This makes it all the more difficult for the compiler to efficiently optimize this pattern and figure out that w and w' are the same, and that the entire `match` construct can be replaced by just `find(tail, pred)`.

Figure 3.12 presents an alternative version of the `else` branch where this optimization is made easier by avoiding allocating a new closure entirely. This is done, unfortunately, using an even more advanced type-theoretic mechanism, which is an existential packing. Since the type-checker is unable to figure out on its own that w can be given the desired type, the user can choose to manually instruct the type-checker to hide `head @ a * guess` as an existential, so as to assert $w @ \text{wand } (x @ a) (xs @ \text{list } a)$. The reason why we didn't show this code sample in the first place is that it uses yet another advanced feature of Mezzo. It makes it, however, much clearer that the recursive call is a tail call.

Another criticism is that we encode a magic wand as a runtime function, even though this function has no runtime effect. Ideally, there should be a way of declaring that a function is “ghost”. The system would check that this function has no runtime effect (including non-termination). This would eliminate the need for allocating a pair (x, w) at runtime.

⁵This style of binding an existential variable may seem unusual. However, there are various reasons why a `let-unpack` construct would not work.

- For reasons that will become apparent in Chapter 9, we must eagerly open existential quantifiers. This means that by the time the programmer reaches line 27, the existential variable has been opened already, and there is no opportunity for a manual `let-unpack guess, w = w in ...`
- In Chapter 7, we explain how function types are translated and many more existential quantifiers are actually added onto the function type. This means that we would need an extra mechanism to ensure the user can only refer to user-introduced existential quantifiers, not translation-induced existential variables.
- Several function types may be available for the same variable, and the user has no way in the surface language to “pick” one. The `let flex` construct allows to let the type-checking engine pick a suitable instantiation.

This is, granted, not completely satisfactory, and deserves further improvements. The reader should keep in mind, though, that this code is very much advanced Mezzo, and that we do not expect the casual user to actually deal with this somehow internal details.

```

else begin
  let r = find (tail, pred) in
  match r with
  | Left ->
    r
  | Right { contents = (x, w) } ->
    let flex guess: perm in
    pack w @ wand (x @ a) (xs @ list a) witness (head @ a * guess);
    r
  end
end
end
end

```

Figure 3.12: Alternative version of the else branch from Figure 3.11

Limits of both approaches In either approach, when one borrows an element x from a list xs , one gains the permission $x @ a$, but loses $xs @ list\ a$. This means that at most one element at a time can be borrowed from a container.

In a way, this restriction makes sense. One definitely cannot hope to borrow a single element x twice, as that would entail duplicating the affine permission $x @ a$. Thus, in order to borrow two elements x and y from a single container, one must somehow prove that x and y are distinct. Such a proof is likely to be beyond the scope of a type system; it may well require a full-fledged program logic.

At this point, the picture may seem quite bleak. One thing to keep in mind, though, is that the whole problem vanishes when the type a is duplicable. This brings us naturally to the next section. We propose a mechanism, adoption and abandon, which can be viewed as a way of converting between an affine type a and a universal duplicable type, `dynamic`. One can then use a container whose elements have type `dynamic`, and look up multiple elements in this container, without restriction. Naturally, the conversion from type `dynamic` back to type a involves a runtime check, so that attempting to borrow a single element twice causes a runtime failure. The proof obligation $x \neq y$ has been deferred from compile time to runtime.

3.3 Breaking out: arbitrary aliasing of mutable data structures

The type-theoretic discipline that we have presented up to this point allows constructing a composite permission out of several permissions and (conversely) breaking a composite permission into several components. For instance, a permission for a tree is interconvertible with a conjunction of separate permissions for the root record and for the sub-trees (§3.2). Thus, every tree-shaped data structure can be described in Mezzo by an algebraic data type.

There are two main limitations to the expressive power of this discipline.

First, because we adopt an inductive (as opposed to co-inductive) interpretation of algebraic data types, a permission cannot be a component of itself. In other words, it cannot be used in its own construction. This holds of both duplicable and affine permissions. Thus, every algebraic data type describes a family of *acyclic* data structures. The permission $xs @ list\ int$, for instance, means that xs is a finite list of integers. (In this sense, Mezzo differs from OCaml, which allows constructing a cyclic list⁶.) This choice is intentional: we believe that it is most often desirable to ensure the absence of cycles in an algebraic data structure.

Second, an affine permission cannot serve as a component in the construction of two separate composite permissions. Because every mutable memory block (and, more generally, every data structure that contains such a block) is described by an affine permission, this means that *mutable data structures cannot be shared*. Put another way, this discipline effectively imposes an *ownership hierarchy* on the mutable part of the heap.

When one wishes to describe a data structure that involves a cycle in the heap or the sharing of a mutable sub-structure, one must work around the restrictions described above. This requires extra machinery.

⁶ In OCaml, one can write `let rec x = 0 :: 1 :: x`, that is, one can define *recursive values*.

```

1  data mutable node a =
2      Node {
3          value    : a;
4          visited  : bool;
5          neighbors: list (node a);
6      }
7
8  val _ : node int =
9      let n = Node {
10         value    = 10;
11         visited  = false;
12         neighbors = ();
13     } in
14     let ns = Cons { head = n; tail = Nil } in
15     n.neighbors <- ns;
16     n

```

Figure 3.13: A failed attempt to construct a cyclic graph

Illustration In order to illustrate the problem, let us define a naïve type of graphs and attempt to construct the simplest possible cyclic graph, where a single node points to itself.

The definition of the type node is straightforward (Figure 3.13, lines 1–6). Every node stores a value of type `a`, where the type variable `a` is a parameter of the definition; a Boolean flag, which allows this node to be marked during a graph traversal; and a list of successor nodes. The type `node` is declared mutable: it is easy to think of applications where all three fields must be writable.

Next (lines 8–16), we allocate one node `n`, set its `neighbors` field to a singleton list of just `n` itself, and claim (via the type annotation on line 8) that, at the end of this construction, `n` has type `node int`. This code is ill-typed, and is rejected by the type-checker. Perhaps surprisingly, the type error does not lie at line 15, where a cycle in the heap is constructed. Indeed, at the end of this line, the heap is described by the following permission:

```

n @ Node {
  value    : int;
  visited  : bool;
  neighbors = ns;
} *
ns @ Cons { head = n; tail: Nil }

```

This illustrates the fact that a cycle of statically known length can be described in terms of structural permissions and singleton types. The type error lies on line 16, where (due to the type annotation on line 8) the type-checker must verify that the above permission entails `n @ node int`. This entailment is invalid because it violates the first limitation that was discussed earlier: since the algebraic data type `node` is interpreted inductively, a node cannot participate in its own construction. Furthermore, even if a co-inductive interpretation was adopted, this entailment would still be invalid, as it would then violate the second limitation that was discussed above: since `n @ Node { ... }` is affine, it cannot be used to *separately* justify that `n` is a node and `ns` is a list of nodes.

To sum up, the type `node` at lines 1–6 is not a type of possibly cyclic graphs, as one might have naïvely imagined. It is in fact a type of trees, where each tree is composed of a root node and a list of disjoint sub-trees.

A solution The problem with this naïve approach stems from the fact that types have an ownership reading. Saying that `neighbors` is a list of nodes amounts to claiming that every node owns its successors, which does not make sense, because ownership must be a hierarchy.

In order to solve this problem, we must allow a node to point to a successor without implying that there is an ownership relation between them. “Who” then should own the nodes? A natural answer is, the set of all nodes should be owned as a whole by a single distinguished object, say, the “graph” object.

```

data mutable node a =
  Node {
    content : a;
    visited : bool;
    neighbors: list dynamic;
  }

data mutable graph a =
  Graph {
    roots : list dynamic;
  } adopts node a

val g : graph int =
  let n = Node {
    content = 10;
    visited = false;
    neighbors = ();
  } in
  let ns = Cons { head = n; tail = Nil } in
  n.neighbors <- ns;
  assert n @ node int * ns @ list dynamic;
  let g : graph int = Graph { roots = ns } in
  give n to g;
  g

val dfs [a] (g: graph a, f: a -> ()) : () =
  let s = stack::new g.roots in
  stack::work (s, fun (n: dynamic
    | g @ graph a * s @ stack dynamic) : () =
    take n from g;
    if not n.visited then begin
      n.visited <- true;
      f n.content;
      stack::push (n.neighbors, s)
    end;
    give n to g
  )

```

Figure 3.14: Graphs, a cyclic graph, and depth-first search, using adoption and abandon

Figure 3.14 presents a corrected definition of graphs, and shows how to build the cyclic graph of one node. It also contains code for an iterative version of depth-first search, using an explicit stack. Let us explain this example step by step.

The type `dynamic` The only change in the definition of `node` `a` is that the `neighbors` field now has type `list dynamic` (line 8).

The meaning of `n @ dynamic` is that `n` is a valid address in the heap, i.e., it is the address of a memory block. When one allocates a new memory block, say via `let n = Node { ... } in ...`, one obtains not only a structural permission `n @ Node { ... }`, but also `n @ dynamic`. Although the former is affine (because `Node` refers to a mutable algebraic data type), the latter is duplicable. Intuitively, it is sound for the type `dynamic` to be considered duplicable because the knowledge that `n` is a valid address can never be invalidated, hence can be freely shared. However, the permission `n @ dynamic` does *not* allow reading or writing at this address. In fact, it does not even describe the type of the memory block that is found there—and it cannot, since this block is owned by “someone else” and its type could change with time.

Because it is duplicable, the type `dynamic` does not have an ownership reading. The fact that `neighbors` has type `list dynamic` does not imply that a node owns its successors; it means only that `neighbors` is a list of heap addresses.

Constructing a cyclic graph As an example, we construct a node that points to itself (lines 17–24). The construction is the same as in Figure 3.13. This time, it is well-typed, though. Because we have `n @ dynamic`, we can establish `ns @ list dynamic`, and, therefore, `n @ node int`. Furthermore, since `ns @ list dynamic` is duplicable, it is not consumed in the process. The (redundant) static assertion on line 24 shows that the desired permissions for `n` and `ns` co-exist.

The type `graph a` (lines 11–14) defines the structure of a “graph” object. This object contains a list of so-called root nodes. Like `neighbors`, this list has type `list dynamic`. Furthermore, the `adopts` clause on line 14 declares that an object of type `graph a` *adopts* a number of objects of type `node a`. This is a way of saying that the graph “owns” its nodes. Thus, an object of type `graph int` is an *adopter*, whose *adoptees* are objects of type `node int`. The set of its adoptees changes with time, as there are two instructions, `give` and `take`, for establishing or revoking an adoptee-adopter relationship.

The `give` and `take` instructions The runtime effect of the *adoption* instruction `give n to g` (line 26) is that the node `n` becomes a new adoptee of the graph `g`. At the beginning of this line, the permissions `n @ node int` and `g @ graph int` are available. Together, they justify the instruction `give n to g`. (The type-checker verifies that the type of `g` has an `adopts` clause and that the type of `n` is consistent with this clause.) After the `give` instruction, at the end of line 26, the permission `n @ node int` has been consumed, while `g @ graph int` remains. A *transfer of ownership* has taken place: whereas the node `n` was “owned by this thread”, so to speak, it is now “owned by `g`”. The permission `g @ graph int` should be interpreted intuitively as a proof of ownership of the object `g` (which has type `graph int`) *and* of its adoptees (each of which has type `node int`). It can be thought of as a conjunction of a permission for just the memory block `g` and a permission for the group of `g`’s adoptees; in fact, in our formalization ([BPP14b]), these permissions are explicitly distinguished.

Although `g @ graph int` implies the ownership of all of the adoptees of `g`, it does not indicate who these adoptees are: the type system does not statically keep track of the relation between adopters and adoptees. After the `give` instruction at line 26, for instance, the system does not know that `n` is adopted by `g`. If one wishes to assert that this is indeed the case, one can use the *abandon* instruction, `take n from g`. The runtime effect of this instruction is to check that `n` is indeed an adoptee of `g` (if that is not the case, the instruction fails) and to revoke this fact. After the instruction, the node `n` is no longer an adoptee of `g`; it is unadopted again. From the type-checker’s point of view, the instruction `take n from g` requires the permissions `n @ dynamic`, which proves that `n` is the address of a valid block, and `g @ graph int`, which proves that `g` is an adopter and indicates that its adoptees have type `node int`. It preserves these permissions and (if successful) produces `n @ node int`. This is a transfer of ownership in the reverse direction: the ownership of `n` is taken away from `g` and transferred back to “this thread”.

Conceptual model The adopter owns its adoptees. Conceptually, one can think of the adopter as an object which maintains a list of the elements that it owns, inserting new elements as `give` operations take place, and

taking them out as `take` operations are performed. More precisely, if the adopter “adopts `t`”, then one can think of it as owning a “list `t`” of its adoptees.

This conceptual view of adoption and abandon can be implemented in Mezzo as a library. The library is shown and commented in §4.2. We chose, however, to use a more optimized representation which performs the `give` and `take` operations in constant time, at the cost of an extra memory word.

Runtime model We maintain a pointer from every adoptee to its adopter. Within every object, there is a hidden `adopter` field, which contains a pointer to the object’s current adopter, if it has one, and `null` otherwise. This information is updated when an object is adopted or abandoned. In terms of space, the cost of this design decision is one field per object⁷.

The runtime effect of the instruction `give n to g` is to write the address `g` to the field `n.adopter`. The static discipline guarantees that this field exists and that its value, prior to adoption, is `null`.

The runtime effect of the instruction `take n from g` is to check that the field `n.adopter` contains the address `g` and to write `null` into this field. If this check fails, the execution of the program is aborted. We also offer an expression form, `g adopts n`, which tests whether `n.adopter` is `g` and produces a Boolean result.

Illustration We illustrate the use of adoption and abandon with the example of depth-first search (Figure 3.14, lines 29–40). The frontier (i.e., the set of nodes that must be examined next) is represented as a stack; we rely on the `stack` module of Figure 3.10. The stack `s` has type `stack dynamic`. We know (but the type-checker does not) that the elements of the stack are nodes, and are adoptees of `g`.

The function `dfs` initializes the stack (line 30) and enters a loop, encoded as a call to the higher-order function `stack::work`. At each iteration, an element `n` is taken out of the stack; it has type `dynamic` (line 31). Thus, the type-checker does not know a priori that `n` is a node. The `take` instruction (line 33) recovers this information. It is justified by the permissions `n @ dynamic` and `g @ graph int` and (if successful) produces `n @ node int`. This proves that `n` is indeed a node, which we own, and justifies the read and write accesses to this node that appear at lines 34–37. Once we are done with `n`, we return it to the graph via a `give` instruction (line 39).

There are various mistakes that the programmer could make in this code and that the type-checker would not catch. Forgetting the final `give` would be one such mistake; it would lead to a runtime failure at a later `take` instruction, typically on line 33. In order to diminish the likelihood of this particular mistake, we propose taking `n from g begin ... end` as syntactic sugar for a well-parenthesized use of `take` and `give`.

Discussion Because adoption and abandon are based on a runtime test, they are simple and flexible. If one wished to avoid this runtime test, one would probably end up turning it into a static proof obligation. The proof, however, may be far from trivial, in which case the programmer would have to explicitly state subtle logical properties of the code, and the system would have to offer sufficient logical power for these statements to be expressible. The dynamic discipline of adoption and abandon avoids this difficulty, and meshes well with the static discipline of permissions. We believe that we have a clear story for the user: “when you need to share mutable data structures, use adoption and abandon”.

Adoption and abandon are a very flexible mechanism, but also a dangerous one. Because `abandon` involves a dynamic check, it can cause a fatal failure at runtime. In principle, if the programmer knows what they are doing, this should never occur. There is some danger, but that is the price to pay for a simpler static discipline. After all, the danger is effectively less than in ML or Java, where a programming error that creates an undesired alias goes completely undetected—until the program misbehaves in one way or another.

To the best of our knowledge, adoption and abandon are new. Naturally, the concept of group, or region, has received sustained interest in the literature [CWM99, DF01, FD02, SHM⁺06]. Regions are usually viewed either as a dynamic memory management mechanism or as a purely static concept. Adoption and abandon, on the other hand, offer a dynamic ownership control mechanism, which complements our static permission discipline.

One could see our contribution as twofold. First, we identify the name of the object with the name of the dynamic region. This makes the story easier to explain to the user. Instead of introducing a notion of region

⁷It would be possible to lessen this cost by letting the programmer decide, for each data type, whether the members of this type should be adoptable (hence, should contain an `adopter` field) or not. Furthermore, one could note that there should never be a need for an immutable object to be adoptable. One should then restrict the `tag update` instruction so as to forbid going from an adoptable data type to a non-adoptable one, or vice-versa. For the moment, for the sake of simplicity, we consider only the uniform model where every object has an `adopter` field.

along with the ownership of that region, we merely talk about transfer of ownership, saying that the ownership is transferred from the currently executing thread to another object. We also only ever mention one name, thus simplifying things. Our second contribution is the optimized representation using the hidden field. To the best of our knowledge, this is a novel mechanism.

One might wonder why the type `dynamic` is so uninformative: it gives no clue as to the type of the adoptee or the identity of the adopter. Would it be possible to parameterize it so as to carry either information? The short answer is negative. The type `dynamic` is duplicable, so the information that it conveys should be stable (i.e., forever valid). However, the type of the adoptee, or the identity of the adopter, may change with time, through a combination of strong updates and `give` and `take` instructions. Thus, it would not make sense for `dynamic` to carry more information.

That said, we believe that adoption and abandon will often be used according to certain restricted protocols, for which more information is stable, hence can be reflected at the type level. For instance, in the bag implementation, a cell only ever has one adopter, namely a specific bag `b`. In that case, one could hope to work with a parameterized type `dynamic' b`, whose meaning would be “either this object is currently not adopted, or it is adopted by `b`”. Ideally, `dynamic'` would be defined on top of `dynamic` in a library module, and its use would lessen the risk of confusion.

The new presentation of adoption/abandon (§8.8) addresses this criticism and allows the user to define, in a library, restricted `give` and `take` primitives, which provide more safety in exchange for less flexibility: an object can be declared to be “dedicated” to only one adopter for the entire course of its lifetime. We believe, from our experience writing Mezzo programs, that this pattern covers the majority of cases, and that once the implementation is updated to use the new adoption/abandon, users will benefit from the added safety.

Another frequently asked question is why, once an exclusive permission goes out of scope, the object cannot be deallocated. The reason is, the user might still have a pointer to it via the `dynamic` type, meaning that we would need a runtime system that supports dangling pointers. This seems hard to accommodate with the presence of a garbage-collector.

Future work The current presentation of adoption/abandon has been refined and is more powerful in the formalization of Mezzo. The implementation is lagging behind and still features the “old” adoption/abandon, which we just described. §8.8 discusses this in greater detail.

A drawback of the current mechanism is the cost of bulk ownership transfer. Consider, for instance, the case where the user wishes to merge two graphs `g1` and `g2` together. As of now, the user cannot just do `g1.roots <- append (g1.roots, g2.roots);` they need to take each node from `g2`, and give it to `g1`. A way to solve this problem is to add yet another hidden field for each object. The field would be used to implement a union-find mechanism on regions. An address in the field would stand for a link, while a null pointer would indicate that the representative of the equivalence class has been found. Path compression can be performed for each adoptee when a lookup is performed. Merging two objects `g1` and `g2` is a constant-time operation, where the address of `g1` is written in the new hidden field of `g2`. A potential drawback is that this mechanism prevents the collection of `g2` until all path compressions have been performed.

Having two hidden fields would be a significant runtime penalty. The current implementation already uses one extra word per object: one may wish to allow the user to decide whether they want to use an extra word or not. An idea we have thus explored is to have two modes (§8.4): either “fat exclusive” or “slim exclusive”. Only the former would have the hidden field. This slightly complicates the formalization, especially for things related to mode computation. This also exposes some implementation details in the semantics of the language: tag updates become possible only between objects with the same “fitness” (slim or fat), due to the size of the underlying blocks.

A last drawback that has bitten us is the restriction that an adopter must be exclusive. This is required for soundness; in the concurrent case, however, this means that if multiple threads compete for taking objects, they must not only use the adoption/abandon mechanism, but also a lock for acquiring ownership of the adopter first. It would be possible to allow adopters to be duplicable; this would require, however, an implementation of `take` based on a compare-and-swap primitive, along with a new `try_take` operation.

More generally, adoption/abandon is only just one means of escaping the restrictions of strict aliasing. We offer another such means later on (§4.1), but experience from other projects such as Cyclone [SHM⁺06] seems to suggest that a wide variety of escape hatches is needed. The difficulty of expressing the borrowing pattern seems to indicate that we may need a primitive mechanism for borrowing, similar to what has been done in Rust. In Rust, one may borrow the contents of a uniquely-owned box. If `&x` borrows a field from a uniquely-owned memory

block `y`, then `y` is statically invalidated as long as `&x` is alive. A similar mechanism may be useful for Mezzo: one may return a “borrowed pointer” into a list, which renders the list unusable unless the borrowed pointer is discarded.

On a related note, fractional permissions also seem quite popular, and Mezzo currently does not offer a way to share a mutable piece of data in a read-only manner (the “`const`” keyword). Fractional permissions would be a way to solve this aliasing problem.

4. Bits from the standard library

The present chapter presents a few excerpts from the Mezzo standard library. These code snippets are somewhat advanced, and sometimes, rather technical. I believe, though, that they illustrate faithfully the expressive power of Mezzo.

4.1 Nesting

The main drawback of our adoption/abandon mechanism is that it relies on run-time tests, meaning that there is a performance penalty when executing the program. Other means of handling arbitrary aliasing over mutable data structures have been proposed. One of them is Boyland's nesting [Boy10].

Unlike adoption/abandon, nesting is a purely static mechanism. In essence, it allows to transfer ownership of a permission to another permission.

We *axiomatize* nesting in Mezzo; the result is slightly different in that it allows to transfer ownership of a permission to another object. The technical ingredients that allow us to axiomatize a theory are:

- abstract permissions, declared at top-level with the `abstract` keyword; abstract permissions, just like regu-

```
1 abstract nests (x : value) (p : perm) : perm
2 fact duplicable (nests x p)
3
4 val nest: [x : value, p : perm, a]
5   exclusive a => (| x @ a * consumes p) -> (| nests x p)
6
7 abstract punched (a : type) (p : perm) : type
8
9 val focus: [x : value, p : perm, a]
10  exclusive a => (| consumes x @ a * nests x p) -> (| x @ punched a p * p)
11
12 val defocus: [x : value, p : perm, a]
13  (| consumes (x @ punched a p * p)) -> (| x @ a)
14
15 val nest_punched: [x : value, p : perm, a, q : perm]
16  (| x @ punched a q * consumes p) -> (| nests x p)
```

Figure 4.1: Axiomatization of nesting in Mezzo

lar data types, may be parameterized and one may reveal facts about them;

- our module mechanism, which allows us to offer the axioms as an interface (Figure 4.1), with an implementation that performs no operation at run-time.

Axiomatization

Let us see how nesting works and is axiomatized in Mezzo.

The key type is `nests x p` (lines 1–2), which asserts that the object `x` (a program variable at kind `value`) owns the permission `p` (an abstract permission variable at kind `perm`). This information, just like the dynamic type, is duplicable. We call it a *nesting witness*.

Just like one can use our `give` operation, one can, with nesting, transfer ownership of a permission `p` to a variable `x` through the `nest` operation (line 4). For this operation to work, the owner `x` must be exclusive, which, again, is reminiscent of adoption/abandon. The permission `p` is lost by the caller, who gains in exchange a witness that `p` is now owned by `x`.

Aside: the exclusive requirement We have seen the `duplicable` a requirement already; the code uses a new requirement, namely `exclusive`. Having `x @ t` with `t exclusive` guarantees that we have *unique* ownership of the memory block at address `x`. Having a mere affine `t` does *not* guarantee unique ownership: one could, for instance, turn a `duplicable` type into an affine type via type abstraction. A corollary is that having `x @ t * x @ u` is impossible (i.e. it implies \perp) and denotes an unreachable piece of code.

All mutable types have a unique owner in Mezzo: they are thus exclusive. We have seen the type `cell` a which is defined as `mutable` in Figure 3.9: such a type satisfies the exclusive requirement.

Since this is purely static, there is no way one can permanently regain ownership of `p`. Indeed, `nests x p` is `duplicable` (line 2). Allowing one to permanently regain `p` means that the operation could be carried out several times, using several copies of `nests x p`.

What nesting provides, instead, is a pair of `focus` (line 9) and `defocus` (line 12) functions.

The `focus` function *temporarily* regains ownership of a permission `p` currently owned by `x`, using the adequate nesting witness. The caller does regain `p`; however, the exclusive permission `x @ a` is consumed. What the caller gains instead is `x @ punched a p`, which embodies the fact that `x @ a` is no longer usable, because we *took* `p` from `x`. In other words, we “punched a hole into `x`”.

Changing the type of `x` effectively prevents further calls to `focus`, hence preserving soundness.

(Let us imagine for a moment we wished to allow calling `focus` on a punched object. For this to be sound, the user would need to prove that they are punching a different permission than the one already punched out. In our adoption/abandon mechanism, this proof obligation is fulfilled via the run-time test. If the test succeeds, we learn that `x` and `x'` are physically distinct, meaning that the caller can safely be granted `x @ t` and `x' @ t`. Lacking any similar facility for nesting, the user must not be allowed to call `focus` on a punched object.)

After a `focus` operation, the user is free to use `p` as they wish; should they need to recover the permission for `x`, they can call `defocus` (line 12), which trades `x @ punched a p * p` for the original `x @ a`. We “plug the punched hole”.

One should note that it is possible to nest further objects while `x` is punched via `nest_punched` (line 15).



Having two different function types is unsatisfactory; it turns out that we can attach multiple function types to the same value, using multiple permissions. This amounts to expressing function overloading in Mezzo.

One can, for instance, change the signature of `nest`, and replace `nest_punched` with:

```
val nest_: (| nest @ [x : value, p : perm, a, q : perm] (| x @ punched a q * consumes p)
  -> (| nests x p))
```

The net effect is that, upon processing this signature item, the type-checker adds `nest_ @ ()` to the current permission, along with the new permission `nest @ ...`. One could conceivably allow in the syntax `val _` and get rid of the (useless) `nest_ name`.

This poses inference challenges, however. In the case of nesting, one frequently instantiates polymorphic quantifiers manually, as the type-checker has no way of figuring out which permission should nest. The result is that, when confronted with `nest [x, p]`, the type-checker must return two instantiated function types, suitable for

```

1  abstract region
2
3  abstract element (ρ: value) a
4  fact duplicable (element ρ a)
5
6  val new: () -> region
7
8  val create: [a] duplicable a => (ρ: region, x: a) -> element ρ a
9
10 val repr: [a] duplicable a => (ρ: region, e: element ρ a) -> element ρ a
11
12 val unify: [a] duplicable a => (ρ: region, x: element ρ a, y: element ρ a) -> ()
13
14 val find: [a] duplicable a => (ρ: region, x: element ρ a) -> a

```

Figure 4.2: A union-find data structure implemented with nesting (interface)

performing a function call. While the type-checker backtracks at the function call level, it does not backtrack at the type application level. For that reason, we left the extra `nest_punched` value in the interface.

Limitations

Nesting only works whenever elements can be given up forever. Implementing, for instance, a bag interface, where one puts and retrieves elements, is not possible with nesting.

Also, nesting cannot express situations where two elements need to be taken out at the same time, which somehow limits its expressiveness, or requires the user to jump through hoops. The example below illustrates this.

Sample usage

We demonstrate the implementation of a union-find data structure using nesting, whose interface is shown in Figure 4.2. The data structure offers equivalence classes (the `element` type); the client may create a new class distinct from all the other ones (using `create`); the client may unify two classes (using `unify`). Moreover, each class is associated with a *representative*; the client may ask for the representative associated to a class (using `repr`); determining whether two classes are equivalent thus amounts to checking that they have the same representative.

The data structure is imperative: the `unify` operation operates in-place, that is, performs an internal mutation. The equivalence classes thus contain mutable data, meaning that they have a unique owner. The client, however, wishes to keep several pointers onto the same class. This is a typical case where we need unrestricted aliasing of mutable data. In Mezzo, this requires the use of an escape mechanism; I chose to use nesting.

Our implementation is shown in Figure 4.3. It performs only one of the two classical optimizations, namely path compression (the other classical optimization does not pose a technical difficulty).

We declare the constructor of regions (line 1); this is the exclusive object where classes will nest. There is one region per instance of the union-find data structure, meaning that creating a fresh union-find creates a new region where classes will live. The region embodies the ownership of the classes, meaning it is up to the caller to properly thread the region object in a linear manner.

Inhabitants of the region are either `Link` or `Root` constructors (line 3): these are the objects that the client manipulates. Two objects are equivalent if following the `link` fields leads onto the same `Root`. Merging two objects involves finding their respective `Roots` and turning one of them into a `Link` to the other one.

I mentioned earlier that the client wishes to keep multiple pointers into the same equivalence class: this means, concretely, that the client wishes to keep pointers to either `Link` or `Root` objects; these objects are, however, mutable, which is why we will use nesting to handle aliasing of mutable data. The problem does not just occur within the client code: internally, multiple `Link` objects may point to the same `Root`, meaning that even within the implementation, mutable data is aliased. The use of an escape hatch is thus doubly mandatory.

4. BITS FROM THE STANDARD LIBRARY

```
1 data mutable region = UnionFind
2
3 data mutable inhabitant (ρ: value) a =
4   | Link { link: element ρ a }
5   | Root { descr: a }
6
7 and element (ρ: value) a =
8   | Element { element:: unknown | nests ρ (element @ inhabitant ρ a) }
9
10 val new (): region =
11   UnionFind
12
13 val create [a] duplicable a => (ρ: region, x: a): element ρ a =
14   let e = Root { descr = x } in
15   nest [ρ, (e @ inhabitant ρ a)] ();
16   Element { element = e }
17
18 val rec repr [a] duplicable a => (ρ: region, e: element ρ a): element ρ a =
19   let elt = e.element in
20   focus [ρ] ();
21   match elt with
22   | Link { link } ->
23     defocus [ρ] ();
24     let r = repr (ρ, link) in
25     focus [ρ, (elt @ inhabitant ρ a)] ();
26     match elt with
27     | Link ->
28       elt.link <- r;
29       defocus [ρ] ();
30       r
31     | Root -> fail
32     end
33   | Root ->
34     defocus [ρ] ();
35     e
36   end
37
38 val unify [a] duplicable a => (ρ: region, x: element ρ a, y: element ρ a): () =
39   let x = repr (ρ, x) and y = repr (ρ, y) in
40   if x == y then
41     ()
42   else begin
43     let elt_x = x.element in
44     focus [ρ, (elt_x @ inhabitant ρ a)] ();
45     match elt_x with
46     | Link -> fail
47     | Root ->
48       tag of elt_x <- Link;
49       elt_x.link <- y
50     end;
51     defocus [ρ] ();
52   end
53
54 val find [a] duplicable a => (ρ: region, x: element ρ a): a =
55   let r = repr (ρ, x) in
56   let elt = r.element in
57   focus [ρ, (elt @ inhabitant ρ a)] ();
58   let v =
59     match elt with
60     | Root -> elt.descr
61     | Link -> fail
62     end
63   in
64   defocus [ρ] ();
65   v
```

Figure 4.3: A union-find data structure implemented with nesting

We nest inhabitants inside the region, so as to get duplicable instances of the `element` type (line 8). The client of the module thus manipulates objects of type `element` which, being duplicable, allow them to hold multiple pointers into the same equivalence class.



The declaration of the `element` type uses a binding field `element ::`. The double colon signals that the field name also introduces a fresh binding, which is bound for the whole record.

Using a data type for `element` seems overkill, as a simple type alias could, in theory, work. It turns out that Mezzo does not allow recursively-defined alias declarations. There is thus no support for mutual recursion between alias definitions and data definitions.

Creating a new region (line 10) and creating a new equivalence class (line 13) are straightforward. Let us just notice that we demand the data stored in each equivalence class to be of a duplicable type `a`. In case the data are not duplicable, we leave it up to the user to use an appropriate abstraction and figure out who owns the elements. We also use at line 15 a type application. Lacking any annotation, the type-checker is faced with too many choices for the actual permission to nest into the region.

Finding the representative of an equivalence class (line 18) requires walking up the chain of `Link` inhabitants, until we find the `Root` one. The function leaves the region `ρ` intact (i.e, not punched): the pre- and post-condition of `repr` mention `ρ` with type `region`, while the `element` returned is nested. A consequence for the caller is that they have to call `focus`.



One could consider a symmetrical signature where `repr` does not return the region intact, but punched. The return value would then be of type `Root { descr: a }`, hence requiring the client to call `defocus` (instead of `focus`) once they're done with the inhabitant. I explain later why this is not a good design.

Let me now explain the `repr` function in greater detail. The first step consists in focusing the `element` (line 20) to obtain the “original” permission for it. This allows us to figure whether it’s a `Link` or a `Root`. The latter case is easier: we merely need to `defocus` the element and return it. This satisfies the function’s post-condition, which is that the region `ρ` is returned in its original state.

In case the element is *not* a root (line 22), we need to call ourselves recursively to find the root. The recursive call to `repr` will not work with a punched `ρ`; we thus `defocus elt` (line 23), meaning we lose the permission `elt @ inhabitant ρ a`. We retain, however, the permission `link @ element ρ a` which is duplicable. This allows us to call `repr` recursively on line 24.

Once the recursive call returns, we need to perform path compression, that is, write `r` into the `link` field of `elt`. We no longer know, however, that `elt` is a `Link`: indeed, for the recursive call to take place, we had to give up our permission `elt @ Link { ... }`. We thus need to discriminate *again* on `elt`. This makes sense: the recursive call could have, potentially, mutated `elt`. We do know, however, that our data structure does not contain cycles: we have the (unchecked) invariant that `elt` is not reachable *via* the `link` field. This means that the recursive call could not modify `elt`, hence that it is still a `Link` and that the `Root` case is impossible (line 31). We perform path compression then finally `defocus` (line 29) so as to return an `element` along with a region.

The `repr` function is tricky: the pair of `focus` and `defocus` operations is not syntactically well-parenthesized. It is, however, operationally well-parenthesized. Conversely, the `unify` and `find` functions are simpler and contain well-parenthesized uses of `focus` and `defocus`.

In the alternative design I mentioned earlier, `repr` would have the following type:

```
val rec repr: [a] duplicable a =>
  (ρ: region, e: element ρ a) ->
  (elt: Root { descr: a } | ρ @ punched ρ (elt @ Root { descr: a }))
```

This signature has two drawbacks. First, it is up to the caller to call `defocus` once they are done with `elt`; this usage of `focus` and `defocus` is ill-parenthesized *across* the client/library interface and requires us to expose nesting to the client. Second, it would prevent the two successive calls to `repr` in `unify` from succeeding, as one would have to insert a call to `defocus` in-between.

The interface of the module is shown in Figure 4.2. One thing to note is that we do not reveal our internal usage of nesting to the client. Another thing to note is that a similar implementation can be achieved using our

adoption/abandon discipline, and that the interface would not change. We thus restrict the use of `dynamic` to the library, and do not expose this information to the client thanks to an abstract type. This thus alleviates the risks associated to the seemingly weak nature of the `dynamic` type.

On to static regions

Nesting can be used as the basis for a library of regions, whose interface matches that of Charguéraud and Pottier [CP08]. We do not show the implementation, which is available in the standard library distributed along with the Mezzo type-checker. It allows re-implementing the union-find algorithm in an even more concise manner.

4.2 Adoption/abandon as a library

We mentioned earlier (§3.3) that the adoption/abandon feature of Mezzo can be defined in a library. Indeed, one can conceptually think of the adopter as possessing a list of its adoptees; `take`'ing amounts to removing an element from the list while `give`'ing amounts to adding an element to the list. The code snippet from Figure 4.4 does precisely that.

An adopter *is* a list of its adoptees (line 4). The list is stored in a reference so as to be mutable. The user has to allocate a new adopter (line 11) and pay the price of an extra indirection; there is no support here for the built-in “adopts” clause.

The `dynamic_` type is implemented using `unknown` (line 6), which is our top type at kind type: the permission `x @ unknown` is always available for any `x`. The user can obtain `x @ dynamic_` at any time, for any `x`, using the `dynamic_appears` function (line 8). This has a run-time cost; the function could conceivably be marked as ghost if we had such a mechanism. Again, lacking any built-in support, we lose the subsumption rule that allows one to obtain `x @ dynamic` out of thin air (`DYNAMICAPPEARS`, §8.2).

While the library can be used with duplicable elements, it makes little sense to do so; we could, should we wish to, restrict the usage of the library by adding `exclusive t =>` in front of all the functions.

The `give_` function (line 14) is as simple as one could expect. The signature of the function corresponds to what the regular `give` operation does.

The `take_` function has been made fallible through the use of a rich boolean (§4.4). One could define a non-fallible version of it by using the `fail` keyword. This function is certainly the most interesting in the library. Its post-condition specifies that, should the operation be successful, the caller gains `child @ t` in addition to `child @ dynamic_`. The function thus cannot afford to just return an `option t`, since the caller would lose the knowledge that the element returned *is* the `child` object.

The search auxiliary function runs through the list (line 22) of adoptees and looks for one that is physically equal to `child`. It returns either the original list *minus* the element found, along with the knowledge that `child` now has type `t` (this is a success), or the original list, untouched (this is a failure). The search function uses a zipper, that is, it keeps a reversed list of the elements it has visited so far (the `prev` parameter) and examines the remaining elements (the `l` parameter).

The function uses the physical comparison `(==)` operator, which has the following type:

```
val ( == ) : (x: unknown, y: unknown) -> rich_bool empty (x = y)
```

That is, if one learns that `x` and `y` are *physically equal*, then in the `then` branch, the current permission is augmented with the `x = y` permission.

If the head of the list *is* `child` (line 29), the type-checker thus refines the environment with the `head = child` permission. We return all the elements that originally were in the list, save for `head`, using a call to `rev_append`. We return the `left` case: the type-checker checks, per the function's post-condition, that the return value is indeed a list `t`; using `head = child * head @ t`, it also checks that `child @ t` has become available.

If the head of the list is not `child` (line 31), we move the zipper one element forward and search for `child` in the remaining elements.

If no more elements remain to be searched (line 34), this means that `child` was not found in the list: we return the original list, which is the reverse of `prev`.

Finally, we call the search function (line 39). Since the function consumes the ownership of the children list, we need to write back into `parent` the list of remaining children. We return the correct boolean value.

```

1  open list
2  open either
3
4  alias adopter_ t = ref (list t)
5
6  alias dynamic_ = unknown
7
8  val dynamic_appears (x: _): (| x @ dynamic_) =
9      ()
10
11 val new_ [t] (): adopter_ t =
12     newref nil
13
14 val give_ [t] (parent: adopter_ t, child: (consumes t)): (| child @ dynamic_) =
15     parent := cons (child, !parent)
16
17 val take_ [t] (
18     parent: adopter_ t,
19     child: dynamic_
20 ): rich_bool empty (child @ t) =
21
22     let rec search (
23         consumes prev: list t,
24         consumes l: list t
25     ): either (list t | child @ t) (list t) =
26
27         match l with
28         | Cons { head; tail } ->
29             if head == child then
30                 left (rev_append (prev, tail))
31             else
32                 search (cons (head, prev), tail)
33         | Nil ->
34             right (rev prev)
35         end
36
37     in
38
39     match search (nil, !parent) with
40     | Left { contents } ->
41         parent := contents;
42         true
43     | Right { contents } ->
44         parent := contents;
45         false
46     end

```

Figure 4.4: Adoption/abandon as a library

```

1  | alias osf a b = {p: perm} (
2  |   (consumes (a | p)) -> b
3  |   | p
4  | )
5  |
6  | val make: [a, b, p: perm] ((consumes (a | p)) -> b | consumes p) -> osf a b =
7  |   identity

```

Figure 4.5: Affine closures as a library

This approach, as a library, naturally has several drawbacks when compared to the in-language adoption/abandon mechanism. The user has to request `dynamic_` explicitly as it is no longer part of the subsumption relation. The user pays for an extra indirection because the adopter field cannot be allocated inline. Also, the more efficient implementation with a hidden field is replaced by a linear search. More fundamentally, though, the direction is reversed: instead of a pointer from adoptee to adopter, the in-library approach uses a pointer from adopter to adoptee.

It is a good thing, however, that the mechanism should be expressible as a library; it provides, in a sense, a guarantee that our language has enough expressive power.

4.3 One-shot functions

I now turn to another interesting excerpt from the standard library, which is that of one-shot functions, a.k.a. affine closures. These form the basis for other libraries, such as iterators.

In type systems which possess a notion of linearity, one needs to be careful with closures. Some works [HP05](§1), [TP10] distinguish between duplicable closures, which can only capture duplicable elements, and affine closures, which can capture all elements, but may only be called once. Mezzo, in contrast, offers only duplicable closures.

We show how to define, as a library, one-shot functions, that is, functions that can capture affine permissions and may only be called once (Figure 4.5), thus sustaining our claim that allowing only duplicable closures is not a restriction.

A one-shot function is defined (line 1) as the conjunction of a function from a to b , which requires *some* permission p to be called, and the permission p itself. The exact nature of p depends on the function itself; in one case p may happen to be $r @ \text{ref int}$, in another case p may be $x @ a$. In order to abstract all these specific one-shot functions into a common type, the permission p is existentially-quantified. The type `osf a b` thus describes *any* function which requires a permission to execute.

Interestingly, the `make value` (line 6) is defined to be the identity function. Indeed, the type-checker is able to perform the right η -expansion and re-pack the existential on the right-hand side of the arrow.

This is a particular case of a general pattern, where we encode a coercion from τ_1 to τ_2 by ascribing the type $\tau_1 \rightarrow \tau_2$ to the identity function. We show more examples of this pattern in Chapter 5.

The library also offers an operation for composing two one-shot functions into another one-shot function, which we omit, as well as `make` and `apply` functions, for creating and applying a one-shot function respectively.

Currently, the library exposes the definition of the `osf` type, meaning that the type-checker can freely pack and apply one-shot functions, without the need for `make` and `apply`. Should the `osf` type be abstract, however, the user would definitely need to go through the `make` and `apply` functions exposed by the library's interface.

4.4 Rich booleans

The standard library module

Data types in Mezzo are more powerful than regular data types, in the sense that a branch can hold permissions in addition to regular fields. We leverage this for the definition of *rich booleans*, which we have seen already (§4.2 for instance).

When discriminating on a rich boolean (Figure 4.6), the user gains additional permissions depending on the branch they are in. A boolean is then defined to be a rich boolean which holds the empty permission in both

```

data rich_bool (p : perm) (q: perm) =
  | False { | p }
  | True { | q }

alias bool = rich_bool empty empty

val not : [p : perm, q : perm] (consumes rich_bool p q) -> rich_bool q p

```

Figure 4.6: The type of rich booleans

branches.

The negation of a rich boolean merely swaps its two type parameters. Expressing conjunction and disjunction, however, is trickier, due to the lazy semantics of the boolean operators. One might be tempted to write the following signature for the conjunction function:

```

val conjunction: [q: perm, p1: perm, p2: perm]
  (rich_bool q1 p1, rich_bool q2 p2) -> rich_bool (q1 v q2) (p1 * p2)

```

That is, in essence, the meaning of the conjunction operator: it is true if *both* arguments are true, and is false if *either one* of the arguments is false. We have no way, however, of expressing the disjunction of two permissions, as there is no anonymous sum in Mezzo. We thus require that q_1 and q_2 be equal, so that $q \vee q$ simplifies to q .

```

val conjunction: [q: perm, p1: perm, p2: perm]
  (rich_bool q p1, rich_bool q p2) -> rich_bool q (p1 * p2)

```

This function signature is now valid Mezzo, but does not correspond to what the `&&` operator should do. Indeed, it forces the evaluation of its second argument, while a regular `&&` operator only evaluates the second argument whenever the first one evaluates to `True`.

We solve this by using the following signature for `conjunction`, which encodes the lazy operational semantics of the `&&` operator:

```

val conjunction : [q : perm, p1 : perm, p2 : perm, s : perm] (
  consumes b1 : rich_bool q p1,
  b2: (| s * consumes p1) -> rich_bool q p2
  | s
) -> rich_bool q p2

```

One can read this signature as follows. Computing the conjunction of two permissions requires a rich boolean b_1 which holds q if `False`, p_1 if `True`.

In the case that b_1 is in the branch `True { | p1 }`, we need to evaluate the second boolean b_2 . To avoid strict evaluation, we make b_2 a function. When evaluated, the function receives p_1 . The function returns p_2 if `True`. The p_2 permission may subsume p_1 . The function may need to perform side effects; it may thus receive a permission s .

In the case that either b_1 or b_2 are false, they have to return the same permission q .

Desugaring

This now leaves the question of: what do we do when confronted with a $e_1 \ \&\& \ e_2$ expression in Mezzo? We need to ensure that type-checking is consistent with the runtime semantics of Mezzo. Indeed, since the type system tracks effects, the type-checking rules must be consistent with the operational semantics of the language.

- We cannot translate this into a call to the first version of `conjunction`: it would be unsound, since the first version of `conjunction` evaluates its second argument strictly. We cannot translate this into a call to the proper version of `conjunction` either, because function types are annotated in Mezzo. We would need to know, in advance, at parsing time, the correct values for q , s , p_1 and p_2 .

```

1 | val _ =
2 |   let x = 1 in
3 |   let y = 2 in
4 |   if x == y && true then
5 |     assert x = y
6 |   else
7 |     ()

```

Figure 4.7: A program involving conjunction rejected by the type-checker

- We could special-case the type-checker for `&&`. This basically amounts to performing type inference for the universally-quantified parameters of conjunction, which is challenging.
- What we do instead is translate `e1 && e2` as follows:

```

| e1 && e2 ≡ if e1 then e2 else false

```

This translation is correct, does not require a special-case for `&&` in the type-checker. It poses, however, similar inference challenges.

Inference difficulties

Anticipating on Chapter 12, here is an explanation of why inferring the right permissions for conjunctions poses technical difficulties.

The code from Figure 4.7 is correct, and should be accepted by the type-checker. Indeed, if one removes the `&& true` part at line 4, Mezzo accepts the program. The reason is that the program is desugared as follows:

```

1 | val _ =
2 |   let x = 1 in
3 |   let y = 2 in
4 |   let b =
5 |     if x == y then
6 |       true
7 |     else
8 |       false
9 |   in
10 |  if b then
11 |    assert x = y
12 |  else
13 |    ()

```

This triggers a merge situation (Chapter 12) for the disjunction at lines 5-8: the type-checker is faced with:

$$x = y * \text{ret}_l @ \text{True} \vee \text{ret}_r @ \text{False}$$

The mapping is trivial, so the algorithm skips directly to the pairwise merging step, and performs two subtractions:

$$x = y * \text{ret}_l @ \text{True} - \text{ret}_l @ \text{rich_bool } p_1 \ q_1$$

and

$$\text{ret}_r @ \text{False} - \text{ret}_r @ \text{rich_bool } p_2 \ q_2$$

Both these subtractions strengthen onto

$$x = y * \text{ret}_l @ \text{True} - q_1 * \text{ret}_l @ \text{True}$$

```

1  abstract lock (p: perm)
2  fact duplicable (lock p)
3
4  abstract locked
5
6  val new: [p: perm] (| consumes p) ->
7    lock p
8
9  val acquire: [p: perm] (l: lock p) ->
10   (| p * l @ locked)
11  val try_acquire: [p: perm] (l: lock p) ->
12   rich_bool empty (p * l @ locked)
13
14  val release: [p: perm] (l: lock p | consumes (p * l @ locked)) ->
15   ()

```

Figure 4.8: Axiomatization of locks

and

$$\text{ret, @ False} - p_2 * \text{ret, @ False}$$

Variables p_1 , p_2 , q_1 and q_2 are flexible; the type-checker must find suitable instantiations for them. In order to succeed, the type-checker would have to perform sophisticated reasoning and figure out that, out of all possible subsets of permissions, $x = y$ is the one it ought to pick. Indeed, in the general case, there are many more permissions available than just the two shown here! Right now, the type-checker refuses to explore all possible subsets, and picks empty which is a valid, albeit useless solution.

4.5 Locks and conditions, POSIX-style

Locks

Locks are remarkably important in Mezzo. We saw a sample usage of locks earlier (§3.1). Since they are used pervasively in typical Mezzo code, I show the interface for locks in Figure 4.8.

A lock is an abstract type which captures a permission (line 1). The lock is duplicable, which allows multiple threads to compete for it.

A lock is created via the `new` function (line 6). The lock is initially released, meaning that the permission is within the lock, hence the `consumes p` in the signature of `new`.

Acquiring a lock is typically done via the `acquire` function (line 9), which takes a lock `l`, blocks and returns when the lock has been acquired. The function returns no value, only the permission `p` which was previously held by the lock.

One can leverage the `rich_bool` type we saw earlier (§4.4) to define the `try_acquire` function (line 11), which is non-blocking. The function may fail to acquire the lock, hence the `rich_bool` return type.

Releasing a lock is done using `release` (line 14). One has to, again, give up the permission `p` protected by a lock in order to release it.

The extra `locked` type (line 4) is useful in the rare case where the lock protects a duplicable permission, or an affine permission of which there exists several copies. It ensures that the client cannot call `release` twice. This type, however, serves a much more essential purpose in the definition of condition variables (Figure 4.9).

Conditions

The type of conditions (line 1) is parameterized (at kind value) over the particular lock the condition refers to, meaning that `condition` is a (value-)dependent type. The lock and the corresponding condition variable are initially tied together when a call to `new` takes place (line 4). The dependent type of conditions is useful in the

```

1  abstract condition (l: value)
2  fact duplicable (condition l)
3
4  val new: [p : perm] (l: lock p) -> condition l
5
6  val wait: [l: value] (condition l | l @ locked) -> ()
7
8  data signal_whom =
9    | SignalOne
10   | SignalMany
11
12  val signal: [l: value] (condition l, signal_whom) -> ()

```

Figure 4.9: Axiomatization of conditions

```

1  abstract channel a
2  fact duplicable (channel a)
3
4  val new: [a] () -> channel a
5  val send_many: [a] (consumes xs: list::list a, c: channel a) -> ()
6  val receive: [a] (c: channel a) -> a

```

Figure 4.10: Specialized implementation of a channel (interface)

definition of `wait` (line 6), as it ensures that one can only wait on a condition variable if one holds the lock first, via the `l @ locked` permission. This provides additional safety.

The `signal` function (line 12) is standard.

An example: the work queue

The final example is the textbook implementation of a work queue. It uses an object-oriented style which I detail later on (Chapter 5). The code is divided in two parts: a specialized implementation of channels (Figure 4.11, Figure 4.10) and a server that leverages this (Figure 4.12).

The channel Channels are a communication primitive, where the *sender* may write asynchronously (in a non-blocking way) data into the channel, while the *receiver* extracts elements from the channel in a synchronous (blocking) manner. Channels are, naturally, duplicable, so as to support communication between multiple threads. Figure 4.11 shows a possible implementation of a `mychannel` module, which offers an interface tailored for the later server example. Figure 4.10 shows the corresponding interface.

A channel (line 3) is, internally, a mutable reference to a list, along with a lock that protects the reference and a corresponding condition variable. A channel is always carried around with the lock in the released state, meaning that the permission for the queue is *within* the lock, thus making the whole type duplicable.

Creating a new channel is done by calling the `new` function (line 9). The first step consists in creating the actual queue. We use a lock to protect the queue. For readability, we define a type alias for the invariant that the lock protects. We also create a new condition variable that will be used to signal to the receiver threads that the queue is non-empty.

The `send_many` function is standard (line 16): it acquires the lock, appends the new items at the back of the queue, and releases the lock. It then signals, to potentially many threads waiting on the condition, that new elements have been added into the queue. (If `send` were to take a single item, `SignalOne` would be sufficient).

The `receive` function (line 22) needs to secure a non-empty queue using the lock, so as to return an element to the caller. Since this function is blocking, it relies on the condition variable to block properly.

```

1  open list
2
3  data channel a = Channel {
4    queue:: unknown;
5    lock:: lock::lock (queue @ ref (list a));
6    condition:: condition::condition lock
7  }
8
9  val new [a] (): channel a =
10   let queue = newref nil in
11   let alias invariant: perm = queue @ ref (list a) in
12   let lock: lock::lock invariant = lock::new () in
13   let condition = condition::new lock in
14   Channel { queue; lock; condition }
15
16  val send_many [a] (consumes xs: list a, c: channel a): () =
17   lock::acquire c.lock;
18   c.queue := append (!(c.queue), xs);
19   lock::release c.lock;
20   condition::signal (c.condition, condition::SignalMany)
21
22  val receive [a] (c: channel a): a =
23   let Channel { queue; lock; condition } = c in
24   let alias invariant: perm = queue @ ref (list a) in
25   lock::acquire lock;
26   let rec loop (| lock @ lock::locked * consumes invariant):
27     (| queue @ ref (Cons { head: a; tail: list a })) =
28     match !queue with
29     | Cons ->
30       ()
31     | Nil ->
32       condition::wait condition;
33       loop ()
34   end
35   in
36   loop ();
37   let head = !queue.head in
38   queue := !queue.tail;
39   lock::release lock;
40   head

```

Figure 4.11: Specialized implementation of a channel


```

1  open list
2
3  data work_queue a =
4    WorkQueue {
5      add: (consumes list a) -> ()
6    }
7
8  val new [a] (n: int, consumer: (consumes x: a) -> ()): work_queue a =
9    let channel: mychannel::channel a = mychannel::new () in
10
11   let add (consumes xs: list a): () =
12     mychannel::send_many (xs, channel)
13   in
14
15   let worker (): () =
16     while true do begin
17       let element = mychannel::receive channel in
18       consumer element
19     end
20   in
21
22   let workers = init (n, fun (_) : _ = worker) in
23   iter (workers, thread::spawn);
24
25   WorkQueue { add }

```

Figure 4.12: Implementation of a server using `mychannel`

```

abstract channel a
fact duplicable (channel a)

val new: [a] () -> channel a
val send: [a] (channel a, consumes a) -> ()
val receive: [a] channel a -> a

```

Figure 4.13: The interface of channels

Let me describe the code in detail. After acquiring the lock, one needs to ensure that the queue is non-empty: this is the purpose of the `loop` function (line 26). The pre-condition of the function is the invariant, and the post-condition ensures that the queue *has become* non-empty: this is precisely what we need. We use a constructor type for that purpose, which embodies the fact that the queue is a `Cons` cell.

The function has to loop. Indeed, the `condition::wait` function (line 33) releases the lock associated with the condition, waits for a signal to be broadcast, then competes for the lock again. In-between the signal reception and the lock acquisition, another thread may have emptied the queue, hence the loop.

By the time we exit the loop (line 37), we learn that the queue is non-empty. We can thus take the head of the list, and release the lock (line 39).

Once the lock has been released, we return to the caller the element we just took out of the queue.

This example works remarkably well, and requires a relatively light amount of annotations. This is due to the duplicable nature of locks: the lock is implicitly captured by all the functions. These functions perform well-parenthesized calls to `acquire` and `release`, meaning that the only function which mentions the invariant in its signature is `loop`, as its purpose is precisely to refine the invariant.

The interface of channels, as found in the Mezzo standard library, is shown in Figure 4.13.

The server The code for the server is shown in Figure 4.12. The server, when initialized via `new` (line 8), spawns multiple threads, and offers an `add` function (line 3) that dispatches data onto the multiple threads. This is a *work queue*: items are dispatched onto *worker* threads.

The `new` function (line 8) takes an integer `n`, which represents the number of workers to be created, and a consumer. Each worker, when fed a new element, will call the consumer on it. In essence, the consumer function represents what the client wishes to do with each element.

Internally, the server relies on a channel (line 9) for the communication between the client and the worker threads. Sending new items for processing merely amounts to sending them onto the channel (line 11).

The worker function represents one thread of execution (line 15): in this simplified example, the worker lives forever (hence the `while true` loop) and repeatedly extracts items from the channel in order to process them by calling `consumer`.

Once the worker function is defined, the server can spawn the multiple threads (line 23). It creates a list of identical workers, then calls `thread::spawn` for each one of them.

Finally, the function returns (line 25) the work queue object that was just created.

4.6 Landin's knot (recursion via the mutable store)

This example is fairly advanced, and rather fragile, as it poses type-checking difficulties. These difficulties are discussed in Chapter 11; §11.7 provides detailed explanations.

Let me nonetheless show this programming pattern. Because the reasons why the code type-checks are fairly advanced, I need to anticipate on Chapter 8 and mention some subsumption rules of Mezzo. Landin's knot (Fig-

```

1  data patched (r: value) a b =
2    Patched { contents : (x : a | consumes r @ patched r a b) -> b }
3
4  val fix [ a, b ] (ff : (a -> b) -> (a -> b)) : a -> b =
5    let r = newref () in
6
7    let f (x : a | consumes r @ patched r a b) : b =
8      let self = r.contents in
9      ff self x
10   in
11
12   r.contents <- f;
13   tag of r <- Patched;
14   f

```

Figure 4.14: Landin's knot

ure 4.14) implements recursion through the mutable store by backpatching a function reference. The `fix` combinator (line 4) allocates a reference cell at line 5; the function `f` assumes *in advance* that the reference cell has been patched properly (line 7) and now contains a reference to `f` itself. At line 9, the current permission is thus:

$$r @ \text{patched } r \ a \ b * \text{self} @ (a \mid \text{consumes } r @ \text{patched } r \ a \ b) \rightarrow b$$

By `HIDEDUPLICABLEPRECONDITION`, we can obtain:

$$\text{self} @ a \rightarrow b$$

which allows the recursive call to take place.

The rest of the code needs to make sure that the yet-unfulfilled pre-condition for `f` is met. This is done by performing a strong-update and writing `f` into the `contents` field of the reference (line 12) which, until now, contained a placeholder unit value. This “ties the knot”.

The key insight is that the type `patched r a b` is *duplicable*, which enables the weakening of the function type into `a -> b`. Without it, we would not be able to subtype `self` into a function from `a` to `b` at line 9 like we did above.

We thus need to freeze the reference into the immutable, hence duplicable, patched data type: this is the purpose of the tag assignment at line 13. The last line contains, again, advanced type-checking reasoning. By using the permission for `f`, we can obtain `r @ patched r a b`; using `HIDE DUPLICABLE PRECONDITION` again, we obtain `f @ a -> b`.

4.7 Other interesting bits

F. Pottier wrote the `lazy.mz` module which defines the lazy type *in a library*. In contrast, OCaml provides a built-in type. The module uses reified coercions to encode bounded polymorphism. The module relies on weak references (protected by a lock); combined with bounded polymorphism, the lazy type can be exported as covariant. The source code is heavily commented, should a curious reader wish to take a look. The `stream` module leverages the lazy module.

F. Pottier and A. Guéneau worked on iterators in Mezzo, while I wrote the supporting code in the Mezzo type-checker [GPP13]. The module uses the same borrowing pattern we demonstrated earlier §3.2, although in a much more sophisticated way. A generic library for iterators is available in the standard Mezzo library as `iterator.mz`; it provides the standard `map`, `filter`, `zip`, `fold` operations over iterators. Specific modules (`list`, `mutableTreeMap`) provide iterators which implement the `iterator` interface.

5. Writing programs in Mezzo

The present chapter highlights a few programming patterns that we discovered while writing Mezzo programs. Oftentimes, these patterns revolve around the difficulty of type-checking programs that have a non-trivial ownership discipline.

Several of these points appeared in the previous chapters. I discuss them more thoroughly. The discussion anticipates over the formal presentation of the Mezzo type-checking rules. The reader may wish to come back later to the present chapter.

5.1 Ownership and function signatures

When writing a function in Mezzo, one must think about the interface the function exposes to its callers. Mezzo revolves around ownership: the programmer hence must be careful. One may *ask for a lot of guarantees*; worded differently, a function may require many permissions to execute, making it difficult to call, but making type-checking its body easier. Conversely, one may write a function by *demanding very little*, which eases things from the caller's perspective, but makes type-checking the body of the function hard, because of missing assumptions.

Which style should be adopted depends very much on the nature of the function. It sometimes may be easy to figure out which style to adopt; in some other situations, demanding the right pre-conditions can be a little tricky. Let me illustrate these points.

Easy case: *not using consumes* The ideal case is when the programmer succeeds in writing a function with type $(x : t) \rightarrow u$. Lacking any *consumes* annotations, the function *preserves* ownership of its argument. This behavior is optimal from the caller's point of view: they do not lose any permission and must only exhibit $x @ t$ in order to call the function.

Informally, we say that “it is better to preserve the argument”. Formally:

$$(x : t) \rightarrow u \leq (\text{consumes } x : t) \rightarrow u$$

(This claim is hard to prove at this stage of the dissertation. Anticipating on the next chapters, let me just say that once the types above are converted into the internal syntax, proving this subtyping relation requires a mere application of the subsumption rule COARROW.)

An example of such a function is `list::length`: the programmer can write the “best type”, so they should just go ahead and not even bother adding a *consumes* annotation.

```
| val length: [a] list a -> int
```

Easy case: *one has to use consumes* Not all functions can be written with such an optimal type. A typical situation is, upon type-checking a function's body, realizing that the permission for the argument is no longer

available at the end of the function body. A natural escape hatch thus consists in adding a `consumes` annotation for the said argument.

Consider the following snippet, where the `mlist` type denotes mutable lists.

```
(* val rev_append: [a] (consumes xs: mlist a, consumes ys: mlist a) -> mlist a *)

val rev [a] (xs: mlist a) : mlist a =
  rev_append (xs, MNil)
```

```
Could not obtain the following permission:
  xs @ mlist::mlist a
```

```
Variable xs is defined around there:
File "test.mz", line 3, characters 13-24:
```

```
val rev [a] (xs: mlist a) : mlist a =
  ^^^^^^^^^^^^^
```

```
No useful permissions are available for it.
```

In the example above, the function cannot type-check since the permission for the argument `xs` is definitely gone. In this case, this is *inherent to the semantics of the function*: one *has to* add the `consumes` keyword. Accepting this function without the `consumes` keyword would be a breach of soundness. This also is an ideal case: it is clear that the programmer *must* use the `consumes` keyword when writing the function.

In case the function consumes its argument, the caller will do its best to save a copy of the corresponding permission, if possible.

A complex case Let us now move on to a slightly subtler situation where not using `consumes` is impossible, and using `consumes` is unsatisfactory. I take the example of a programmer who wishes to implement the `ref::get` function.

```
val get [a] (x: ref a): a = x.contents
```

The function, with such a signature, will be rejected by the type-checker. Indeed, the function duplicates ownership of the reference's contents. Lacking any extra assumption, the function cannot admit such a type in Mezzo.

A solution is, per the paragraph above, to add a `consumes` annotation.

```
val get1 [a] (consumes x: ref a): a = x.contents
```

This gives, however, a pretty useless type to the `get` function, since the caller can no longer use the reference after getting its contents. A better solution is to add an extra duplicability assumption.

```
val get2 [a] duplicable a => (x: ref a): a =
  (* x @ ref a *)
  (* x @ Ref { contents = c; } * c @ a *)
  x.contents
  (* x @ Ref { contents = c; } * c @ a * ret = c *)
  (* x @ Ref { contents = c; } * c @ a * ret @ a * ret = c *)
  (* x @ ref a * ret @ a *)
```

Seeing why the function is now well-typed is not trivial. The type-checker applies several reasoning steps. The first one consists in expanding `ref`, a data type with one branch, into the corresponding concrete type. The type-checker then introduces a name for the `contents` field, namely, `c`.

If `ret` denotes the name of the function’s return value, then `ret` is `c`; knowing that `a` is duplicable, the type-checker duplicates the permission `c @ a`, and uses `ret = c` to rewrite the copy into `ret @ a`. The type-checker is then able to assert both `x @ ref a` and `ret @ a`: the function is well-typed.

It seems that this function signature is excessively restrictive: indeed, at first sight, one can only get the contents field of a reference if it is duplicable.

From the caller’s point of view, however, things are just fine. The type-checker applies similar transformations.

```
(* x @ ref (m!ist int) *)
(* x @ Ref { contents = c } * c @ m!ist int *)
let y = get2 x in
(* x @ Ref { contents = c } * c @ m!ist int * y = x *)
...
```

Type-checking the call to `get2` requires the type-checker to exhibit `x @ ref ?a` with `?a` suitably chosen to be a duplicable type. Remember that a singleton type conveys no ownership information and is thus duplicable. The problem hence admits a solution with `?a` equals to `(=c)`.

The caller keeps `x @ Ref { contents = c }` but also gains `y @ (=c)`, that is, `y = c`.

This non-trivial reasoning that happens in the type-checking engine is only made possible because the `ref` module exposes the concrete definition of the `ref` type; lacking this knowledge, the type-checker would be unable to perform the sophisticated reasoning steps shown above.

Using a singleton type to enforce proper usage There is, however, a catch: the problem of finding `x @ ref ?a` with `?a` duplicable admits, besides the singleton type, another solution, which is unknown, our top type. For design reasons, the type-checker does *not* backtrack at the expression level. That is, the type-checker does not consider multiple solutions for type-checking a function call and picks an arbitrary one. It turns out that, for numerous reasons (§11.7, §12.4), the type-checker never picks a solution that involves a singleton type. This means that even though the typing derivation described above is the “right one”, the type-checker’s inference procedure will fail to use it.

The solution consists in rewriting the type of the `get` function.

```
val get3 [y: value] (x: ref (=y)): (=y) =
  x.contents
```

This way, we make sure that the type-checker is forced to use a singleton type. One could wonder whether this signature is better than that of `get2`. The two are equivalent. They are, however, strictly better than the initial, bad `get1` signature.

Figure 5.1 shows how one makes sure that the type-checker agrees with the informal discourse and is indeed able to figure out that these types are equivalent. (This anticipates slightly on §5.3.) Essentially, whenever *asked to* verify that the two types are equivalent, the type-checker is able to do so; it does not, however, introduce singleton types by itself.

Discussion: which type is better? It is hard to come by with a definitive answer, and this is perhaps a weakness of Mezzo. Right now, there are several possible ways for writing a function type; picking the best one demands some expertise, and a good deal of thinking about what the function really means in terms of ownership. Not only that, but it also requires the programmer to know how inference works under-the-hood.

Preserving the argument is, naturally, desirable. Not all functions are meant to preserve their arguments, however. In that case, the `consumes` keyword seems inevitable, unless one can get by with a singleton type.

Whether one should use a singleton type or not requires knowledge about the type-checker. The fundamental reason why using a singleton type with a reference works is that the type-checker applies the following coercion transparently (`UNFOLD, DECOMPOSEBLOCK`):

$$\text{ref } a \equiv \exists(x : \text{value}) \text{ ref } (=x) * x @ a$$

One may be tempted to write an infallible `option::get` with the following type:

```
val get: [x: value] (option (=x)) -> (=x)
```

```

val id [a] (consumes x: a): a = x
alias subtype t u: perm = id @ t -> u
alias equiv t u: perm = subtype t u * subtype u t

alias t1 =
  [a] (consumes (x: ref a)) -> a

alias t2 =
  [y: value] (x: ref (=y)) -> (=y)

alias t3 =
  [a] duplicable a => (x: ref a) -> a

val _ =
  (* t2 ≡ t3 *)
  assert equiv t2 t3;
  (* t3 ≤ t1 *)
  assert subtype t3 t1;
  (* t2 ≤ t1 *)
  assert subtype t2 t1;

```

Figure 5.1: Using the identity to encode coercions

This, however, would fail to work, as it requires the following rule which is false, as it only holds in the Some case.

$$\text{option } a \equiv \exists(x : \text{value}) \text{ option } (=x) * x @ a$$

5.2 Higher-order effects and crafting signatures

As we saw earlier, higher-order functions need to take into account effects.

The map example Let us start with the most basic signature for the map function, shown below.

```

val map1: [a, b] (consumes xs: list a, f: (consumes a) -> b) -> list b

```

Before tackling the issue of effects, let us make sure that consumes annotations are used properly in the context of a higher-order function.

We saw earlier that we *must* consume the ownership of xs , and that the caller will duplicate, if possible, the permission. The first occurrence of consumes is thus used properly.

A subtler point is the signature of f . We favor the caller: the map function accepts the least restrictive type for its argument. A caller can pass a stronger type for f if they wish, that is, a function that preserves its argument will work for f .

Formally, using contravariance in the domain of arrows as well as the earlier coercion (§5.1), we can establish that:

$$\forall a, \forall b. (\text{consumes list } a, (\text{consumes } a) \rightarrow b) \rightarrow \text{list } b \leq \forall a, \forall b. (\text{consumes list } a, a \rightarrow b) \rightarrow \text{list } b$$

This thus means that the second consumes annotation is used properly: omitting it would place a stronger burden on the caller.

Now that we have made sure that the consumes annotations are used properly, let us take a closer look at the signature of map1. This function type is suboptimal because f cannot perform effects. A way to improve upon the signature of map1 is to introduce an extra permission s .

```
val map2: [a, b, s: perm] (
  consumes xs: list a,
  f: (consumes a | s) -> b
  | s
) -> list b
```

Let me first point out that this signature is a subtype of the former, since picking $s = \text{empty}$ gives back `map1`.

This signature is more interesting, because now f can, seemingly, perform any sort of side-effects. The function can, for instance, increment a counter. There is however one variety of side-effects that f cannot perform, which is to modify the element in the list xs . For that, we need yet another type for `map`.

```
val map3: [a1, a2, b, s: perm] (
  consumes xs: list a1,
  f: (consumes (x: a1) | s) -> (b | x @ a2)
  | s
) -> (list b | xs @ list a2)
```

This type is a subtype of `map2`, so we are indeed performing refinements of the original type. This time, it looks like we have the “best” type for `map`.

Quite frustratingly, there is no theoretical argument to support the claim that this is the “best type”. It is, indeed, a subtype of the other two, but there is no meta-theoretic argument justifying that we cannot obtain a better one.

Another source of dissatisfaction is that the type above, again, requires quite some expertise to craft. This is the type that the standard library of Mezzo uses. We suspect, however, that only authors of libraries need to preoccupy themselves with optimizing the types in their signatures; casual users, as long as their code works, will probably stick to one of the less sophisticated versions.

Anticipating on subsequent sections, let us just remark that having four universal quantifications raises challenges for the type-checker. Fortunately, since function types are annotated, by comparing the formal type for f and the effective type of the argument, the type-checker can figure out the correct values of a_1 , a_2 , b and s .

Counting function calls with types Let us now imagine that we wish to write versions of `map` for references and options. To better illustrate the point, I use immutable references.



We assume the type `iref` to have an opaque definition, meaning that we do not have $(\text{iref } a) | p \equiv \text{iref } (a | p)$. If the definition of `iref` (shown below) were transparent, the type-checker would be able to prove this fact.

```
data iref a = IRef { contents: a }
```

We wish, just like with `map2`, to allow the function passed as an argument to perform effects. I stay at the level of detail of `map2`; that is, the function f is not expected to modify the element stored in the container. Interestingly, the signature of `map_iref` and `map_option` is going to be slightly different from `map2`.

```
val map_iref: [a, p: perm, s: perm] (
  consumes x: iref a,
  f: (consumes (a | p)) -> (b | s)
  | consumes p
) -> (iref b | s)

val map_option: [a, p: perm] (
  consumes x: option a,
  f: (consumes (a | p)) -> b
  | consumes p
) -> (option b)
```


These functions exhibit slight variations on the pattern in `map2`. The `map_iref` function takes `f`, which consumes `p` and produces `s`; then, `map_iref` itself consumes `p` and produces `s`, unconditionally. This type reveals that, assuming it terminates, `map_iref` calls its argument `f` *exactly once*.

The `map_option` function, however, does not promise to return any permission that `f` would return: from this, we can conclude that `map_option` calls its argument `f` *at most once*.

Finally, looking back at the signature of `map2`, the fact that `map2` preserves any permission its argument `f` preserves reveal that `f` may be called an *arbitrary number of times*.

Sequencing function calls with types In the same vein, the signature below reveals that:

- `h` calls `f` at most once;
- `h` calls `g` only after calling `f`;
- if `h` terminates, it has called `g`.

Worded differently, assuming it terminates, `h` calls `f`, once, then `g`, once. (The functions return the unit type for the sake of readability.)

```
val h: [p, q, r] (
  f: (| consumes p) -> (| q),
  g: (| consumes q) -> r
  | consumes p
) -> (| r)
```

5.3 Reifying coercions

A coercion is a function from types to types, which has no run-time representation. A coercion allows one to convert an element of type `t` into type `u`. There is no notion of first-class coercions in Mezzo; however, coercions can be emulated.

Beware: this section uses the internal syntax which has not been introduced yet (Chapter 7).

First encoding using the identity function

This first encoding is the one we saw in Figure 5.1. The definition of the identity gives $\text{id} @ \forall a.a \rightarrow a$. The classic subtyping rule for functions is as follows.

$$\text{SUB-ARROW} \quad \frac{t_2 \leq t_1 \quad u_1 \leq u_2}{t_1 \rightarrow u_1 \leq t_2 \rightarrow u_2}$$

If we have $\text{id} @ \forall a.a \rightarrow a$, we can *instantiate* this type and obtain $\text{id} @ \tau \rightarrow \tau$. If $\tau \leq \sigma$, then SUB-ARROW gives $\text{id} @ \tau \rightarrow \sigma$.

Phrased differently, if $\tau \leq \sigma$ is true in the system, one can obtain $\text{id} @ \tau \rightarrow \sigma$. We call this a *coercion witness*.

Having a coercion witness is strictly less powerful than a “real” coercion. Indeed, having $\text{id} @ \tau \rightarrow \sigma$, the type-checker, lacking any special understanding of the `id` function, cannot conclude that $\tau \leq \sigma$ and use this fact accordingly.



Intuitively, by having $\text{id} @ t \rightarrow u$, one can establish $\Vdash x @ t \leq x @ u$, which is the definition of $t \leq u$. Deriving this relation is not, however, supported by the current Coq formalization of Mezzo.

Having this downgraded version of coercions thus requires the user to manually apply coercions *via* expressions. In particular, the user has to write an expression to apply a coercion under a context. Take the case of lists: if $\tau \leq \sigma$, a system with full coercions would allow one to deduce $\text{list } \tau \leq \text{list } \sigma$ thanks to a variance analysis on the list data type. With our coercion witnesses, one needs to write specific code for that purpose (Figure 5.2), as one cannot obtain $\text{id} @ \text{list } t_1 \rightarrow \text{list } t_2$ from $\text{id} @ t_1 \rightarrow t_2$. Moreover, the encoding has the drawback that `apply_list` does not convey the fact that the return value is the same value as the argument.

This encoding of coercions has one drawback: several permissions become available for the identity function.

```

open coercions
open list

val apply [t1, t2] (consumes x: t1 | subtype t1 t2): t2 =
  id x

val rec apply_list [t1, t2] (consumes x: list t1 | subtype t1 t2): (list t2) =
  match x with
  | Cons -> Cons { head = apply x.head; tail = apply_list x.tail }
  | Nil -> Nil
  end

```

Figure 5.2: Manually applying a coercion

```

(* id @ [a] consumes a -> a *)
assert subtype t1 t2
(* id @ [a] consumes a -> a * id @ consumes t1 -> t2 *)

```

One cannot discard these coercion witnesses. The user provided information by asserting that t_1 is a subtype of t_2 ; lacking this user-provided hint, the type-checker would have to decide on its own to obtain a new coercion witness and try all possible t_2 's to find out which subtyping witness works: our implementation, quite obviously, does not perform such a massive exploration.

Keeping these witnesses has a performance impact, since the type-checker now needs to backtrack and try every possible coercion witness when trying to justify a call to `apply`.

Alternative encoding using fresh functions

An alternative encoding of subtyping would be as follows.

```

alias subtype' t1 t2 = [x: value] (| consumes x @ t1) -> (| x @ t2)

```

This encoding is more powerful than the one based on the identity function, because it preserves the identity of the element being coerced. Just like the other encoding, one can, using `subtype' t1 t2`, exhibit `subtype' (list t1) (list t2)` (Figure 5.3)

```

alias subtype' t1 t2 = (x: unknown | consumes x @ t1) -> (| x @ t2)

open list

val lift_list [t1, t2] (c: subtype' t1 t2): subtype' (list t1) (list t2) =
  let rec c_list (x: unknown | consumes x @ (list t1)): (| x @ (list t2)) =
    match x with
    | Cons { head; tail }->
      c head;
      c_list tail
    | Nil ->
      ()
    end
  in
  c_list

```

Figure 5.3: Lifting a coercion on lists

The encoding chosen in Figure 5.3 is slightly different, though: we need x to be a program variable, not just a type variable, in order to match on it. Admittedly, a ghost code mechanism would allow matching on fictional variables; in the current state of things, though, we must add an x : unknown parameter to the function and pass it the element to coerce.

An advantage of this encoding is that we no longer have multiple function types attached to the same program variables, that is, the `id` function. This requires the user to specify, using its name, which coercion should be applied: this is more readable, and makes the job of the type-checker easier.

A drawback of this encoding is that we rely on coercion authors to be well-behaved and write terminating functions that perform no side-effects.

This encoding was used in Figure 3.11 at line 18 and line 28. One-shot functions (§4.3) also use this encoding, except that they quantify over an arbitrary p and q instead of $x @ t_1$ and $x @ t_2$.

Two unsatisfactory solutions

This digression on coercions shows both the expressive power of Mezzo and its limitations. The system can express coercions, but the user has to manually apply them, possibly incurring a run-time cost for just doing nothing (the cost of calling `apply` or, worse, `apply_list`).

One could get away with having a ghost code mechanism, and declare the `apply` functions to be ghost. Applications of coercions would still look like regular term applications, but wouldn't incur a run-time penalty. This ghost-code mechanism has been used successfully in other verification tools [Fil03]; we are not sure yet as to whether we want to add this mechanism in Mezzo. In the meanwhile, we rely on the compiler to eliminate the calls to `apply`. The calls to `apply_list` seem harder to get rid of.

The encoding can be made general (the `id` function), but has a cost in terms of exploration in the actual type-checker, meaning the user may want to use a less powerful, more specific encoding to accommodate the implementation constraints.

These encodings are already useful as is, as they allow us to express bounded polymorphism [CW85] in Mezzo. The encoding of $\forall[t_1 \leq t_2]. t_1 \rightarrow u$ becomes, using the type aliases from Figure 5.1:

```
[t1] (x: t1 | subtype t1 t2) -> u
```

5.4 Object-oriented programming

Modeling object-oriented programming using a simpler calculus such as System F has been an active topic of research since the end of the 80's until the 90's. One line of work uses bounded polymorphism and subtyping relations to model inheritance [CW85]; subsequent refinements have been proposed, such as F-bounded polymorphism which better accounts for "self types" [CCH⁺89]. Another line of work tries to stay within the context of ML type inference and uses row polymorphism [Ré89].

Mezzo currently does *not* have extensible records, meaning that there is no concept of row polymorphism. We have seen, however, that subtyping witnesses could be expressed within Mezzo. Let us see how much of objects can be encoded that way.

Encodings using Mezzo "as is"

Currently, one can program in an object-oriented style in Mezzo [GPP13] using records with functions. The object may have some internal state; the record will thus be quantified over some abstract permission.

The pattern from Figure 5.4 demonstrates what we call the "object-oriented style": the record has *methods* (functional fields), *internal state* (the existentially-quantified permission), a *member field* (r) and a *constructor* (the new function). This matches closely Cook's description of data abstraction via procedural abstraction [Coo91].

(The counter data type will *not* be made abstract in the interface; the abstraction already lies within the existentially-quantified permission.)

Therefore Mezzo can, in a way, express "objects", that is, model procedural abstraction. There is no support, however, for objects in our subtyping relation. Indeed, our subtyping relation is designed around data types, which are central in an ML-inspired language, and is not particularly well suited for performing subtyping operations in the object-oriented sense.

```

data counter = { state: perm }
  Counter {
    incr: (| state) -> ();
    get:  (| state) -> int
    | state
  }

val new (): counter =
  let r = newref 0 in
  let alias state: perm = r @ ref int in
  let incr (| state): () =
    incr r
  in
  let get (| state): int =
    !r
  in
  Counter { incr; get }

```

Figure 5.4: An object-oriented encoding of a counter

The following subtyping operations would be needed for encoding a “width-subtyping” relation suited to objects:

- shrinking a record by removing fields from the end;
- transparently changing an immutable record into another one as long as the tags match *and* the types match.

These two informal points above could be expressed via the following subtyping rules:

$$\begin{array}{c}
 \text{SUB-SHRINK} \\
 \hline
 x @ A \{ \vec{f} : \vec{t}; \vec{f}' : \vec{t}' \} \leq x @ A \{ \vec{f} : \vec{t} \}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{SUB-TAG-IMM} \\
 A \text{ and } B \text{ are the same integer} \\
 \hline
 x @ A \{ \vec{f} : \vec{t} \} \leq x @ B \{ \vec{f} : \vec{t} \}
 \end{array}$$

With these two rules, one could define a subtype of counter, say, `decr_counter`, with an extra `decr` method. The system would then be able to upcast automatically into counter whenever needed.

```

decr_counter ≤

DecrCounter {
  incr: (| state) -> ();
  get:  (| state) -> int;
  decr: (| state) -> ();
}
≤ DecrCounter {
  incr: (| state) -> ();
  get:  (| state) -> int;
}
≤ Counter {
  incr: (| state) -> ();
  get:  (| state) -> int;
}
≤ counter

```

The two rules are compatible with the run-time model of Mezzo. `SUB-SHRINK`, however, must *not* perform at run-time a shrinking of a GC block, even though the operation is supported by the OCaml runtime system. Indeed, the object may be duplicable, meaning that there may be other aliases to it; it would be unsound to shrink the corresponding memory block. (In any case, a subtyping operation should not translate into a run-time operation!) A potential drawback of `SUB-SHRINK`, which may explain why we haven’t added the rule to the language, is that it creates potential memory leaks. Indeed, the user may not be aware of the extra field, meaning that they may inadvertently keep alive references they are not aware of.

The encoding is still fairly cumbersome, and many built-in mechanisms that make object-oriented programming practical are missing. There are no visibility modifiers for fields; there are no facilities for subclassing an existing class definition and having default implementations for methods. In short, Mezzo can model procedural abstraction, but lacks support for proper subtyping and inheritance mechanisms.

Comparison with row polymorphism and F-bounded quantification

In OCaml, row polymorphism is used primarily for inferring object types, since row variables are a natural fit for working with a Damas-Milner style unification algorithm. Moreover, they also work naturally in the presence of “self types” and inheritance.

An earlier alternative to row polymorphism appeared in the literature, namely, bounded polymorphism and later variants, such as F-bounded polymorphism. Since Mezzo can express subtyping (as we saw in the preceding sections), let us see how far Mezzo can go, using explicit annotations on function types and subtyping witnesses.

A problem discussed extensively in Mitchell *et al.*'s '89 paper [CCH⁺89] is that of self types. Self types appear when the type of an object references itself. The first example from the paper is as follows, where the “self type” appears in covariant position.

```
Movable = Rec mv. { move: Real × Real → mv }
```

In OCaml, this would be expressed as follows.

```
# type movable = < move: float * float -> 'a > as 'a
```

The challenge in the paper is to type-check a translate function, and infer the proper type for the function. In OCaml:

```
# let translate x = x # move (1.0, 1.0);;
val translate : < move : float * float -> 'a; .. > -> 'a = <fun>
```

The important thing to notice here is that the translate function takes an argument of type 'a and returns a result with the exact same type 'a: this is the technical difficulty that the paper describes. Using row polymorphism, the function type is correctly inferred.

Regular bounded quantification, however, fails to express this pattern properly, as it assigns the following type to translate:

```
translate = fun (x: Movable) x.move(1.0, 1.0)
translate: ∀r ⊆ Movable. r → Movable
```

The paper's F-bounded quantification succeeds in inferring the right type, however:

```
translate = fun (x: Movable) x.move(1.0, 1.0)
translate: ∀r ⊆ F-Movable(r). r → r
```

Let us see now how Mezzo fares on this example. In Mezzo, writing translate is trickier. We need to slightly tweak our earlier encoding of subtypes for this to work (Figure 5.5).

Mezzo succeeds in type-checking translate with an expressive enough type. It may thus seem that Mezzo has the expressive power of F-bounded polymorphism; however, the second example from the paper seems impossible to express in Mezzo. It thus looks like Mezzo stands in-between bounded polymorphism and F-bounded polymorphism.

These encodings are cumbersome and we do not expect users to write such code; it remains important, however, to explore the expressive power of the language.

Objective Mezzo?

Instead of wondering what can be added to Mezzo to make it more object-oriented friendly, we can instead ask: what would the language look like if we had chosen to design an OO-like language instead of an ML-like language?

Indeed, we believe several of the key Mezzo mechanisms to be reusable in an object-oriented setting:

- the entire permission framework;
- the notion of duplicable *vs.* affine objects;
- the “fact” mechanism;
- the adoption/abandon mechanism.

```
(* A object of class [c] which has been coerced to its interface [movable] *)
data movable c =
  Movable { move: (int, int) -> c }

(* A type [c], also a subtype of [movable]. *)
alias sub c = (c | subtype c (movable c))

(* Example 3.2 from the paper *)
val translate [c] (consumes x: sub c): sub c =
  if true then
    x
  else begin
    (* Apply the coercion *)
    let x = apply x in
    x.move (1, 2)
  end
```

Figure 5.5: Encoding Mitchell *et al.*'s movable example in Mezzo

The central feature of Mezzo, data types, would probably be traded for a class-based mechanism that features inheritance, visibility modifiers, and possibly interfaces. The compilation model may require switching to a runtime environment that it better suited to objects, such as the JVM or CLR.

This is purely speculative, but may constitute a good research topic should anyone want to explore this direction.

Part III

Mezzo, formalized

6	A Mezzo reference	
6.1	Differences with the informal presentation	79
6.2	Meta-variables	80
6.3	Types and kinds	80
6.4	Type definitions	81
6.5	Modes and facts	82
6.6	Programs	83
6.7	Module layer	85
7	Translating Mezzo to Simple Mezzo	
7.1	Examples	88
7.2	Kind-checking	90
7.3	Translation	95
8	Type-checking Simple Mezzo	
8.1	Memory model	99
8.2	Subsumption rules	101
8.3	Type-checking rules	103
8.4	The duplicable and exclusive modes	108
8.5	Facts	112
8.6	Variance	116
8.7	Signature ascription	117
8.8	Differences between Simple Mezzo and Core Mezzo	118
8.9	Reflecting on the design of Mezzo	120

This part presents Mezzo in a formal way.

The first chapter (Chapter 6) provides a reference of the Mezzo language as exposed to the user. All the constructs of the language are listed along with a brief explanation. This should hopefully help the reader write programs and understand what is legal Mezzo and what is not.

The user-facing syntax of Mezzo provides several syntactic facilities that ease writing programs; when reasoning, however, it is easier if these syntactic facilities have been eliminated. We thus define in Chapter 7 a variant of Mezzo called Simple Mezzo which features none of the syntactic facilities.

This allows us to write the various rules that govern the type-checking of Mezzo in Chapter 8. We detail the procedures for kind-checking, type-checking, as well as other side checks such as fact inference and variance computations.

6. A Mezzo reference

The present chapter aims at providing a complete reference of the user syntax of Mezzo. Since such a reference does not exist elsewhere, the present dissertation seems like a natural fit.

This chapter hopefully should make it apparent that we devoted a lot of effort into providing constructs that make Mezzo a *usable* language. We do not claim that a novice programmer can jump in and program in Mezzo—we are far from that goal. Yet, many of the constructs presented below have been added into the language with the sole goal of making it easier to program in Mezzo. Features such as type aliases, record “punning”, optional permissions nested in data type branches, or the special syntax for function types (Chapter 7) bring absolutely nothing from a theoretical point of view. Indeed, all the formal reasoning is done over a core calculus that features none of these constructs. Yet, we believe that many, if not all of the examples we have seen before, would have been impossible to write in the core calculus, as they would have been too verbose.

This is not a rigorous reference, in the sense that I do not list all the syntactic categories that the parser uses (there are too many of them). Also, the notation is “informal BNF”. I write \bar{b} for a possibly-empty list of b , $b?$ for an optional b , and b^+ for a non-empty list of b .

The curious reader can consult `parsing/grammar.mly` and `parsing/SurfaceSyntax.ml` in the source code of Mezzo; the former contains the grammar for the language, while the latter contains data type definitions which pretty much match the syntactic categories below.

6.1 Differences with the informal presentation

This chapter marks the beginning of a more formal presentation of Mezzo, and covers in an exhaustive manner the user-facing language, that is, the set of programs that the user can write.

The previous part may have elided some implementation details: this chapter makes some features apparent, while the previous part left them “under the hood”. This chapter also starts referencing some files from the Mezzo prototype: indeed, things described from now on correspond faithfully to what I implemented in the Mezzo type-checker during my PhD.

The two points that were not made explicit during the informal presentation revolve around *modes*.

Modes Modes have been pretty much kept out of the discussion. Indeed, we only mentioned duplicable and affine permissions, saying that “the system” knew how to decide whether a permission was duplicable or not. At some point, a new mode appeared (§4.1), the *exclusive* mode. It turns out that there are three such *modes* in the system: *affine*, *duplicable* and *exclusive*. The *duplicable* mode is familiar already. The *exclusive* mode denotes a memory block in the heap that we have *unique ownership of*. (Currently, the only such blocks are mutable blocks: any nominal or structural permission for a data type marked as mutable is, in fact, *exclusive*.) The *affine* mode is a strict superset of the two: some types, such as `list (ref int)` are neither *duplicable* or *exclusive*. The fact that only *duplicable* permissions may be copied remains true.

Mode constraints While the programmer can think of `x @ t` and `duplicable a` as both being permissions, it turns out that a different mechanism is used for the latter. In fact, `duplicable a` is not treated as a permission but as a *mode constraint*, whose satisfiability depends on an underlying lattice of known *facts*. This part makes the distinction apparent.

6.2 Meta-variables

I use meta-variables in this chapter. Their names hint at a specific kind.

- `k` stands for a kind
- `a`, `b` stand for type variables at kind type
- `x`, `y` stand for type variables at kind value or program variables
- `X`, `Y` stand for type variables at any kind
- `t`, `u` stand for types at kind type
- `p`, `q` stand for types at kind perm
- `T`, `U` stand for types at any kind
- `A`, `C`, `D` stand for data constructors
- `e` stands for an expression
- `m` stands for a mode

6.3 Types and kinds

Types are classified using *kinds*.

`k ::= ...`

- `type` the kind of regular types,
- `perm` the kind of permissions,
- `value` the kind of program variables.

One may *bind* a type variable at a specific kind.

`B ::= ...`

- `X: k` annotated type binding,
- `a` unannotated type binding (default kind is `type`)

Here is the exhaustive list of all type constructs the user may write in Mezzo.

`t ::= ...`

- `X` reference to a type variable
- `[B] T` universal binding; kind depends on `T`; mode depends on `T`
- `{B} T` existential binding (same as above)
- `unknown` the top type at kind `type`; conveys no ownership information (duplicable)
- `dynamic` allows to test for ownership; has kind `type`; conveys no ownership information (duplicable)
- `A { \vec{t} : \vec{t} }, C, or D { \vec{t} : \vec{t} | p } structural (also referred to as “constructor”) type; has kind type; conveys ownership (duplicable or exclusive) of the block; conveys ownership of the fields according to their types \vec{t} ; p is a permission packaged along with the constructor; the last two forms are sugar for C { } and D { \vec{t} : \vec{t} } | p respectively`

- (\vec{t}) tuple type with at least two fields, $()$ tuple type with zero fields, i.e. unit type; has kind type; conveys duplicable ownership of the block which holds the tuple; conveys ownership of the fields according to their types \vec{t}
- $=x$ singleton type; has kind type; conveys no ownership information (duplicable)
- $T \bar{U}$ type application; has kind type or perm depending on the definition of t ; ownership information depending on the known fact for T
- $t \rightarrow u$ arrow type; has kind type; always duplicable
- `empty` the top type at kind perm (we have no “polymorphic” top type in Mezzo, so we have two separate top types for kinds type and perm); conveys no ownership information (duplicable)
- $x @ t$ anchored permission; has kind perm; ownership information depends on t
- $p * q$ conjunction of permissions; has kind perm; ownership information depends on p and q
- $t | p$ type / permission conjunction; has kind type; ownership information depends on t and p
- $t | m X$ mode constraint / type conjunction; kind depends on T ; ownership information depends on T
- $m X \Rightarrow t \rightarrow u$ mode implication; syntactic sugar for $(t | m X) \rightarrow u$, hence always duplicable
- $x: t$, name introduction; kind is type; ownership information depends on t ; this binding introduces a new name with special scoping rules (Chapter 7)
- `consumes T`, consumes annotation; has kind type or perm depending on T ; this construct has a special interpretation (Chapter 7)

6.4 Type definitions

type definition
fact?

A type definition is followed by an optional fact. A type definition is either a data type definition, an alias definition, or an abstract type definition.

Data type definitions are frequently used in Mezzo, as we saw through the various examples.

```
data mutable? d  $\vec{B}$  =
  branch+
  adopts?

and ...
```

- d is the name of the data type being defined
- \vec{B} is a possibly empty list of type bindings (the data type’s parameters); each binding may be preceded by $+$ (respectively $-$, $=$) to assert that the type is covariant (respectively invariant, bivariate) in this parameter
- `branch+` is a non-empty list of branches optionally preceded by, and separated by `|`
- `branch` is a data type branch; any combination of the following is admissible for a data type branch: quantified types, type / permission conjunction, name introduction, as long as they wrap around a constructor type. For instance, $(x: A \{ \dots \}), \{ a \} B \{ \dots \}, (A \{ \dots \} | \dots)$, are all valid branches.
- `adopts?` is an optional adopts clause, that is, either `adopts t` or nothing.

Valid data type definitions have been shown over the previous chapters (Figure 5.4, Figure 3.8, Figure 3.9, Figure 3.14).

Simultaneous definitions of data types are recursive by default. There is no `nonrec` keyword.

While the surface syntax allows for complex branch types, the formalization (Chapter 8) assumes that branches of data type definitions are plain concrete types of the form $A \{ \dots \}$. Handling more complex branch types adds no theoretical difficulty; the implementation of Mezzo uses a set of helpers to find the constructor A of a data type branch.

To simplify the code, one may define *type aliases*. These aliases are expanded eagerly; hence, they cannot be recursive. This means that they add no theoretical difficulty as, *unlike* in OCaml, they do not bring equirecursive types into the language. They receive no special treatment in the formalization and we assume them to be transparently expanded by the type-checker. Type aliases will therefore not be mentioned unless they receive some special treatment (such as variance or fact computation).

```
alias d  $\vec{B}$  ret? =
  T
```

- d is the name of the alias being defined;
- \vec{B} is a possibly empty list of variance-annotated type bindings;
- `ret?` is an optional return kind annotation, that is, either `: k` or nothing – the default for k is type;
- T is the definition of the alias, that is, any possible type.

Alias definitions have appeared previously (Figure 3.11, Figure 5.1). We also allow the user to write axioms using *abstract type* definitions.

```
abstract d  $\vec{B}$  ret?
```

- d is the name of the abstract type being defined;
- \vec{B} is a possibly empty list of variance-annotated type bindings;
- `ret?` is an optional return kind annotation, that is, either `: k` or nothing – the default for k is type;

Abstract definitions have appeared previously (Figure 4.1, Figure 4.8, Figure 4.9).

6.5 Modes and facts

We saw several examples of modes and facts throughout the examples. There are three *modes*.

```
m ::= ...
```

- `duplicable`
- `exclusive`
- `affine`

Facts build on modes to reveal information about the duplicability of abstract types; they can also be used to assert that a concrete definition (data or alias) verifies the intended fact.

```
fact  $\vec{h}$  => c
```

- \vec{h} is a possibly empty list of *hypotheses* separated by `=>`

- `h` is a fact hypothesis of the form `m X` where `X` is one of the type's parameters, that is, `X` is one of the variables from \vec{B}
- `c` is a fact conclusion of the form `d \vec{B}` , that is, the type applied to all its formal arguments.

An example of a fact can be found in the signature of our `mychannel` module (Figure 4.10):

```
abstract channel a
fact duplicable (channel a)
```

The mode is implicitly quantified over the formal parameters of the type it refers to. That is, the mode above is to be understood as “ $\forall a$, duplicable channel a ”. If the type of lists were abstract, it would enjoy a more complex fact:

```
abstract list a
fact duplicable a => duplicable (list a)
```

Again, this should be understood as “ $\forall a$, duplicable $a \Rightarrow$ duplicable list a ”. Facts have appeared previously (Figure 4.8, Figure 4.9).

6.6 Programs

Programs are written using *expressions*.

- ```
e ::= ...
```
- `e : t` type annotation; in theory, this is equivalent to `let x = e in assert x @ t; x`; however, in the implementation, the type annotation `t` is propagated downwards to help inference
  - `x` reference to a variable
  - `let p = e and ... and p = e in e` possibly multiple, *nonrecursive* value definitions;
  - `fun [ $\vec{X}$ :  $\vec{K}$ ] t : u = e` anonymous function; return kinds  $\vec{K}$  are optional; `t` is interpreted as a pattern (Chapter 7)
  - `let rec? f [ $\vec{X}$ :  $\vec{K}$ ] t : u = e and ... and f [ $\vec{X}$ :  $\vec{K}$ ] t : u = e in e` possibly multiple, possibly recursive *function* definitions<sup>1</sup>
  - `let flex B in e` flexible variable introduction with optional kind annotation
  - `let type_definition in e` local type definition (anything other than an alias definition should be considered experimental)
  - `x.f <- e` field assignment
  - `tag of e <- A` tag assignment
  - `e.f` field reference
  - `assert p` assertion
  - `pack {X: k} p witness t` pack a permission using an existential with the provided witness; alternatively, `pack T  $\vec{U}$  witness t` is allowed as long as `T  $\vec{U}$`  is an alias definition that expands to the first form;
  - `e e` function application
  - `e [T , ..., T ]` type application
  - `e [X=T]` if `e` is a polymorphic type with multiple variables, just instantiate the one named `X`
  - `match e with  $\vec{p} \rightarrow \vec{e}$`  pattern-matching
  - `(e , ..., e )` tuple construction ( $n \neq 1$ )
  - `A {  $\vec{f} = \vec{e}$ ; | p } , C` constructor expression

<sup>1</sup> Unlike OCaml, recursive *value* definitions are not allowed; these can be attained using progressive initialization.

- `if e then e else e` conditional
- `preserving p while e do e` while loop
- `preserving p for x = e to/downto/below/above e do e` for loop
- `e ; e` sequence
- `i` integer constant
- `give e to e` give ownership
- `take e from e` take ownership
- `e adopts e` test ownership
- `fail` run-time failure
- `builtin f` reference to a primitive function  $f$ , i.e. not implemented in Mezzo, to be provided by the runtime environment

Not all these expressions are taken into account in the formalization; for instance, simultaneous let-bindings are a syntactic convenience. They do not bring in extra theoretical difficulties; I'd rather not clutter the already-fairly-dense typing rules with provisions for `let ...` and bindings. Local type definitions are very experimental, and they are currently only used to define type aliases over type variables in scope. Binding flexible variables is a mere convenience and is not worth formalizing. Packing is also remarkably standard and is not formalized either. The semantics of our labeled type applications are dubious, so I'd rather not try to formalize them. Loops are desugared into recursive functions; I spare these boring transformations to the reader. Integers are actually axiomatized using:

```
abstract int
fact duplicable int
```

The parser parses integers, and the type-checker assigns the axiomatized `int` type to integer constants.

A word about the semantics of type annotations. Type annotations may *weaken* the current set of permissions. Consider, for instance, the example below, which involves mutable (uniquely-owned) lists.

```
val _ =
 let x = MNil in
 (* x @ MNil *)
 assert x @ mlist int
 (* x @ mlist int *)
```

The permission `x @ MNil` has been lost. The same goes for type annotations on expressions.

A word about field name overload. Mezzo, unlike classic OCaml, allows to use the same field name for multiple data types, without any restrictions. Anticipating slightly on Chapter 8, the rules involving field names (`READ`, `WRITE`) demand *structural permissions*, meaning that the type-checker can easily resolve the field access `x.f` by examining the permission available for `x`.

Mezzo also offers *pattern-matching*. Unlike OCaml, we do *not* offer or-patterns, that is, nested disjunctions. This would be hard to type-check because it would incur other instances of the *merge problem* (Chapter 12).

- ```
p ::= ...
```
- `x` binding pattern
 - `(p , ..., p)` tuple pattern ($n \neq 1$)
 - `A { $\vec{f} = \vec{p}$ } , C` constructor pattern
 - `p : t` type-annotated pattern
 - `p as x` conjunction pattern
 - `_` wildcard

We also offer some *syntactic sugar* related to constructor types.

- In patterns, types and expressions, `A { ...; f; ... }` is understood to be `A { ...; f = f; ... }`.
- In types, `A { ...; f:: t; ... }` is expanded to `{ f: value } A { ...; f: (=f | f @ t); ... }`
- In types, `A { ...; f, f': t; ... }` is expanded to `A { ...; f: t; f': t; ... }` (also works with `::`).

6.7 Module layer

Implementations

Implementations may contain any sequence of type definitions or value definitions `v`. Implementations are written in files with the extension `.mz`.

```
v ::= ...
  • val p = e and ... and p = e in e possibly multiple, nonrecursive toplevel value definitions;
  • val rec? f [X̄: K̄] t : u = e and ... and f [X̄: K̄] t : u = e in e possibly multiple, possibly recursive toplevel function definitions
```

Modules

Mezzo provides a primitive module system.

- The *implementation* for module `m` is to be written in `m.mz`
- If one wants to use the definitions from `m.mz` from another file, one *must* write an *interface*, in `m.mzi`
- The user may then use *qualified references* to the definitions exported by module `m`.
 - `m : x` is a qualified reference to the value `x`
 - `m : T` is a qualified reference to the type `T`
 - `m : A` is a qualified reference to the data constructor `A`

When type-checking a file `m.mz`, Mezzo first type-checks the implementation. Then, the compiler checks if `m.mzi` is found in the same directory. Should an interface for `m` be found, Mezzo checks the implementation *against* the interface.

When type-checking a module `n`, Mezzo first collects the references to external modules present in `n`. Mezzo then parses the *interfaces* only, and trusts that the interfaces are correct.

It is thus up to the user's build system to make sure that all modules which `n` refers to have been type-checked first.

Interfaces

Interfaces may contain any sequence of type definitions or value signatures `s`.

```
s ::=
  val x: t
```

Auto-loaded modules

Mezzo automatically loads the modules located in the `corelib/` directory of the distribution. These contain the basic definitions: integers, booleans, references...

7. Translating Mezzo to Simple Mezzo

The examples that we discussed so far are valid Mezzo code, expressed in the *surface syntax*. The multiple conventions related to ownership (the `consumes` keyword) as well as name binding (the $x : t$ construct) make it difficult, however, to formally reason about the semantics of the language. The type-checker of Mezzo thus translates the surface syntax (Mezzo) into its own internal language, which I dub Simple Mezzo. The internal language has none of the fancy features.

This dissertation hence faithfully describes what is currently implemented in the type-checker. Other related publications [BPP14a], including a to-be-submitted paper journal, deal with an even simpler language called Core Mezzo. This language is used to perform the proof of soundness and is not covered in this dissertation; §8.8 at the end of the next chapter lists the differences between Simple Mezzo and Core Mezzo.

We believe that the conventions related to ownership facilitate writing programs in Mezzo, and that many of the earlier examples would be much more heavyweight if not for the `consumes` keyword as well as the syntactic puns over name binding.

Simple Mezzo has a lot in common with Mezzo. Differences lie within the name introductions, of the form $x : t$, and the `consumes` keywords. These two constructs are removed when switching to Simple Mezzo. Name introductions are removed, along with the special scoping rules, and give way to regular \forall and \exists binders with traditional scoping rules. Function types also change: indeed, the surface arrow (\rightarrow in concrete syntax, \rightsquigarrow in math font) assumes its argument to be preserved, save for the parts which are marked with the `consumes` keyword. We do not wish to deal with this unusual convention when formalizing the language, and thus rewrite these external arrows into regular, internal arrows (\rightarrow in math font) that always consume their argument.

One important point to note is that, in this formal presentation, we do not introduce Mezzo and Simple Mezzo as separate languages. Rather, we work within the union of the two languages:

- initially, we start with types that live in the Mezzo subset;
- a first translation pass removes the name introductions, thus sending the types in a smaller subset of Mezzo;
- a second translation pass removes the `consumes` annotation and translates \rightsquigarrow into \rightarrow in one go, thus producing a type that belongs to the Simple Mezzo subset.

Having one common language greatly simplifies the presentation of kind-checking rules, as they operate on the union of the two languages. This allows us to claim that types remain well-kinded throughout the two translation phases.

As things get more formal, the typesetting changes. This chapter uses a math font except when showing code snippets. This means that $[x : k] t$ becomes $\forall(x : \kappa) t$, that $\{x : k\} t$ becomes $\exists(x : \kappa) t$, and that $t \rightarrow u$ becomes $t \rightsquigarrow u$. The complete syntax appears in Figure 7.1.

This chapter tries to remain in sync with the conventions from the previous chapter: t hints at a type at kind type while T hints at a type at kind perm or type. Since the presentation of types does not need to remain stable by substitution, I take the liberty of writing X whenever we know that only a type variable may occur.

$\kappa ::=$	type value perm $\kappa \rightarrow \kappa$	kind
$T, t, p ::=$		type or permission
	X	variable (a, x, \dots)
	(\vec{t})	tuple type
	$A \{\vec{f}: \vec{t}\}$ adopts t	constructor type
	$X \vec{T}$	n -ary type application
	$\forall (X : \kappa) T$	universal quantification
	$\exists (X : \kappa) T$	existential quantification
	$=x$	singleton type
	$(t p)$	type/permission conjunction
	$(t mX)$	mode constraint/type conjunction
	dynamic	dynamic permission
	unknown	top type
	$x @ t$	atomic permission
	empty	empty permission
	$p * q$	permission conjunction
<u>Only in Simple Mezzo :</u>		
	$t \rightarrow u$	function type (internal syntax)
<u>Only in Mezzo :</u>		
	$x : t$	name introduction
	consumes T	consumes annotation
	$t \rightsquigarrow u$	function type (surface syntax)

Figure 7.1: Syntax of types in Mezzo and Simple Mezzo

7.1 Examples

Let us consider the following type, which is a simplified version of the type of `find` (§3.2, Figure 3.11).

```
[a] (consumes xs: list a) -> (x: a, wand (x @ a) (xs @ list a))
```

The name introduction construct `xs: list a` binds the variable `xs`. The scope of `xs` encompasses the domain and codomain of this function type. There is, in fact, a free occurrence of `xs` in the codomain: `xs` occurs free in the permission `xs @ list a`.

The codomain of this function type is a pair (\dots, \dots) . The left-hand component of this pair is another name introduction construct `x: a`. The scope of `x` is the whole pair. There is, in fact, a free occurrence of `x` in the right-hand component of the pair: `x` occurs free in the permission `x @ a`.

One way of explaining the meaning of these name introduction constructs, and of making it clear where the names `xs` and `x` are bound, is to translate away the name introductions. In this example, this can be done as follows. This type is equivalent to the previous formulation, and is also valid surface syntax:

```
[a] [xs : value] (consumes (=xs | xs @ list a)) ->
  {x : value} ((=x | x @ a), wand (x @ a) (xs @ list a))
```

The name `xs` is now universally quantified (at kind `value`) above the function type. Thus, its scope encompasses the domain and codomain of the function type. The name `x` is existentially quantified (also at kind `value`) above the codomain. Thus, its scope is the codomain.

The name introduction `xs: list a` is now replaced with `(=xs | xs @ list a)`. This is a conjunction of a (singleton) type and a permission. This means that the function `find` expects a value (which is passed at runtime) and a permission (which exists only at type-checking time). Although placing a singleton type in the domain of a function type may seem absurdly restrictive, the universal quantification on `xs` saves the day. By instantiating `xs` with `ys`, one finds that, for any value `ys`, the call `find ys` is well-typed, provided the caller is able to provide the permission `ys @ list a`. Similarly, the name introduction `x: a` is replaced with `(=x | x @ a)`.

The encoding of dependent products and dependent sums in terms of quantification and singleton types is standard. It is worth noting that our name introduction form is more expressive than traditional dependent products and sums, as it does not have a left-to-right bias. For instance, in the type $(x: t, y: u)$, both of the variables x and y are in scope in both of the types t and u . If taken in isolation, this type would be translated as $\exists(x, y: \text{value}).((=x \mid x @ t), (=y \mid y @ u))$.

The function type we mentioned earlier can thus be easily translated into the internal syntax:

$$\forall(a: \text{type})\forall(xs: \text{value})(=xs \mid xs @ \text{list } a) \rightarrow \exists(x: \text{value})((=x \mid x @ a), \text{wand } (x @ a) (xs @ \text{list } a))$$

In this example, because the argument is entirely consumed, the translation is trivial. All we have to do is erase the consumes keyword. In the core syntax, by convention, the plain arrow \rightarrow denotes a function that consumes its argument, so this type has the desired meaning.

The translation of consumes is slightly more complex when only part of the argument is consumed: e.g., when the argument is a pair, one component of which is marked with the keyword consumes. Consider, for instance, the type of a function that merges two sets, updating its first argument and destroying its second argument:

```
val merge: [a] (set a, consumes set a) -> ()
```

The domain of this function type is a pair, whose second component is marked with the keyword consumes. We translate this into the core syntax by introducing a name, say x , for this pair, and by writing explicit pre- and postconditions that refer to x :

$$\forall(a: \text{type})\forall(x: \text{value})(=x \mid x @ (\text{set } a, \text{set } a)) \rightarrow ((\mid x @ (\text{set } a, \text{unknown}))$$

Informally, the caller regains the permission for the argument, except for the parts that were marked with consumes.

Thus, in order for the call `merge (s1, s2)` to be accepted, the caller must provide proof that $s1$ and $s2$ are valid sets; but, after the call, only $s1$ is known to still be a set. Here is how Mezzo type-checks a call to merge:

```

1 | let x = y, z in
2 | (* x @ (=y, =z) * y @ set a * z @ set a *)
3 | (* x @ (=y, =z) * x @ (set a, set a) *)
4 | merge x;
5 | (* x @ (=y, =z) * x @ (set a, unknown) *)
6 | (* x @ (=y, =z) * x @ (=y', =z') * y' @ set a * z' @ unknown *)
7 | (* x @ (=y, =z) * x @ (=y', =z') * y = y' * z = z' * y' @ set a * z' @ unknown *)
8 | (* x @ (=y, =z) * y @ set a * z @ unknown *)

```

After defining x to be the pair (y, z) at line 1, the type-checker introduces the expanded permission from line 2. Seeing that the call to merge demands $x @ (\text{set } a, \text{set } a)$, the type-checker seeks to obtain that permission; the permission $x @ (=y, =z)$, however, is duplicable and kept alongside by the type-checker (line 3). After the call to merge, a new permission is obtained (line 5). This permission is expanded as well (line 6). The type-checker then applies the UNIFYTUPLE subsumption rule (line 7):

$$x @ ((=y, =z)) * x @ ((=y', =z')) \Rightarrow x @ (=y, =z) * y = y' * z = z'$$

This means that the final permission (line 8) is the one we expected. We have $y @ \text{set } a$ after the call, which is precisely what we expected.



The signature of merge does not convey the fact that the identity of the tuple is kept, that is, that the first element of the tuple remains the same. Indeed, one may expect merge to have the following type, so as to preserve the permission for y , from the caller's point of view.

$$\forall(a: \text{type})\forall(x, y: \text{value})(=x \mid x @ (=y, \text{set } a) * y @ \text{set } a) \rightarrow ((\mid x @ (=y, \text{unknown}) * y @ \text{set } a)$$

Fortunately, thanks to UNIFYTUPLE, the equation $y = y'$ appears and guarantees that the permission $y @ \text{set } a$ is preserved.

The same mechanism is applied for immutable structural permissions. In the case of a mutable structural permission, however, the caller will be unable to preserve the structural permission, because it is affine. This means that, unless the author of the function uses a singleton type, the identity is “lost”.

$\text{names}(x : t)$	$=$	$(x, \text{value}) \uplus \text{names}(t)$	(Name introduction)
$\text{names}(\vec{t})$	$=$	$\uplus \text{names}(\vec{t})$	(Tuple type)
$\text{names}(A \{ \vec{f} : \vec{t} \} \text{ adopts } u)$	$=$	$\uplus \text{names}(\vec{t})$	(Constructor type)
$\text{names}(t \mid m X)$	$=$	$\text{names}(t)$	(Mode constraint/permission conjunction)
$\text{names}(t \mid p)$	$=$	$\text{names}(t)$	(Type/permission conjunction)
$\text{names}(\text{consumes } t)$	$=$	$\text{names}(t)$	(Consumes annotation)
$\text{names}(x @ t)$	$=$	$\text{names}(t)$	(Atomic permission)
$\text{names}(p * q)$	$=$	$\text{names}(p) \uplus \text{names}(q)$	(Permission conjunction)
$\text{names}(t)$	$=$	empty	(Any other type)

Figure 7.2: Name collection function

$$\frac{\text{K-OPENNEWSCOPE} \quad \Gamma; \text{names}(t)/c \vdash t : \kappa}{\Gamma/c \vdash \#t : \kappa}$$

Figure 7.3: Types and permissions: well-kindedness (auxiliary judgement)

```

val f1 [a] (Ref { contents: a }): () =
  ()

val f2 [a, contents: value] (Ref { contents } | contents @ a): () =
  ()

val test [a] (y: a): () =
  let x = Ref { contents = y } in
  f2 x;
  assert y @ a
  (* works, identity is preserved *)
  f1 x;
  assert y @ a
  (* fails, y @ a is gone, in favor of "x.contents @ a" *)

```

7.2 Kind-checking

Notations

In order to keep the kind-checking (and soon to follow, in Chapter 8, type-checking) rules concise, I use a vector notation. For instance, in K-TUPLE , the premise signifies that there is a sequence of hypotheses, for each element in the vector.

Checking types for proper scoping

The well-kindedness judgement checks (among other things) that every name is properly bound. Thus, in a slightly indirect way, it defines the scope of every name. In addition to the universal and existential quantifiers, which are perfectly standard, Mezzo offers the name introduction construct $x : t$, which is non-standard, since x is in scope not just in the type t , but also “higher up”, so to speak. For instance, in the type $(x_1 : t_1, x_2 : t_2)$, both x_1 and x_2 are in scope in both t_1 and t_2 . In order to reflect this convention, in the well-kindedness rules, one must at certain well-defined points go down and *collect* the names that are introduced by some name introduction form, so as to *extend* the environment with assumptions about these names.

The auxiliary function $\text{names}(T)$, which collects the names *introduced* by the type T , is defined in Figure 7.2. In short, it descends into tuples and constructors, looking for name introduction forms, and collects the names

$\frac{\text{K-VAR} \quad (x, \kappa) \in \Gamma}{\Gamma/c \vdash x : \kappa}$	$\text{K-UNKNOWN} \quad \Gamma/c \vdash \text{unknown} : \text{type}$	$\text{K-SING} \quad \frac{\Gamma/c \vdash x : \text{value}}{\Gamma/c \vdash =x : \text{type}}$	$\text{K-INTERNALARROW} \quad \frac{\Gamma/\bullet \vdash t_1 : \text{type} \quad \Gamma/\bullet \vdash t_2 : \text{type}}{\Gamma/\bullet \vdash t_1 \rightarrow t_2 : \text{type}}$
$\text{K-BAR} \quad \frac{\Gamma/c \vdash t : \text{type} \quad \Gamma/c \vdash \#p : \text{perm}}{\Gamma/c \vdash (t \mid p) : \text{type}}$	$\text{K-TUPLE} \quad \frac{\Gamma/c \vdash \vec{t} : \text{type}}{\Gamma/c \vdash (\vec{t}) : \text{type}}$	$\text{K-CONCRETE} \quad \frac{\Gamma/c \vdash \vec{t} : \text{type} \quad \Gamma/c \vdash \#u : \text{type} \quad u \text{ is exclusive}}{\Gamma/c \vdash A \{\vec{f} : \vec{t}\} \text{ adopts } u : \text{type}}$	
$\text{K-ATOMIC} \quad \frac{\Gamma \vdash x : \text{value} \quad \Gamma/\bullet \vdash t : \text{type}}{\Gamma/c \vdash x @ t : \text{perm}}$	$\text{K-EMPTY} \quad \Gamma/c \vdash \text{empty} : \text{perm}$	$\text{K-CONJUNCTION} \quad \frac{\Gamma/c \vdash p : \text{perm} \quad \Gamma/c \vdash q : \text{perm}}{\Gamma/c \vdash p * q : \text{perm}}$	
$\text{K-QUANTIFIER} \quad \frac{\Gamma; (x, \kappa')/\bullet \vdash \#T : \kappa \quad \Gamma/c \vdash \forall(x : \kappa') T : \kappa \quad \Gamma/c \vdash \exists(x : \kappa') T : \kappa}{\Gamma/c \vdash \forall(x : \kappa') T : \kappa}$	$\text{K-EXTERNALARROW} \quad \frac{\Gamma; \text{names}(t_1)/\circ \vdash t_1 : \text{type} \quad \Gamma; \text{names}(t_1)/\bullet \vdash \#t_2 : \text{type}}{\Gamma/c \vdash t_1 \rightsquigarrow t_2 : \text{type}}$	$\text{K-NAMEINTRO} \quad \frac{\Gamma \vdash x : \text{value} \quad \Gamma/c \vdash t : \text{type}}{\Gamma/c \vdash (x : t) : \text{type}}$	$\text{K-CONSUMES} \quad \frac{\Gamma/\bullet \vdash T : \kappa \quad \kappa \in \{\text{type}, \text{perm}\}}{\Gamma/\circ \vdash \text{consumes } T : \kappa}$
$\text{K-AND} \quad \frac{\Gamma \vdash a : \kappa_1 \quad \kappa_1 \in \{\text{type}, \text{perm}\} \quad \Gamma/c \vdash T_2 : \kappa_2}{\Gamma/c \vdash (T_2 \mid m a) : \kappa_2}$		$\text{K-APP} \quad \frac{\Gamma \vdash X : \vec{\kappa} \rightarrow \kappa \quad \Gamma/\bullet \vdash \#\vec{T} : \vec{\kappa}}{\Gamma/c \vdash X \vec{T} : \kappa}$	

Figure 7.4: Types and permissions: well-kindedness

$\text{K-DATADefinition} \quad \frac{\text{data } X (\vec{Y} : \vec{\kappa}) = \vec{t} \quad \Gamma, (\vec{Y} : \vec{\kappa})/\bullet \vdash \#\vec{t} : \text{type}}{\Gamma \vdash X : \vec{\kappa} \rightarrow \text{type}}$	$\text{K-ALIASDefinition} \quad \frac{\text{alias } X (\vec{Y} : \vec{\kappa}) : \kappa = T \quad \Gamma, (\vec{Y} : \vec{\kappa})/\bullet \vdash \#T : \kappa}{\Gamma \vdash X : \vec{\kappa} \rightarrow \kappa}$	$\text{K-ABSTRACTDefinition} \quad \frac{\text{abstract } X (\vec{Y} : \vec{\kappa}) : \kappa}{\Gamma \vdash X : \vec{\kappa} \rightarrow \kappa}$
--	---	--

Figure 7.5: Types definitions: well-kindedness

$$\text{K-FUN} \quad \frac{\Gamma \vdash [\vec{X} : \vec{\kappa}] t_1 \rightarrow t_2 : \text{type} \quad \Gamma, (\vec{X} : \vec{\kappa}), \text{names}(t_1) \vdash e}{\Gamma \vdash \text{fun } [\vec{X} : \vec{\kappa}] t : u = e}$$

Figure 7.6: Expressions: well-kindedness

that they introduce. These names always have kind value. The names function uses a disjoint union \uplus , meaning that any name that is bound twice is an error.

The well-kindedness judgement takes the form $\Gamma/c \vdash T : \kappa$. It means that under the kind assumptions in Γ , the type T has kind κ . The extra parameter c is explained later on. The definition of the judgement (Figure 7.4) relies on an auxiliary judgement, $\Gamma/c \vdash \#T : \kappa$, which is just an abbreviation for $\Gamma; \text{names}(T)/c \vdash T : \kappa$ (Figure 7.3). Intuitively, $\#$ is a “beginning-of-scope” mark: it means that, at this point, the names collected by the auxiliary function names are in scope.

This mark does not explicitly exist in the surface syntax, but is introduced on the fly, so to speak, by the well-kindedness rules, at certain conventional points.

The names function and the kind-checking rules are related. Intuitively, the result that we wish to state is that, when kind-checking, names are “properly bound”. Defining what “properly bound” means is hard, though, because the binding structure of Mezzo is unusual, and because the question is in a sense very much a design decision for the language. There is no such thing as “being correct” for the translation: it is, after all, a translation, meaning that it *defines* what the surface syntax means. There are, however, a few ways to check that our translation means something.

A way of convincing ourselves that our definitions make sense is to ensure neither *too few* or *too many* names are introduced. Having too few binders would mean that an occurrence of $x : t$ failed to be accounted for; having too many binders would mean that names introduced a binder that has no use.

Remark 7.1. *The names function, when called at kind type, only ever collects names that correspond to an irrefutable pattern. I call them irrefutable names.*

Seeing a type as a syntactical tree, the function starts from the root of the value, and descends recursively into the sub-components of the value. Tuple and constructor fields, left-hand sides of conjunctions ($t \mid p$) correspond to sub-components of the value, and are visited. Arguments of type applications may not exist, hence are not irrefutable: they are not visited. The same goes for the right-hand sides of conjunctions ($t \mid p$), which have no existence at run-time, hence do not correspond to a pattern.

As an example, descending into, say, parameters of type applications (K-APP) makes no sense: in list $(x : a)$, there may be zero, one, or several elements which x refers to: the existence of $y @ \text{list } x : a$ does not imply the existence of a unique value x at type a . It thus makes no sense to bind x above the type application. The same goes for arrows: the domain of an arrow cannot be matched via a pattern, but rather refers to the future arguments that the function will receive. A name introduced within the domain of an arrow should thus not be collected by the names function.

The kind-checking rules and the names function go hand-in-hand. The following two remarks clarify this informal statement.

Remark 7.2. *Every name introduced in the kind-checking rules via names is used to satisfy the premise in K-NAMEINTRO exactly once.*

Remark 7.3. *As a consequence, when kind-checking, we always visit a value t with $\text{names}(t) \subseteq \Gamma$, meaning that the first premise in K-NAMEINTRO is always satisfied and is therefore redundant.*

Proof. The kind-checking rules for checking t instrument the names function to make it cover the entire syntactic tree representing t .

For each construct that names does not traverse (parameters of type applications, quantifiers, right-hand side of conjunctions, arrows), the corresponding kind-checking rules calls names on the sub-components. For each construct that names does traverse (tuples, constructors, right-hand side of conjunctions), the kind-checking rules do not call names on the sub-components.

This means that assuming the traversal starts with $\vdash \#$, then names visits each node of the syntactic tree that represents t exactly once. ☺

The fact that names does not descend into quantifiers (meaning that K-QUANTIFIER takes care of using $\vdash \#$ which, in turn, calls names) may seem arbitrary. It is, indeed, a design choice that we made. It seemed to us that, from a design perspective, this behavior would be slightly un-intuitive.

K-EXTERNALARROW embodies the convention we described informally in the earlier example (§7.1). Names which appear in the domain are bound within both the domain and the codomain; conversely, names introduced

in the codomain can be referred to only in the codomain. Again, this makes intuitive sense: one cannot talk about the return value of a function call before the function has run.

Other checks for types

There are additional restrictions that the well-kindedness rules should impose: for instance, the `consumes` keyword should appear only in the left-hand side of an arrow, and should not appear under another `consumes` keyword. This can be expressed by extending the well-kindedness judgement with a Boolean parameter, which indicates whether `consumes` is allowed or disallowed.

The \bullet parameter indicates that `consumes` annotations are not allowed; the \circ parameter indicates that `consumes` annotations are allowed. Initially (`K-OPENNEWSCOPE`), `consumes` annotations are forbidden. Only when moving to the left-hand side of an *external* arrow (`K-EXTERNALARROW`) does the parameter change to \circ . When encountering a `consumes` keyword (`K-CONSUMES`), the parameter *must* be \circ . `K-CONSUMES` also encodes the fact that no nested `consumes` annotations are allowed.

In order to reduce clutter, all other rules are assumed to work indifferently with \circ or \bullet ; the rules are also assumed to pass the extra parameter unmodified to their premises.

Another point that kind-checking rules enforce is that all type applications should be complete (`K-APP`): Mezzo does not allow a type constructor to be partially applied. The rule also asserts that the type being applied is a type variable: Mezzo does not allow higher-kinded types. The only form of arrow kinds is thus $\vec{\kappa}_0 \rightarrow \kappa_0$ (where κ_0 is one of the base kinds), and the only types which possess such arrow kinds are user-defined types (`K-DATATYPEDEFINITION`).

Other standard well-formedness checks are performed, such as making sure all the fields of a constructor type exist, or that data constructors are well-bound. These checks are boringly standard and I thus omit them.



Field names can be overloaded; that is, multiple data constructors A and B may share a field name f . The user must make sure, however, that both the types and the expressions they write make sense. The kind-checking phase, for every structural type $A \{ \vec{f} : \vec{t} \}$:

- looks up the definition of constructor A ;
- makes sure that the fields \vec{f} are exactly the fields mentioned in the definition, modulo the order they appear in;
- normalizes this type to make sure that the fields \vec{f} are in the same order as the definition.

In expressions, the kind-checking phase ensures that:

- every constructor expression $A \{ \vec{f} = \vec{e} \}$ contains exactly the fields from the definition of A ;
- for every constructor pattern $A \{ \vec{f} : \vec{p} \}$, the fields f exist in the definition of A .

The lookup of A is un-ambiguous since data constructors, unlike fields, cannot be overloaded: they are lexically scoped and defining a data constructor by the same name as an existing one masks the previous definition.

Duplicate field names are also ruled out by the kind-checking phase.

Checking expressions

For expressions (Figure 7.7), we adopt the same approach, which is to put Mezzo and Simple Mezzo in the same syntactic category, and single out the constructions that can only appear in one of the two subsets.

Kind-checking expressions is remarkably standard and is not shown here: it mostly consists in making sure that names are well-bound, along with the other checks related to field names I mentioned above. Types that appear in expressions (assertions, type annotations, type applications) are checked using $\Gamma/\bullet \vdash \#$.

The only interesting case is that of the external function. The syntax takes a type for an argument, hence performing a “pun”: the type is either interpreted as a type (when determining the type of the function, when calling the function) or as a pattern (when type-checking the function body). The formal rule is presented in Figure 7.6, where $\Gamma \vdash e$ just means “ e is well-kinded”. Let us take some time to comment this rule.

Argument as a type We interpret the function argument as a type for kind-checking, and for assigning a type to the function itself. If the external function is:

$$\text{fun } [\vec{X} : \vec{\kappa}] t : u = e$$

$e ::=$		expression
x		variable
$e : t$		type annotation
let $p = e$ in e		local definition
$e [t : \kappa]$		type instantiation
$e e$		function application
(\vec{e})		tuple (not unary)
$A \{\vec{f} = \vec{e}\}$		data constructor application
$e.f$		field access
$e.f \leftarrow e$		field update
match e with $\vec{p} \rightarrow \vec{e}$		case analysis
tag of $e \leftarrow A$		tag update
assert p		assert permission p
if e then e else e		conditional
give e to e		adoption
take e from e		abandon
e adopts e		ownership test
fail		dynamic failure
<u>Only in Simple Mezzo :</u>		
$\lambda(x : t) : t. e$		<i>internal</i> anonymous function
$\Lambda(X : \kappa). e$		type abstraction
<u>Only in Mezzo :</u>		
$e; e$		sequence
fun $[\vec{X} : \vec{\kappa}] t : t = e$		<i>external</i> anonymous function
$p ::=$		pattern
x		variable
(\vec{p})		tuple pattern
$A \{\vec{f} = \vec{p}\}$		data constructor pattern
p as x		conjunction pattern
$p : t$		type annotation
—		wildcard pattern

Figure 7.7: Syntax of expressions

then we first need to check that t and u are both well-kinded. We follow the same conventions that prevail for types, and check the validity of $\forall(\vec{X} : \vec{\kappa}) t \rightsquigarrow u$.

Argument as a pattern Abusing the $x : t$ notation for types, we understand the name introduction construct to also introduce a name whose scope is the argument, the return type, as well as *the entire function body*. This interpretation stems from the similarity of the name introduction construct with the classic notation for function arguments. More formally, name introductions correspond to components of the argument that have a unique existence at run-time, hence can be referred to in the function body.

This interpretation of a type as a pattern is governed by the `t2p` function (Figure 7.8). The names bound by t in e are `names(t)`. This is embodied by `K-FUN`.

Later on, the translation `T-FUN` makes this pun explicit by converting the external function definition into an internal one that uses an actual pattern.

Checking data type definitions

Data types are also subject to a few checks, in order to make sure that branches are well-formed constructor types. The checks are listed in Figure 7.5.

$$\begin{array}{lcl}
\text{t2p}(x : t) & = & \text{t2p}(t) \text{ as } x \\
\text{t2p}(\vec{t}) & = & \overrightarrow{(\text{t2p}(t))} \\
\text{t2p}(A \{\vec{f} : \vec{t}\} \text{ adopts } u) & = & A \{\vec{f} = \text{t2p}(\vec{t})\} \\
\text{t2p}(t \mid m a) & = & \text{t2p}(t) \\
\text{t2p}(t \mid p) & = & \text{t2p}(t) \\
\text{t2p}(\text{consumes } t) & = & \text{t2p}(t) \\
\text{t2p}(t) & = & -
\end{array}$$

Figure 7.8: Type-to-pattern function

$$\frac{\text{T1-OPENNEWSCOPE} \quad T \blacktriangleright T'}{\#T \blacktriangleright \exists(\text{names}(T)) T'}$$

Figure 7.9: Types and permissions: first translation phase (auxiliary judgement)

$$\begin{array}{c}
\begin{array}{ccccc}
\text{T1-VAR} & \text{T1-UNKNOWN} & \text{T1-SING} & \text{T1-BAR} & \text{T1-TUPLE} \\
x \blacktriangleright x & \text{unknown} \blacktriangleright \text{unknown} & =x \blacktriangleright =x & \frac{T \blacktriangleright T' \quad \#P \blacktriangleright P'}{(T \mid P) \blacktriangleright (T' \mid P')} & \frac{\vec{T} \blacktriangleright \vec{T}'}{(\vec{T}) \blacktriangleright (\vec{T}')}
\end{array} \\
\begin{array}{ccccc}
\text{T1-CONCRETE} & & \text{T1-ATOMIC} & \text{T1-EMPTY} & \text{T1-CONJUNCTION} \\
\frac{\vec{T} \blacktriangleright \vec{T}' \quad \#U \blacktriangleright U'}{A \{\vec{f} : \vec{T}\} \text{ adopts } U \blacktriangleright A \{\vec{f} : \vec{T}'\} \text{ adopts } U'} & & \frac{T \blacktriangleright T'}{x @ T \blacktriangleright x @ T'} & \text{empty} \blacktriangleright \text{empty} & \frac{P_1 \blacktriangleright P'_1 \quad P_2 \blacktriangleright P'_2}{P_1 * P_2 \blacktriangleright P'_1 * P'_2}
\end{array} \\
\begin{array}{ccccc}
\text{T1-QUANTIFIER} & & \text{T1-EXTERNALARROW} & & \text{T1-NAMEINTRO} \\
\frac{\#T \blacktriangleright T'}{\forall(x : \kappa) T \blacktriangleright \forall(x : \kappa) T' \quad \exists(x : \kappa) T \blacktriangleright \exists(x : \kappa) T'} & & \frac{T_1 \blacktriangleright T'_1 \quad \#T_2 \blacktriangleright T'_2}{T_1 \rightsquigarrow T_2 \blacktriangleright \forall(\text{names}(T_1)) T'_1 \rightsquigarrow T'_2} & & \frac{T \blacktriangleright T'}{x : T \blacktriangleright (=x \mid x @ T')}
\end{array} \\
\begin{array}{ccccc}
\text{T1-CONSUMES} & & \text{T1-AND} & & \text{T1-APP} \\
\frac{T \blacktriangleright T'}{\text{consumes } T \blacktriangleright \text{consumes } T'} & & \frac{T \blacktriangleright T'}{(T \mid m a) \blacktriangleright (T' \mid m a)} & & \frac{\#\vec{T} \blacktriangleright \vec{T}'}{X \vec{T} \blacktriangleright X \vec{T}'}
\end{array}
\end{array}$$

Figure 7.10: Types and permissions: first translation phase

7.3 Translation

We now define the translation of (well-kinded) types and permissions from the surface syntax into the core syntax. For greater clarity, we present it as the composition of two phases. In the implementation, the two phases can be merged.

In the first phase, we eliminate the name introduction construct. In the second phase, we transform the external function type into its internal counterpart, and at the same time eliminate the consumes construct.

Phase 1

The first phase is described by the translation judgement $T \blacktriangleright T'$, whose definition (Figure 7.10) relies on the auxiliary judgement $\#T \blacktriangleright T'$ (Figure 7.9). Both T and T' live within the external syntax subset; T' , however, no longer contains name introductions.

Remark 7.4. *The translation rules mirror the kind-checking rules: whenever kind-checking uses $\vdash \#$, the translation*

$$\begin{array}{c}
\text{T2-EXTERNALARROW} \\
\frac{T_1 \triangleright T'_1 \quad T_2 \triangleright T'_2 \quad t_1^{\text{in}} = [T/\text{consumes } T]T'_1 \quad t_1^{\text{out}} = [?/\text{consumes } T]T'_1}{T_1 \rightsquigarrow T_2 \triangleright \forall(x : \text{value}) (=x \mid x @ t_1^{\text{in}}) \rightarrow (T'_2 \mid x @ t_1^{\text{out}})}
\end{array}$$

Figure 7.11: Types and permissions: second translation phase (only one rule shown)

uses # \blacktriangleright . Thus, the translation materializes the implicit name-binding convention by introducing explicit quantifiers, i.e. “the translation is faithful to the kind-checking rules”.

The main rules of interest are T1-OPENNEWSCOPE, which introduces explicit existential quantifiers for the names whose scope begins at this point; T1-EXTERNALARROW, which introduces explicit universal quantifiers, above the function arrow, for the names introduced by the domain of the function; and T1-NAMEINTRO, which translates a name introduction form to a conjunction of a singleton type $=x$ and a permission $x @ T'$. The two occurrences of x in this conjunction are free: they refer to a quantifier that must have been introduced higher up by T1-OPENNEWSCOPE or T1-EXTERNALARROW.

Lemma 7.5. *Well-kindedness is preserved by the first translation phase:*

- $\Gamma/c \vdash T : \kappa$ and $T \blacktriangleright T'$ imply $\Gamma/c \vdash T' : \kappa$, where c stands for either \bullet or \circ .
- $\Gamma/c \vdash \#T : \kappa$ and $\#T \blacktriangleright T'$ imply $\Gamma/c \vdash T' : \kappa$.

In the conclusion, \vdash or $\vdash \#$ are the same, since there are no name intros.

Lemma 7.6. *If $T \blacktriangleright T'$ or $\#T \blacktriangleright T'$ holds, then T' does not contain a name introduction construct.*

Phase 2

The second phase is described by the translation judgement $t \triangleright t'$. t does not contain name introductions, but remains within the external syntax. t' no longer contains name introductions nor external arrows; it may still contain consumes annotations where allowed (per the kind-checking rules). Thus, when \triangleright is called at top-level, where consumes annotations are not allowed (K-OPENNEWSCOPE), it produces a type which no longer contains name introductions, consumes keywords or external arrows, that is, a type that belongs to the internal syntax.

The definition of the judgement appears in Figure 7.11. Only one rule is shown, as the other rules (omitted) simply encode a recursive traversal. The rule T2-EXTERNALARROW does several things at once.

First, it transforms an external arrow \rightsquigarrow into an internal arrow \rightarrow . Second, it introduces a fresh name, x , which refers to the argument of the function; this is imposed by the singleton type $=x$. This name is universally quantified above the arrow; this is the same idiom that we would have used to eliminate a name introduction form that appears at the root of the function domain. Finally, in order to express the meaning of the consumes keywords that may appear in the type t'_1 , it constructs distinct pre- and postconditions, namely $x @ t_1^{\text{in}}$ and $x @ t_1^{\text{out}}$. These permissions respectively represent the properties of x that the function requires (prior to the call) and ensures (after the call).

The type t_1^{in} is defined as $[t/\text{consumes } t]t'_1$. By this informal notation, we mean “a copy of t'_1 , where every subterm of the form consumes t is replaced with just t ”, or in other words, “a copy of t'_1 , where every consumes keyword is erased”.

The type t_1^{out} is defined as $[?/\text{consumes } t]t'_1$. By this informal notation, we mean “a copy of t'_1 , where every subterm of the form consumes t is replaced with either unknown (at kind type) or empty (at kind perm), as appropriate”. These two types are our \top types for their respective kinds.

Thus, the permission $x @ t_1^{\text{in}}$ represents the ownership of the argument, including the components marked with consumes, whereas the permission $x @ t_1^{\text{out}}$ represents the ownership of the argument, deprived of these components. In other words, we take a permission for the “full” argument, and return a modified permission for the argument, where all parts marked consumed have been “carved out”.

$$\begin{array}{c}
\text{T-FUN} \\
\frac{t \rightsquigarrow u \# \blacktriangleright \forall (\vec{X}' : \vec{\kappa}') t' \rightarrow u' \quad p = \text{t2p}(t) \quad e \triangleright e'}{\text{fun } [\vec{X} : \vec{\kappa}] t : u = e \triangleright \\ \Lambda(\vec{X} : \vec{\kappa}). \Lambda(\vec{X}' : \vec{\kappa}'). \lambda(x : t') : u'. \text{let } p = x \text{ in } e'}
\end{array}$$

Figure 7.12: Expressions: translation

Remark 7.7. T'_2 does not contain consumes annotations. Indeed, $K\text{-EXTERNALARROW}$ uses $/\bullet$ for checking T_2 , meaning that consumes annotations are ruled out. T'_1 , conversely, may contain consumes annotations. These annotations only appear in specific positions, though. For instance, if the argument T_1 is a tuple, one of the components of the tuple may be of the form consumes u . However, if the T_1 is of the form list u , u may not contain any consumes keyword, as it would make no sense.

A way to understand this is to think of consumes keywords are being only allowed for sub-components of the function's argument. One can consume a tuple component, or a record component. In the list u example above, one cannot consume u , as it may not exist (Nil case) or as they may be several such u .

Lemma 7.8. Well-kindedness is preserved by the second translation phase: assuming that t contains no name introduction forms, $\Gamma/c \vdash t : \kappa$ and $t \triangleright t'$ imply $\Gamma/c \vdash t' : \kappa$.

Lemma 7.9. Assuming t does not contain name introductions, if $t \triangleright t'$ and $\Gamma/\bullet \vdash t : \kappa$ hold, then t' contains no external arrow \rightsquigarrow and no consumes keyword.

Proof. Per the definition of \triangleright , t' contains no external arrows. Per Lemma 7.8, $\Gamma/\bullet \vdash t' : \kappa$. Since t' no longer contains external arrows, the \bullet parameter in the kind-checking rules will never switch to \circ , meaning that t' no longer contains consumes keywords. ☹

Lemma 7.10. $\Gamma/\bullet \vdash \#t : \kappa, \#t \blacktriangleright t'$ and $t' \triangleright t''$ imply that t'' is in the internal syntax and $\Gamma \vdash t'' : \kappa$.

Proof. Composition of Lemma 7.5 and Lemma 7.8 for the well-kindedness conclusion, and of Lemma 7.6 and Lemma 7.9 for the internal syntax conclusion. ☹

Translating data type definitions

Data type definitions are also translated via $\# \blacktriangleright$ followed by \triangleright .

Translating expressions

Expressions are translated in one phase, and rely on the combination of the two translations on types, which we write as $\# \blacktriangleright$. We write $e \triangleright e'$ for translating e into e' .

Minor translations are performed: for instance, $e; e'$ is translated into $\text{let } () = e \text{ in } e'$. This is a mere matter of convenience. Types that occur inside expressions (type annotations, assertions, type applications) are translated with $\# \blacktriangleright$.

Just like kind-checking, the translation of functions is more interesting and is shown in Figure 7.12. Again, the argument type is paired with the return type so as to be translated like a regular function type. We obtain another function type with some universal quantifiers. Both user-introduced quantifiers (that is, \vec{X}) and translation-introduced quantifiers (that is, \vec{X}') are made explicit with Λ -abstractions.

The type-as-argument convention is made explicit using the type-to-pattern conversion (Figure 7.8) which we mentioned earlier; this allows us to use a classic, one-argument λ -abstraction followed by a let-pattern binding. In the internal syntax of expressions, just like in the internal syntax of types, λ -abstractions consume their argument.

Let us take an example to illustrate this convention. The left projection for a pair `proj1` is defined as follows.

```
val proj1 [a, b] (consumes x: a, y: b): a =
  x
```

This is implicitly converted as:

```
val proj1 = fun [a, b] (consumes x : a, y : b) : a =
  x
```

The function-expression, in math font, is:

$$\text{fun } [a, b]. (\text{consumes } x : a, y : b) : a = x$$

The function type $(\text{consumes } x : a, y : b) \rightsquigarrow a$ is translated as:

$$\forall (r, x, y : \text{value}). (=r \mid r @ ((=x \mid x @ a), (=y \mid y @ b))) \rightarrow (a \mid r @ (\text{unknown}, (=y \mid y @ b)))$$

The conversion of the argument into a pattern gives:

$$\text{t2p}(\text{consumes } x : a, y : b) = (x, y)$$

The entire function is thus translated as:

$$\Lambda(a, b) \Lambda(r, x, y : \text{value}). \\ \lambda(r' : (=r \mid r @ ((=x \mid x @ a), (=y \mid y @ b)))) : (a \mid r @ (\text{unknown}, (=y \mid y @ b))) = \\ \text{let } x', y' = r' \text{ in } x'$$

This makes the pun completely explicit, and introduces fresh names for the variables r' , x' and y' , which are expression variables, not just type variables.

The first thing that the type-checker does, upon entering the function body, is to assume the following permission:

$$r' @ (=r \mid r @ ((=x \mid x @ a), (=y \mid y @ b)))$$

which is rewritten into:

$$r @ (=x, =y) * x @ a * y @ b * r = r'$$

After that, writing $\text{let } x', y' = r' \text{ in } \dots$ simply adds two new equations $x = x'$ and $y = y'$ into the context. The pun is complete.

This “extended” scope for the type variables that, beyond the argument, may also appear in the function’s body, bears some resemblance to the “explicit-forall” extension of Haskell, where a type variable, even if only bound in the type annotation, may also appear in the function body later on.

```
f :: forall a. a -> a
f = \x: a -> x
```

8. Type-checking Simple Mezzo

The previous chapter defined the translation from Mezzo to Simple Mezzo. We are now ready to (finally!) state the typing rules of Simple Mezzo.

There are two essential judgements: the first one is permission subsumption (§8.2), which defines which operations are legal for transforming a permission into another. It may be understood as our subtyping relation for permissions. The second one is the type-checking judgement for expressions (§8.3).

The subsumption and type-checking rules frequently rely on a “is duplicable” predicate; this predicate is defined later on (§8.4). Some rules also rely on a notion of variance. Variance is standard and I write a little bit about it in this chapter as well (§8.6).

(As a side note, from now on, the # symbol disappears and # is understood to mean “length of”.)

As I mentioned earlier in the introduction, the formalization and the metatheory of Mezzo are not covered in this dissertation, but in a separate journal paper [BPP14a].

8.1 Memory model

Before introducing the type-checking rules, I believe the memory model of Mezzo deserves a brief presentation. Mezzo is compiled, as I mentioned already, into OCaml; therefore, our memory model is very close to that of OCaml.

Values are tagged; they are either 31-bit integers, or pointers to memory blocks. A tuple (x_1, \dots, x_n) is represented as a block of size n where the i -th field contains the value x_i . A constructor $A \{\vec{f} = \vec{x}\}$ is represented as a memory block of size $n + 2$, where the 0-th field contains the integer associated to constructor A , the next field contains the “hidden field” required for the adoption/abandon mechanism, while the subsequent fields contain the values \vec{x} . A constructor B with no fields is represented as a memory block of size two, where the first field corresponds to the constructor and the second field is used for the adoption/abandon mechanism. Currently, all data types are assigned an extra, hidden field, even if immutable. We plan to optimize this in the future. The first constructor field may be mutated; this operation is supported by the OCaml run-time system.

Just like in OCaml, we assume memory management to be handled by a garbage-collector.

This memory model differs from that of OCaml ([LDF⁺14], §19.3):

- in OCaml, tags of constructor blocks may not be mutated;
- in OCaml, a constant constructor B is represented as an immediate integer *value*,
- in OCaml, there is no hidden field.

REFLEXIVE $P \leq P$	TRANSITIVE $\frac{P_1 \leq P_2 \quad P_2 \leq P_3}{P_1 \leq P_3}$	EMPTYTOP $P \leq \text{empty}$	EMPTYAPPEARS $P \equiv \text{empty} * P$	STARCOMMUTATIVE $P_1 * P_2 \equiv P_2 * P_1$
STARASSOCIATIVE $P_1 * (P_2 * P_3) \equiv (P_1 * P_2) * P_3$	EQUALITYREFLEXIVE $\text{empty} \leq (x = x)$	EQUALSFOREQUALS $(y_1 = y_2) * [y_1/x]P \equiv (y_1 = y_2) * [y_2/x]P$		
COPYDUP $\frac{P \text{ is duplicable}}{C[t] * P \leq C[(t P)] * P}$	HIDEDUPLICABLEPRECONDITION $\frac{P \text{ is duplicable}}{(x @ (t_1 P) \rightarrow t_2) * P \leq x @ t_1 \rightarrow t_2}$	MIXSTAR $x @ t * P \equiv x @ (t P)$		
EXISTSINTRO $[T/X]P \leq \exists(X : \kappa) P$	EXISTSSTAR $P_1 * \exists(X : \kappa) P_2 \equiv \exists(X : \kappa) (P_1 * P_2)$	EXISTSATOMIC $\begin{aligned} x @ \exists(X : \kappa) t \\ \equiv \exists(X : \kappa) (x @ t) \end{aligned}$		
FORALLELIM $\forall(X : \kappa) P \leq [T/X]P$	FORALLSTAR $P_1 * \forall(X : \kappa) P_2 \equiv \forall(X : \kappa) (P_1 * P_2)$	FORALLATOMIC $\begin{aligned} x @ \forall(X : \kappa) t \\ \equiv \forall(X : \kappa) (x @ t) \end{aligned}$		
DECOMPOSETUPLE $\begin{aligned} & y @ (\dots, t, \dots) \\ \equiv \exists(x : \text{value}) & (y @ (\dots, =x, \dots) * x @ t) \end{aligned}$	DECOMPOSEBLOCK $\begin{aligned} & y @ A \{F[f : t]\} \text{ adopts } u \\ \equiv \exists(x : \text{value}) & (y @ A \{F[f = x]\} \text{ adopts } u * x @ t) \end{aligned}$			
FOLD $\frac{A \{\vec{f} : \vec{t}\} \text{ adopts } u \text{ is an unfolding of } X \vec{T}}{x @ A \{\vec{f} : \vec{t}\} \text{ adopts } u \leq x @ X \vec{T}}$	UNFOLD $\frac{A \{\vec{f} : \vec{t}\} \text{ adopts } u \text{ is an unfolding of } X \vec{T} \quad X \vec{T} \text{ has only one branch}}{x @ X \vec{T} \leq x @ A \{\vec{f} : \vec{t}\} \text{ adopts } u}$		UNKNOWNAPPEARS $\text{empty} \leq x @ \text{unknown}$	
DYNAMICAPPEARS $\frac{t \text{ is exclusive}}{x @ t \leq x @ t * x @ \text{dynamic}}$	CoTUPLE $\frac{\vec{t} \leq \vec{u}}{x @ (\vec{t}) \leq x @ (\vec{u})}$	CoBLOCK $\frac{\vec{t} \leq \vec{u} \quad t \leq u}{x @ A \{\vec{f} : \vec{t}\} \text{ adopts } t \leq x @ A \{\vec{f} : \vec{u}\} \text{ adopts } u}$		
CoSTAR $\frac{P_1 \leq P_2 \quad Q_1 \leq Q_2}{P_1 * Q_1 \leq P_2 * Q_2}$	CoARROW $\frac{u_1 \leq t_1 \quad t_2 \leq u_2}{x @ t_1 \rightarrow t_2 \leq x @ u_1 \rightarrow u_2}$	CoFORALL $\frac{P \leq Q}{\forall(X : \kappa) P \leq \forall(X : \kappa) Q}$		
CoAPP $\frac{\text{variance}(X) = \vec{v} \quad T_i \leq^{v_i} T'_i}{x @ X \vec{T} \leq x @ X \vec{T}'}$		FRAME $x @ t \rightarrow t' \leq x @ (t P) \rightarrow (t' P)$		
COMMUTEARROW $x @ (\exists(X : \kappa).t) \rightarrow t' \equiv x @ \forall(X : \kappa).(t \rightarrow t')$		UNIFYAPP $\frac{A \{\vec{f} : \vec{t}\} \text{ is an unfolding of } X \vec{T}}{y @ X \vec{T} * y @ A \{\vec{f} = \vec{x}\} \equiv A \{\vec{f} : \vec{t}\} * y @ A \{\vec{f} = \vec{x}\}}$		
UNIFYBLOCK $\begin{aligned} & y @ A \{\vec{f} = \vec{x}\} * y @ A \{\vec{f} = \vec{x}'\} \\ \equiv & y @ A \{\vec{f} = \vec{x}\} * \vec{x} = \vec{x}' \end{aligned}$		UNIFYTUPLE $\begin{aligned} & y @ (= \vec{x}) * y @ (= \vec{x}') \\ \equiv & y @ (= \vec{x}) * \vec{x} = \vec{x}' \end{aligned}$		

Figure 8.1: Permission subsumption

$$\begin{array}{c}
\text{WEAKEN} \\
P_1 * P_2 \leq P_2
\end{array}
\qquad
\begin{array}{c}
\text{DUPLICATE} \\
P \text{ is duplicable} \\
\hline
P \leq P * P
\end{array}
\qquad
\begin{array}{c}
\text{ETA} \\
u_1 \leq (t_1 \mid P) \quad (t_2 \mid P) \leq u_2 \\
\hline
t_1 \rightarrow t_2 \leq u_1 \rightarrow u_2
\end{array}
\qquad
\begin{array}{c}
\text{COARROW}_2 \\
P \text{ is duplicable} \\
(u_1 \mid P) \leq t_1 \quad (t_2 \mid P) \leq u_2 \\
\hline
P * x @ t_1 \rightarrow t_2 \leq x @ u_1 \rightarrow u_2
\end{array}$$

$$\begin{array}{c}
\text{DECOMPOSEANY} \\
T \equiv \exists(x : \text{value})(=x \mid x @ T)
\end{array}$$

Figure 8.2: Some sample derivable subsumption rules

$$\begin{array}{c}
\text{COVARIANT} \\
T \leq T' \\
\hline
T \stackrel{\text{co}}{\leq} T'
\end{array}
\qquad
\begin{array}{c}
\text{CONTRAVARIANT} \\
T' \leq T \\
\hline
T \stackrel{\text{contra}}{\leq} T'
\end{array}
\qquad
\begin{array}{c}
\text{INVARIANT} \\
T \equiv T' \\
\hline
T \stackrel{\text{inv}}{\leq} T'
\end{array}
\qquad
\begin{array}{c}
\text{BIVARIANT} \\
T \stackrel{\text{bi}}{\leq} T'
\end{array}$$

Figure 8.3: Variance-dependent subsumption judgement

$$\begin{array}{c}
\text{SUB} \\
x @ T \leq x @ T' \\
\hline
T \leq T'
\end{array}$$

Figure 8.4: Subtyping judgement

8.2 Subsumption rules

Subsumption rules are shown in Figure 8.1. The judgement is of the form $P \leq Q$, meaning that one can weaken P in order to obtain Q . Some of these rules are reversible, that is, they work both ways: this is written as \equiv instead of \leq . This relation is similar to entailment in separation logic. One may write (§2.2) $P \Vdash Q$; in Mezzo, one writes $P \leq Q$.

Subsumption soundness is backed by an interpretation of permissions \Vdash [BPP14a] in terms of a run-time configuration. The proof of soundness is not covered in the present thesis.

Subsumption is useful in numerous situations. Subsumption is required, for instance, to show that the post-condition of a function is satisfied; that the current permission can be recombined to match the pre-condition required for a function call; that a type annotation can be satisfied.

The subsumption relation is, naturally, reflexive (REFLEXIVE). It is also transitive (TRANSITIVE): this property is essential for stepping through a program and performing intermediary reasoning steps. The weakest permission we can possibly have is empty (EMPTYTOP).

The next rules talk about our $*$ conjunction. The conjunction of permissions admits a neutral element empty (EMPTYAPPEARS); it is commutative and associative (STARCOMMUTATIVE, STARASSOCIATIVE). These rules merely state that $*$ is well-behaved and provides the usual properties one expects.

Combining TRANSITIVE, EMPTYTOP and EMPTYAPPEARS allows one to drop any permission at any time (this is WEAKEN, in Figure 8.2).

Equalities between program variables have a distinguished status among permissions. First, the permission $x @ =x$, also written $x = x$, is always available (EQUALITYREFLEXIVE). Second, whenever $y_1 = y_2$ is available, one may use y_1 or y_2 indifferently (EQUALSFOREQUALS). The latter rule formalizes our earlier claim that having $y_1 = y_2$ means we can use y_1 and y_2 interchangeably.

Some operations are only possible for duplicable permissions. For instance, duplicable permissions may be freely copied (DUPLICATE in Figure 8.2): this is the essence of “being duplicable”. This rule, however, is a par-

ticular case of the more general COPYDUP, which states that one may stash a duplicable permission under any context. The context may be non-linear (parameter of a list type application; codomain of an arrow), but we can conceptually obtain infinitely many copies of the permission.

HIDE DUPLICABLE PRECONDITION is also essential, since it allows one to drop the pre-condition of a function provided the pre-condition is available *and* duplicable. Combined with COARROW, it allows to derive COARROW₂ (Figure 8.2), which will be useful later on as it corresponds more closely to what the type-checker implements. Rules HIDE DUPLICABLE PRECONDITION and COARROW₂ can be derived from one another: we chose to keep HIDE DUPLICABLE PRECONDITION.

MIXSTAR states that a permission within a type/permission conjunction may be floated out into the outer conjunction. Combined with DECOMPOSE TUPLE and DECOMPOSE BLOCK, it allows to float out or nest permissions at arbitrary depth.

The next rules concern existential quantification. One may pack a permission with an existential quantifier (EXISTS INTRO). Quantifiers may be floated out of / into conjunctions (EXISTS STAR) and atomic permissions (EXISTS ATOMIC). We omit the non-capture premises of these rules which are standard.

The symmetrical rule for instantiation (FORALLELIM) is included, along with FORALLSTAR and FORALLATOMIC for moving universal quantifiers.

One may name tuple and constructor fields using singleton types (DECOMPOSE TUPLE, DECOMPOSE BLOCK). These rules are used pervasively throughout all the examples we described earlier; whenever we said earlier “the type-checker expands this permission”, we meant that the type-checker applied the decomposition subsumption rules.

More generally, one can decompose any type (DECOMPOSE ANY, Figure 8.2); this rule is derivable using the definition of subtyping, using EQUALITY REFLEXIVE followed by EXISTS INTRO.

The system is equipped with rules for dealing with data types. One may weaken a permission by transforming a concrete type into a nominal type (FOLD); in the special case that the data type only has one branch, one may perform the converse operation at any time (UNFOLD).

The dynamic type is *always* available as long as the object is exclusive (DYNAMIC APPEARS).

The system features an unknown type, that is, a top type at kind type. It could be, conceivably, defined as syntactic sugar for $\exists(a : \text{type}) a$; however, the implementation manipulates unknown as a separate type, so I chose to keep it in the remainder of the dissertation.

The system possesses covariant tuples (CO TUPLE), constructors (CO BLOCK), conjunctions (CO STAR) and arrows (CO ARROW). Type applications have a specific variance for each parameter (CO APP). We assume a variance function which returns the variance of each parameter of the type X . We then use a variance-dependent subsumption judgement (Figure 8.3).

Arrows are equipped with some special rules. The most important one is perhaps FRAME: if a function can be type-checked without P , then it can safely be passed P and will not use it. There is no premise: this works for any P , not just duplicable P 's. FRAME is used, in combination with COARROW, for deriving ETA, which will be used later on (Chapter 11) for the algorithmic specification of subtyping. COMMUTE ARROW is technical, but important for implementing the type-checker: it states that existential quantifiers in the domain of arrows are universal quantifiers above the arrow.



COARROW is a true deduction rule, not an equivalence rule. In other words:

$$t_2 \leq t_1 \wedge u_1 \leq u_2 \not\equiv t_1 \rightarrow u_1 \leq t_2 \rightarrow u_2$$

This equivalence holds in simpler systems, such as simply-typed lambda calculus with subtyping.

Let us see why this rule does not make sense in Mezzo with a simple example. Let us take $\text{id} @ () \rightarrow ()$. Using the frame rule (FRAME), one obtains $\text{id} @ (| p) \rightarrow (| p)$. If COARROW were an equivalence rule, going right to left we would obtain $() \leq (| p)$, that is, we would be able to obtain permissions out of thin air.

The last three rules are simplification rules that leverage information contained in redundant conjunctions. Intuitively, if I have both $y @ (x_1, \dots, x_n)$ and $y @ (x'_1, \dots, x'_n)$, then it must be the case that $x_1 = x'_1 * \dots * x_n = x'_n$. This is rule UNIFY TUPLE. A similar rule exists for constructors (UNIFY BLOCK). We used these rules in an earlier digression (§7.1) to illustrate how structural identity information for tuple arguments is preserved. These rules are useful in other contexts as well [GPP13]. In the case of constructors, a third rule comes in handy (UNIFY APP): it states that if I know that y is both a list and a Cons cell, then I can unfold the list type to regain full

information for the fields of the Cons cell. This rule is usually applied along with `DECOMPOSEBLOCK`, followed by `UNIFYBLOCK`, so as to further simplify the conjunction.

Once the subsumption relation is defined on types at kind perm, one can define a corresponding subtyping relation on types at kind type easily (Figure 8.4).

The subsumption relation is leveraged by two key algorithms: normalization (Chapter 9) and subtraction (Chapter 11).

About inconsistent conjunctions

The subsumption rules could, in theory, state which situations lead to inconsistent conjunctions. Possessing an inconsistent conjunction means that the corresponding program point is unreachable. The system is sound: execution cannot reach an inconsistent state.

A subsumption rule for an inconsistent conjunction will typically produce $\forall(P : \text{perm}) P$, that is, the \perp type at kind perm. One could also imagine an `inconsistent` keyword which runs over the current permission to somehow figure out that it is, indeed, inconsistent, and produces type \perp .

Examples of inconsistent permissions are, for instance, $x @ t * x @ t'$ where both t and t' are exclusive. Another inconsistent situation is $x @ (\vec{t}) * x @ (\vec{t}')$ when the number of fields don't match.

It turns out that this feature is hardly ever useful in practice, because Mezzo does not enforce pattern-matching exhaustivity. While the type-checker does detect certain inconsistent states, it is seldom leveraged by the user in order to type-check an actual program. We thus omit this aspect from the discussion.

8.3 Type-checking rules

The typing rules of expressions are presented in Figure 8.5. The format of our typing judgements is $K; P \vdash e : t$, meaning that under kinding environment K , consuming permission P allows one to type-check e with type t . The permission P may be understood as “the current permission”, that is, the permission which is available at the program point located right before the expression e . The kinding environment K contains existentially-quantified variables which have been opened in scope, either program variables or type variables. Program variables in K stand for the variables x , where $x @ t$ is in P . K binds all (existentially-quantified) program variables x that appear via $x @ t$ in P .

These typing rules have been proved sound using a syntactic approach [BPP14a]. This aspect is not developed in this thesis.

General comments

A Hoare-style presentation The rules do not use the traditional presentation “ $\{P\} s \{Q\}$ ”, which is common in statement-oriented languages, but an expression-based presentation of the form “ $\mathbb{Q} \vdash e : t$ ”. That is, the output of a rule is a type. This type can be interpreted as a predicate which holds for the return value `ret`. Phrased differently, the *post-condition* is $t(\text{ret})$.

Another way to think of it is, even though the right-hand side of a conclusion reads t , to imagine that t is of the form $(t' \mid P)$. The `LET` rule, for instance, embodies a sequence. One way to see it, without loss of generality, is as follows.

$$\frac{\text{LET} \quad K; P_0 \vdash e_1 : (t_1 \mid P_1) \quad K, x : \text{value}; P_1 * x @ t_1 \vdash e_2 : (t_2 \mid P_2)}{K; P_0 \vdash \text{let } x = e_1 \text{ in } e_2 : (t_2 \mid P_2)}$$

We thus thread a “current permission” throughout sequences, just like in the usual presentation of separation logic.

Since Mezzo blends permissions and types together, there is actually no reason to have a separate post-condition with a distinguished status in the typing rules. It thus feels natural to have the typing judgement yield a type instead of a type and a permission, since the latter can be recovered via the type/permission conjunction.

A declarative presentation Another thing to note is that this presentation is very far from being algorithmic. Indeed, designing an actual procedure for determining whether a program in Mezzo is well-typed or not is a complex task and I devote an entire part of this thesis to this problem (Part IV). For instance, the type-checking rules

<p>SUB</p> $\frac{K; P_2 \vdash e : t_1 \quad P_1 \leq P_2 \quad t_1 \leq t_2}{K; P_1 \vdash e : t_2}$	<p>VAR</p> $K \vdash x := x$	<p>LET</p> $\frac{K; P \vdash e_1 : t_1 \quad K, x : \text{value}; x @ t_1 \vdash e_2 : t_2}{K; P \vdash \text{let } x = e_1 \text{ in } e_2 : t_2}$	
<p>FUNCTION</p> $\frac{K; P * x @ t_1 \vdash e : t_2 \quad P \text{ is duplicable}}{K; P \vdash \lambda(x : t_1) : t_2. e : t_1 \rightarrow t_2}$	<p>ABSTRACTION</p> $\frac{K, X : \kappa; P \vdash e : t}{K; P \vdash \Lambda(X : \kappa). e : \forall(X : \kappa) t}$	<p>INSTANTIATION</p> $\frac{K; P \vdash e : \forall(X : \kappa) t_1}{K; P \vdash e : [T_2/X]t_1}$ $K; P \vdash e [T_2] : [T_2/X]t_1$	
<p>APPLICATION</p> $K; x_1 @ t_2 \rightarrow t_1 * x_2 @ t_2 \vdash x_1 x_2 : t_1$	<p>TUPLE</p> $K \vdash (\vec{x}) : (= \vec{x})$	<p>NEW</p> $\frac{A \{\vec{f}\} \text{ is defined}}{K \vdash A \{\vec{f} = \vec{x}\} : A \{\vec{f} = \vec{x}\} \text{ adopts } \perp}$	
<p>READ</p> $\frac{t \text{ is duplicable} \quad P \text{ is } x @ A \{F[f : t]\} \text{ adopts } u}{K; P \vdash x.f : (t \mid P)}$	<p>WRITE</p> $\frac{A \{.. \} \text{ is exclusive}}{K; x_1 @ A \{F[f : t_1]\} \text{ adopts } u \vdash x_1.f \leftarrow x_2 : (\mid x_1 @ A \{F[f = x_2]\} \text{ adopts } u)}$		
<p>MATCH</p> $\frac{\text{for every } i, \quad K; P \vdash \text{let } p_i = x \text{ in } e_i : t}{K; P \vdash \text{match } x \text{ with } \vec{p} \rightarrow \vec{e} : t}$	<p>WRITETAG</p> $\frac{A \{.. \} \text{ is exclusive} \quad B \{\vec{f}\} \text{ is defined} \quad \#\vec{f} = \#\vec{f}}{K; x @ A \{\vec{f} : \vec{t}\} \text{ adopts } u \vdash \text{tag of } x \leftarrow B : (\mid x @ B \{\vec{f} : \vec{t}\} \text{ adopts } u)}$		
<p>GIVE</p> $\frac{t_2 \text{ adopts } t_1}{K; x_1 @ t_1 * x_2 @ t_2 \vdash \text{give } x_1 \text{ to } x_2 : (\mid x_2 @ t_2)}$	<p>TAKE</p> $\frac{t_2 \text{ adopts } t_1}{K; x_1 @ \text{dynamic} * x_2 @ t_2 \vdash \text{take } x_1 \text{ from } x_2 : (\mid x_1 @ t_1 * x_2 @ t_2)}$		
<p>ADOPTS</p> $\frac{t_2 \text{ adopts } t_1}{K; P * x_1 @ \text{dynamic} * x_2 @ t_2 \vdash x_2 \text{ adopts } x_1 : (\text{bool} \mid x_2 @ t_2 * P)}$	<p>FAIL</p> $K; P \vdash \text{fail} : t$	<p>FRAME</p> $\frac{K; P_1 \vdash e : t}{K; P_1 * P_2 \vdash e : (t \mid P_2)}$	<p>EXISTS ELIM</p> $\frac{K, X : \kappa; P \vdash e : t}{K; \exists(X : \kappa) P \vdash e : t}$
<p>ASSERT</p> $K, P \vdash \text{assert } P : (\mid P)$	<p>ANNOT</p> $\frac{K, P \vdash e : t}{K, P \vdash (e : t) : t}$	<p>CONDITIONAL</p> $\frac{K, P * x @ \text{True} \vdash e_1 : t \quad K, P * x @ \text{False} \vdash e_2 : t}{K, P * x @ \text{bool} \vdash \text{if } x \text{ then } e_1 \text{ else } e_2 : t}$	

Figure 8.5: Typing rules

$$\begin{array}{c}
\text{LET TUPLE} \\
\frac{(\vec{t}) \text{ is duplicable} \quad K, \vec{x} : \text{value}; P * y @ (\vec{t}) * \vec{x} @ \vec{t} \vdash e : t}{K; P * y @ (\vec{t}) \vdash \text{let } (\vec{x}) = y \text{ in } e : t}
\end{array}
\qquad
\begin{array}{c}
\text{LET DATA MATCH} \\
\frac{(\vec{t}) \text{ is duplicable} \quad K, \vec{x} : \text{value}; P * y @ A \{\vec{f} : \vec{t}\} \text{ adopts } u * \vec{x} @ \vec{t} \vdash e : t}{K; P * y @ A \{\vec{f} : \vec{t}\} \text{ adopts } u \vdash \text{let } A \{\vec{f} = \vec{x}\} = y \text{ in } e : t}
\end{array}$$

$$\begin{array}{c}
\text{LET DATA MISMATCH} \\
\frac{A \text{ and } B \text{ belong to a common algebraic data type}}{K; P * y @ A \{\vec{f} : \vec{t}\} \text{ adopts } u \vdash \text{let } B \{\vec{f} = \vec{x}\} = y \text{ in } e : t}
\end{array}$$

$$\begin{array}{c}
\text{LET DATA UNFOLD} \\
\frac{A \{\vec{f} : \vec{t}\} \text{ adopts } u \text{ is an unfolding of } X \vec{T} \quad K; P * y @ A \{\vec{f} : \vec{t}\} \text{ adopts } u \vdash \text{let } A \{\vec{f} = \vec{x}\} = y \text{ in } e : t}{K; P * y @ X \vec{T} \vdash \text{let } A \{\vec{f} = \vec{x}\} = y \text{ in } e : t}
\end{array}$$

Figure 8.6: Auxiliary typing rules for pattern matching

do not provide a way to decide when to instantiate universal quantifiers (INSTANTIATE), when and how to rewrite the current permission (SUB), or which part of the current permission to frame (FRAME).

Some of the rules are also very general: WRITE, for instance, allows any t_2 . In practice, we want t_2 to be a singleton type, so as to keep local aliasing information, as we argued earlier. The type-checker will thus use a restricted instantiation of the rule. Conversely, some of the rules are restrictive and demand duplicable types (READ); the type-checker will use a proper normalized form for the current permission P to make sure such a premise is always satisfied.

A-normal form Many of the rules (APPLICATION, CONDITIONAL) require variables (named x) instead of expressions (named e). Having a name for a subexpression is sometimes mandatory, as in CONDITIONAL (we need a name x so as to refine the corresponding permission), sometimes a matter of convenience, as in APPLICATION (it saves the need for inlining the LET rule). In a rule such as TAKE, having x_2 allows us to state that type-checking the abandon operation leaves x_2 unchanged: having a name there is mandatory as well.

Introducing names for subexpressions can be done via additional let-bindings. This transformation is sometimes known as A-normal form. In the type-checker, this transformation is not performed explicitly through AST transformations, but rather performed on-the-fly, implicitly, when type-checking expressions.

This may be understood as additional typing rules, which introduce extra let-bindings. For instance, in the case of application:

$$\begin{array}{c}
\text{APP-GENERAL} \\
\frac{K; P \vdash \text{let } x_1 = e_1 \text{ in let } x_2 = e_2 \text{ in } x_1 x_2 : t}{K; P \vdash e_1 e_2 : t}
\end{array}$$

Reviewing the rules

SUB ties the subsumption and the type-checking relation together: we can recombine the current permission according to the subsumption rules we just saw.

VAR is the axiom rule.

LET is the sequencing rule; it does not specify how to craft t_1 so that it contains enough knowledge to type-check e_2 : again, this task is the job of an algorithm (which will, typically, put “everything left” in t_1).

FUNCTION is central. It allows a closure to capture any permission as long as it is duplicable. This restriction is necessary to justify our earlier claim that function types are always duplicable.

INSTANTIATION covers two cases. The first one is user-provided type applications, which have to be performed with the user-provided type T_2 . The second one is spontaneous type applications. Indeed, the translation phase (Chapter 7) introduces universal quantifiers above function definitions. This means that the type-checker needs to be able to instantiate these quantifiers without user-provided type annotations.

APPLICATION is standard. It corresponds to procedure call in separation logic. This rule is, of course, intended to be used in conjunction with FRAME to save permissions that are not required for performing the function call.

Rules TUPLE and NEW correspond to the allocation of tuple and constructors respectively. While the two rules are fairly standard, the adopts clause in NEW is interesting. A freshly-allocated constructor initially has its adopts clause set to \perp , that is, $\forall a.a$, provided that the constructor is exclusive. In other words, the constructor currently adopts nothing, but can adopt anything. This means that writing:

```
let g = Graph { ... } in
```

yields a new permission $g @ \text{graph } \{ \dots \}$ adopts \perp . Fortunately, constructor types are covariant in their adopts clause, meaning that we can instantiate at will the \perp type should we need, later on, to match a type annotation for a nominal type, or to perform an actual adoption/abandon operation. If the user writes:

```
(* n @ node a *)
give n to g;
```

or:

```
assert g @ graph a
```

Then the type-checker will apply the subsumption step:

$$\begin{array}{l} g @ \text{graph } \{ \dots \} \text{ adopts } \perp \\ \leq g @ \text{graph } \{ \dots \} \text{ adopts node } a \end{array}$$

In the case that the data constructor A is immutable, the adopts clause is irrelevant and is omitted.

READ has an interesting premise: it demands that t be duplicable. This seems restrictive; remember, however, that a combination of SUB and DECOMPOSEBLOCK will allow one to introduce a singleton type for the field f . Singleton types are always duplicable, meaning that the premise of this rule can always be satisfied without any loss of generality.



Requiring t to be duplicable allows us to sidestep completely complex reasoning that would need to take place should t be affine. Intuitively, we would need to remember we “carved a hole” in x , and track the missing field whose ownership has been taken. Not only would this be excessively complex, but we would also lack technical mechanisms for expressing it: Mezzo has no support for such reasoning. The use of singleton types in Mezzo is thus crucial, because we would not know how to type-check field accesses without it!

WRITE uses a singleton type to merely record that the field f of x_1 now points to x_2 . Thus, it does not need to mention any permission of the form $x_2 @ t_2$, which simplifies the discussion.

WRITETAG modifies the tag of a memory block in-place. The constructor A must be, naturally, exclusive for the mutation to take place. The constructor B, however, may be immutable; in this case, this is the “freeze” operation we mentioned earlier. The precondition states that the constructors must have the same number of fields: that way, the memory block does not have to be resized. While the OCaml garbage collector supports shrinking a block, we have not exposed this implementation detail in the typing rules of the language. The adopts clauses remain identical throughout the update, because changing a type does *not* change the type of elements currently adopted by this memory block.



This restriction is essential to ensuring soundness: otherwise, one could give an element, write a different tag into its adopter, and take the same element out with a different type.

An extra set of rules describe let-pattern bindings (Figure 8.6). This is not a separate judgement, but since the rules are quite heavyweight, they have been put in a separate figure for clarity. These rules cover *shallow* patterns, that is, patterns which are not nested. A deep pattern can always be transformed into a shallow pattern by introducing extra let-bindings in-between the let-pattern bindings: I do not describe this technical, uninteresting translation.

Interestingly, pattern-matching may *refine* the current permission and turn a nominal type into a concrete type, according to the constructor that is being matched (LETDATAUNFOLD). For instance, this rule will turn list a into $\text{Cons } \{\text{head} : a; \text{tail} : \text{list } a\}$ whenever matching on Cons . (The usual, converse direction, that is, weakening a constructor type into the corresponding nominal type, can be obtained by applying SUB.)

LETTUPLE and LETDATAMATCH are similar; they describe the destructuring of tuples and constructors. In the case of tuples, if one has $y @ (t_1, \dots, t_n)$ and writes $\text{let } (x_1, \dots, x_n) = y \text{ in } e$, this gives rise to $x_1 @ t_1 * \dots * x_n @ t_n$. Just like READ, and for the same reasons we exposed earlier, these two rules require that the types \vec{t} be duplicable. Indeed, the permission $y @ (\vec{t})$ is kept in the premise, along with $\vec{x} @ \vec{t}$. This is sound only if \vec{t} is duplicable. Again, an actual type-checking algorithm will ensure the duplicable premise is always satisfied by using the decomposition rules.

LETDATAMISMATCH makes sure that matching on a (wrong) constructor *from the same data type* is not an error. Intuitively, the information we have is too precise; we know that matching will fail, meaning that the remainder of the code is unreachable and any t is acceptable. Technically, this is important to make sure that programs do not break if the user refines their code with a more precise permission (e.g. False instead of bool).

There is no rule for the wildcard pattern, as it is similar to LET, except that x is fresh, hence not reachable by the user. Type annotations inside patterns are translated as type annotations on expressions, which are then checked by ANNOT. For instance:

```
let x, (y: int) = 1, 2 in
```

becomes:

```
let x, y = (1, 2): unknown, int in
```

As-patterns are translated with a let-binding: $\text{let } p \text{ as } x = e_1 \text{ in } e_2$ is translated into $\text{let } x = e_1 \text{ in let } p = x \text{ in } e_2$.

Pattern-matching (MATCH) piggybacks on the let-pattern rules to define the type-checking of match expressions. In the premise, the branches are all typed with the exact same t . Finding t in practice is difficult, since we have to apply subtyping rules on each t_i to make sure they converge onto the same t without losing useful information (Chapter 12).

Mezzo currently does not have an exhaustiveness check for pattern-matchings; this is a feature wish for the future.

GIVE, TAKE and ADOPTS are the rules which power our adoption/abandon mechanism. Rule GIVE does *not* need to return $x_1 @ \text{dynamic}$: the permission can be obtained at any time using the subsumption rules.

Dynamic failure (FAIL) means the expression can be given any type, since program execution will stop anyway.

FRAME is similar to the frame rule of separation logic. It allows one to put aside a permission P_2 , and type-check e with the remainder P_1 . One then regains the unused P_2 . This neat presentation, again, hides algorithmic difficulties: in practice, one has to synthesize P_2 and figure out which permissions should be put aside when type-checking, say, a function call. This is the frame inference problem from separation logic (§2.2).

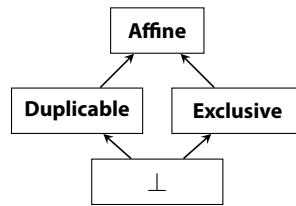
EXISTSELIM opens an existentially-quantified variable into scope.

The programmer can assert that a permission is available at a given program point (ASSERT) or that a given expression has the expected type (ANNOT). A point to note is that our assertions are prescriptive. Here is a code snippet that illustrates this.

```
(* x @ MNil *)
assert x @ mlist int;
(* x @ mlist int *)
```

In other words, even though we have a more precise permission, we stick by the annotation and weaken any permission we may have so as to exactly match the annotation. The same goes for annotations on expressions:

```
let x: mlist int = MNil in
...
(* x @ mlist int *)
```

Figure 8.7: The hierarchy of *modes*

CONDITIONAL type-checks if-then-else expressions. It behaves like a match statement, in the sense that the permission for the boolean is refined in each branch. This is, however, a simplified version: in our implementation, this rule works for any $x @ t$ as long as t is a data type with two branches. The first branch is understood to correspond to the “falsy” case (False, Nil, etc.) while the second branch is taken to be the “truthy” case (True, Cons, etc.). Permissions are refined accordingly in each branch.

This feature is kind of dubious, and we have not found it to be life-changing. We will probably remove it; I don't mention it in the remainder of this dissertation, even though it is still implemented at the time of this writing.

There is no type-checking rule for the mode constraint / type conjunction. Indeed, this requires discussing modes and facts. The following section (§8.4) will introduce the missing typing rules.

8.4 The duplicable and exclusive modes

I have mentioned at the beginning of Chapter 6 that even though Chapter 3 presents “duplicable a ” as a permission (and even though, indeed, the user may think of it as a permission), it is treated differently under-the-hood. I also mentioned that there were “modes” in the system. The present section makes this concept explicit.

The typing rules (Figure 8.5) mention predicates of the form “is duplicable” or “is exclusive”; “exclusive” and “duplicable” are *modes*, which we formally introduce, along with the corresponding lattice. We then provide a formal definition for the two predicates. Parameterized algebraic data types enjoy more complex predicates, which we call *facts*. Facts are introduced as well; however, computing them is the topic of the subsequent section.

The current status of the “is exclusive” predicate is unclear. It will either disappear once the new, more powerful axiomatization of adoption/abandon is implemented (§8.8), or it will be refined into “slim exclusive” and “fat exclusive” once a finer-grained control of the hidden field is offered to the user (§3.3).

In any case, even though the next two sections focus on a limited set of modes and facts, we envision a much more general use for these analyses. For instance, we could add a truly linear mode, where a variable cannot be discarded. This would extend the lattice of modes with a new top element; the framework would need very few modifications to properly analyze and compute facts within this extended mode lattice. We could even imagine user-provided modes, that is, user-provided predicates, which would play the role of type classes. The lattice would then be extended dynamically, but the analyses would essentially remain the same.

The mode/fact environment

Modes appear in types via the mode/type conjunction (Figure 7.1). From now on, our various judgements which operate on types will thus need to talk about modes.

Modes are predicates over types which form a lattice shown in Figure 8.7. These predicates only make sense for types at kind `perm` and `type`. Types at kind `value` have no mode. Parameterized types, such as `list`, have an arrow kind and enjoy more complex predicates called *facts*.

The affine mode is a strict superset of duplicable and exclusive. No object in Mezzo can be both duplicable and exclusive, i.e. no object can have mode \perp ; however, the bottom element is required for the fact computation which we will see later on. The mode lattice is thus complete. The lattice induces two natural meet \sqcap and join \sqcup operations.

We introduce a new parameter to our judgements, written F , which is the fact environment. It is used in both the extra typing rules (Figure 8.11) and the “is duplicable” judgement (Figure 8.10). F maps types to *facts*.

$f ::= \forall(\vec{Y} : \vec{\kappa}). \bigwedge \vec{v}$	fact
$i ::= h \Rightarrow m$	implication
$h ::= m_1 Y_1 \wedge \dots \wedge m_n Y_n$	hypothesis
false $::= \perp Y_1 \wedge \dots \wedge \perp Y_n$	syntactic sugar
true $::= \text{affine } Y_1 \wedge \dots \wedge \text{affine } Y_n$	syntactic sugar

Figure 8.8: Syntax of facts

A fact is of the following form (Figure 8.8):

$$\forall \vec{Y}. \bigwedge_m (m_1 Y_1 \wedge \dots \wedge m_n Y_n \Rightarrow m)$$

Here is the sample fact for lists, which we have seen already:

$$F(\text{list}) = \forall Y. \bigwedge \left\{ \begin{array}{ll} \text{true} & \Rightarrow \text{affine} \\ \text{duplicable } Y & \Rightarrow \text{duplicable} \\ \text{false} & \Rightarrow \text{exclusive} \\ \text{false} & \Rightarrow \perp \end{array} \right.$$

To simplify the discussion:

- we omit the universal quantification $\forall \vec{Y}$ over the formal parameters of the data type we are talking about; since we always study facts for one given data type, we assume the discussion to always be parameterized over a fixed set \vec{Y} ;
- we just write $F(\text{list}) = \text{duplicable } Y \Rightarrow \text{duplicable}$, as the implication whose conclusion is affine always takes the true hypothesis and other implications necessarily take the false hypothesis;
- we sometimes adopt an alternative presentation: we see a fact as a *complete map* from *modes* (in the conclusion) to *hypotheses*; we write:

$$F(\text{list}) = H \quad \text{and} \quad H(\text{duplicable}) = \text{duplicable } Y$$

We also see each hypothesis as a *complete map* from *formal parameters* to *modes*; we write:

$$F(\text{list})(\text{duplicable}) = h \quad \text{and} \quad h(Y) = \text{duplicable}$$

In the special case of non-parameterized types (user-defined types or type variables), $F(X)$ maps X into a special form of fact called a *constant fact*, which is isomorphic to a *mode*.

Definition 8.1 (Constant fact). *A constant mode m maps into a fact as follows:*

$$\text{constant}(m) = \begin{cases} \text{true} \Rightarrow m' & \text{if } m' \geq m \\ \text{false} \Rightarrow m' & \text{otherwise} \end{cases}$$

We write $F(X) = m$ to state that the best (smallest) known mode for X is m . We write $F(t) = f$ for “parameterized type t satisfies fact f ”. Computing the best fact for a user-defined parameterized type is the topic of the next section (§8.5).

Definition of modes

We now define the subset of types that enjoy these predicates, which we write respectively “ t is duplicable” and “ t is exclusive” (Figure 8.10, Figure 8.9). The two judgements are mutually exclusive and a type that satisfies neither is affine.

$\frac{\text{X-DEF}}{\text{data mutable } X (\vec{Y} : \vec{\kappa}) = \dots}{X \text{ is exclusive}}$	$\frac{\text{X-CONCRETE}}{A \text{ is a data constructor of } X \quad X \text{ is exclusive}}{A \{\vec{f} : \vec{t}\} \text{ adopts } u \text{ is exclusive}}$	$\frac{\text{X-APP}}{X \text{ is exclusive}}{X \vec{t} \text{ is exclusive}}$
$\frac{\text{X-AND}}{F' = \text{assume}(F, m X) \quad F' \vdash t \text{ is exclusive}}{F \vdash (t \mid m X) \text{ is exclusive}}$	$\frac{\text{X-VAR}}{F(X) = \text{exclusive}}{F \vdash X \text{ is exclusive}}$	$\frac{\text{X-BAR}}{F \vdash t \text{ is exclusive}}{F \vdash (t \mid P) \text{ is exclusive}}$
$\frac{\text{X-FORALL}}{F' = \text{extend}(F, \text{affine } X) \quad F' \vdash t \text{ is exclusive}}{F \vdash \forall(X : \kappa) t \text{ is exclusive}}$	$\frac{\text{X-EXISTS}}{F' = \text{extend}(F, \text{affine } X) \quad F' \vdash t \text{ is exclusive}}{F \vdash \exists(X : \kappa) t \text{ is exclusive}}$	

Figure 8.9: Definition of the “exclusive” judgement (some rules omitted)

$\frac{\text{D-CONCRETE}}{A \text{ is a data constructor of } D \quad D \text{ not mutable} \quad F \vdash \vec{t} \text{ is duplicable}}{F \vdash A \{\vec{f} : \vec{t}\} \text{ is duplicable}}$	$\frac{\text{D-TUPLE}}{F \vdash \vec{t} \text{ is duplicable}}{F \vdash (\vec{t}) \text{ is duplicable}}$	$\frac{\text{D-APP}}{F \vdash \text{the unfoldings } \vec{A} \{\vec{f} : \vec{t}\} \text{ are duplicable}}{F \vdash X \vec{T} \text{ is duplicable}}$
$\frac{\text{D-ABSTRACT}}{F(X) \text{ (duplicable)} = h \quad \forall i, u_i \text{ is } h(i)}{F \vdash X \vec{u} \text{ is duplicable}}$	$\frac{\text{D-ARROW}}{F \vdash t \rightarrow u \text{ is duplicable}}$	
$\frac{\text{D-FORALL}}{F' = \text{extend}(F, \perp X) \quad F' \vdash t \text{ is duplicable}}{F \vdash \forall(X : \kappa) t \text{ is duplicable}}$	$\frac{\text{D-EXISTS}}{F' = \text{extend}(F, \text{affine } X) \quad F' \vdash t \text{ is duplicable}}{F \vdash \exists(X : \kappa) t \text{ is duplicable}}$	
$\frac{\text{D-SINGLETON}}{F \vdash =x \text{ is duplicable}}$	$\frac{\text{D-BAR}}{F \vdash t \text{ is duplicable} \quad F \vdash P \text{ is duplicable}}{F \vdash (t \mid P) \text{ is duplicable}}$	$\frac{\text{D-DYNAMIC}}{F \vdash \text{dynamic is duplicable}}$
$\frac{\text{D-EMPTY}}{F \vdash \text{empty is duplicable}}$	$\frac{\text{D-STAR}}{F \vdash p \text{ is duplicable} \quad F \vdash q \text{ is duplicable}}{F \vdash p * q \text{ is duplicable}}$	$\frac{\text{D-ANCHORED}}{F \vdash t \text{ is duplicable}}{F \vdash x @ t \text{ is duplicable}}$
	$\frac{\text{D-AND}}{F' = \text{assume}(F, m X) \quad F' \vdash t \text{ is duplicable}}{F \vdash (t \mid m X) \text{ is duplicable}}$	$\frac{\text{D-VAR}}{F(X) = \text{duplicable}}{F \vdash X \text{ is duplicable}}$

Figure 8.10: Definition of the “duplicable” judgement

The duplicable judgement The “ t is duplicable” judgement is defined co-inductively. The definition of the judgement assumes F to be filled with facts for abstract data types.

Initially, all variables are affine; F may be updated for a mode/type conjunction (D-AND); F is used when looking up the fact for a variable (D-VAR). The exact definition of assume and extend is provided later (§8.4); intuitively, extend just extends the environment F , while assume performs a *meet* operation on the lattice of modes.

Tuples, being immutable, are duplicable as long as their contents are (D-TUPLE): the type (int, int) is duplicable, while the type $(\text{ref int}, \text{ref int})$ is not. For data constructors, the same goes as long as the constructor itself is immutable (D-CONCRETE).

For a data type application (D-APP), we need to consider all possible branches. For that, we take each projection, that is, each branch where the formal parameters of the data type have been replaced with the effective parameters of the type application, and make sure that each one of them is in turn duplicable. This, indirectly, also makes sure (D-CONCRETE) that X is not defined as mutable.

Abstract types (D-ABSTRACT) have *facts* attached to them. For abstract types, facts are user-provided, so we look up the fact associated to X using F . Seeing, as we mentioned earlier, the fact as a map from conclusions to hypotheses, we look up the conditions associated to the duplicable conclusion. If all hypotheses are satisfied for the effective arguments, then we can draw the conclusion that the type application is indeed duplicable.

Arrows (D-ARROW) are duplicable, since they can capture nothing but duplicable permissions.

Quantified types (D-FORALL, D-EXISTS) are duplicable if the underlying type is.

The rest of the rules are standard.

The exclusive judgement The “ t is exclusive” is only defined for a small subset of types, as the only exclusive types are concrete types and type applications that belong to an mutable-defined data type. These can take the form of type variables, though; rules X-AND and X-VAR make sure that $(X \mid \text{exclusive } X)$ is exclusive too.

An incomplete set of rules These rules are incomplete, and are unable to infer the proper mode in some cases. The first reason is that $\exists a.(a, (\text{duplicable } a \mid a))$ will be considered affine. Indeed, in the first occurrence of a is treated as affine, while the second occurrence is treated as duplicable. The implementation performs a limited *hoisting* phase, where hypotheses of the form $\text{duplicable } a$ are collected and recorded into F before descending on the actual type. We have no completeness results for this hoisting procedure.

The second reason why the rules are incomplete is more subtle. Consider the type $\exists(a : \text{type}) a$. We have:

$$\exists(a : \text{type}) a \equiv \exists(x : \text{value}) =x \equiv \exists(a : \text{type}) (\text{duplicable } a \mid a)$$

From $\exists(a : \text{type}) a$, we can obtain $(=x \mid x @ a)$, then drop $x @ a$ to gain the second type; from $\exists(x : \text{value}) =x$, we can re-pack on $=x$ to gain the third type; from $\exists(a : \text{type}) (\text{duplicable } a \mid a)$, we can just drop the mode hypothesis and regain the first type.

The rules will find the first type to be affine, while they will find the second and third types to be duplicable, even though the three are equivalent.

Interpretation of modes

One can give a semantic interpretation to these mode predicates using the permission interpretation relation \Vdash [BPP14a].

Lemma 8.2 (Semantics of exclusive). *If $K \vdash t : \text{type}$, t is exclusive $\Rightarrow x @ t * y @ t \Vdash x$ and y are distinct values*

Lemma 8.3 (Semantics of duplicable). *If $K \vdash p : \text{perm}$, p is duplicable $\Rightarrow p \Vdash p * p$. If $K \vdash t : \text{type}$, t is duplicable if and only if $x @ t$ is duplicable.*

Type-checking rules

We mentioned earlier that we would introduce later on the typing rules for the mode constraint/type conjunction.

These extra typing rules are shown in Figure 8.11. MODEELIM says that whenever we *know* that $m X$ holds, we can update the fact environment F with it. We refine the already-existing mode for X with the more precise mode m using a meet operation \sqcap . By default, the mode for a variable is the top element of the lattice, that is, affine. MODEINTRO says that whenever we have learned that $m X$ holds, we can produce a witness of it via a mode constraint/type conjunction.

$$\begin{array}{c}
\text{MODEELIM} \\
\frac{K; F \{X \mapsto F(X) \sqcap m\}; P * x @ t \vdash e : t'}{K; F; P * x @ (t \mid mX) \vdash e : t'}
\end{array}
\qquad
\begin{array}{c}
\text{MODEINTRO} \\
\frac{K; F; P \vdash e : t \quad F(X) \leq m}{K; F; P \vdash e : (t \mid mX)}
\end{array}$$

Figure 8.11: Extra typing rules for dealing with mode constraint/type conjunctions

Computing “the mode of a type”

The coinductive definition of the “is duplicable” judgement allows one to check that a given type satisfies a mode, but provides no way to *compute* “the” mode of a type. By this informal expression, we mean the lowest mode in the lattice that holds for a given type.

One could read Figure 8.10 as a set of recursive rules that, when applied, perform a computation. This would be sound, but too strict. Indeed, looking at D-APP, we see that a type application is duplicable if and only if all its unfoldings are duplicable. For instance, list t is duplicable if both Nil and Cons {head : t ; tail : list t } are duplicable. A recursive reasoning, for lack of a co-inductive “list t is duplicable” hypothesis, would either loop or determine Cons {head : t ; tail : list t } to be affine and conclude that list t is affine as well.

One could also perform several attempts, first trying to show that “list t is duplicable” then, failing that, try to show that “list t is exclusive”. This would be terribly inefficient, because, not knowing the mode of t , we would need to recursively consider all possible modes for t . If t itself contains type applications, the complexity becomes exponential.

The problem, fundamentally, lies with algebraic data types, as the mode of a type application depends on the parameters. We cannot afford to compute the mode of list t for every possible value of t . What we want instead is to state the following *fact*: “if t is duplicable, then list t is duplicable”.

Having this would allow us to compute the fact for list once and for all; then, any time we need to determine the mode of list t , we merely need to look up the fact associated to list, and check what conclusion the mode of t allows us to draw.

8.5 Facts

Facts are used all throughout the various rules (type-checking, modes, signature ascription). The present section formally defines facts; describes what it means for facts to be correct; describes how to present fact computation as a fixed point computation on a lattice.

Syntax of facts

The syntax of facts is presented in Figure 8.8. A fact is always parameterized over a set of formal variables \vec{Y} which represent the formal parameters of a type application. Facts are computed for data types and alias definitions; they are user-provided for abstract types. We discussed the notation earlier (§8.4), and introduced a more concise notation which identifies a fact f with a map H from conclusions (modes) to hypotheses; the notation also identifies a hypothesis with a map from formal parameters to modes.

The general form of a fact is a quantification $\forall(\vec{Y} : \vec{\kappa})$, where $\vec{\kappa}$ matches the expected kinds of the type application $X \vec{Y}$ we are considering. The quantification is followed by a conjunction of implications. For each implication, if the hypotheses are satisfied, then the conclusion is a valid mode for X .

In the rest of the discussion, we focus on the analysis of a single type X whose formal parameters are $\vec{Y} : \vec{\kappa}$; we thus omit the universal quantifications and consider our judgements to be parameterized over the formal variables \vec{Y} .

A trivial fact that any type X enjoys is: “true \Rightarrow affine” (quantification omitted for brevity). That is, for a fixed set of parameters \vec{Y} , facts form an upward-closed semi-lattice, whose \top element is “true \Rightarrow affine”.

We assume facts in the lattice to be total, that is, to be a conjunction of exactly four implications, one for each possible mode. Adopting the “map” view, this means the map from modes to hypotheses is total. Ensuring a fact is total is easy: we just need to extend its conjunction with implications of the form “false $\Rightarrow m$ ” for the missing modes \vec{m} . As mentioned earlier, when writing a fact, as in the list example above, we omit the implications whose

premise is false.

We also assume our hypotheses to be total, that is, to either have exactly one clause of the form $m Y$ for each Y . This is not a restriction: one can always request affine Y without loss of generality. We then define true to be a synonym for \wedge (affine \bar{Y}) and false to be a synonym for \wedge ($\perp \bar{Y}$).

Operations on the facts lattice

The following two operations allows us to compute the conjunction of hypotheses $\wedge m Y$, where $m Y$ is called a *clause*.

Definition 8.4 (Conjunction of hypotheses). *The conjunction of hypotheses $h \wedge h'$ is defined as the pairwise conjunction of clauses. That is, adopting the map view, $(h \wedge h')(Y_i) = (m_i \sqcap m'_i)$.*

Hypotheses are total, meaning that the conjunction of hypotheses is always defined.

The conjunction of clauses “goes down” on the mode lattice: for instance, if we demand that a parameter be both affine and duplicable, then we need the strictest (i.e. lowest) mode of the two for the parameter, that is, duplicable \sqcap affine = duplicable.

This definition makes sense: expanding the definitions of true and false, one gets $h \wedge \text{false} = \text{false}$ and $h \wedge \text{true} = h$.

Being able to compute $h \wedge h'$ allows us to define a join operation on the lattice of facts.

Definition 8.5 (Join operation). *We define the join operation \sqcup on our lattice as follows. Adopting the map view:*

$$(f \sqcup f')(m) = (f(m) \wedge f'(m))$$

We also need a meet operation for our lattice. Indeed, the grammar of types features the conjunction of a mode hypothesis and a type, written $(t \mid m X)$. During the traversal of a type, we maintain an environment of facts F ; this environment needs to be refined to take into account the new hypothesis for X .

I mentioned earlier that intuitively, assume performed a meet on the mode lattice. F , however, contains *facts*: we thus need to define the corresponding meet operation:

$$(f \sqcap f')(m) = (f(m) \vee f'(m))$$

The disjunction of hypotheses is, however, not defined. Indeed, our hypotheses are made up of conjunctions, and do not handle disjunction. We therefore define the “meet” operation on our lattice only in the case where one of the arguments is a constant fact.

Definition 8.6 (Meet operation). *Assuming that f_1 is a constant fact (i.e. is made up of implications of the form $\text{false} \Rightarrow m$ or $\text{true} \Rightarrow m$), we define the “meet” of f_1 and f_2 to be:*

$$(f_1 \sqcap f_2)(m) = \begin{cases} f_2(m) & \text{if } f_1(m) = \text{false} \\ \text{true} \Rightarrow m & \text{if } f_1(m) = \text{true} \end{cases}$$

This is not a restriction: the grammar of types only allows for mode hypotheses over a type variable, meaning that we always compute the meet operation between a fact $F(X)$ and a constant fact, which is well-defined.

Definition 8.7. *Now that the meet operation is properly defined, we can present the definition of assume.*

$$\text{assume}(F, m X) = F[X \mapsto F(x) \sqcap \text{constant}(m)]$$

Another operation I mentioned earlier is extend.

Definition 8.8 (Extend operation). *If $X \notin F$, then:*

$$\text{extend}(F, m X) = F[X \mapsto \text{constant}(m)]$$

Correctness of a fact environment

A fact environment F maps a type t to a fact f , that is, $F(t) = f$. We define what it means for F to be correct with regard to a specific type t . There are three forms of facts for t :

- the type enjoys no special fact, meaning that the only implication with a non-false premise is $true \Rightarrow$ affine: such a fact is always correct, meaning we have nothing to check;
- the type is duplicable, meaning that it enjoys an implication of the form $m_1 Y_1 \wedge \dots \wedge m_n Y_n \Rightarrow$ duplicable, a trivial implication $true \Rightarrow$ affine, and that all others implications have a false premise (a type cannot be both duplicable and exclusive): in this case, we only check the non-trivial implication;
- the type is exclusive: this case is similar to the duplicable case.

The rule for the correctness of a fact environment with regard to t thus checks a single implication, where the conclusion m is either duplicable or exclusive.

$$\frac{\text{FACTS-CORRECT-T} \quad F(t) = m_1 Y_1 \wedge \dots \wedge m_n Y_n \Rightarrow m \quad \forall \vec{t}, (\forall i, t_i \text{ is } m_i) \Rightarrow \text{every unfolding } A \{\vec{f} : \vec{u}\} \text{ of } t \vec{t} \text{ is } m}{F \text{ is correct with regard to } t}$$

The check is co-inductive: we assume from the start that the fact f holds for t ; next, we also assume that the hypotheses for the effective parameters \vec{t} hold. We check that for each branch of the data type, the m mode holds, meaning that the fact for t is correct.

Definition 8.9. *A fact environment F is correct if it is correct with regard to every parameterized type t .*

A fact environment that is trivially correct is that which assigns constant(affine) to every type. This is, most of the time, not the best fact environment F , though.

The “best” fact environment F is that which assigns the lowest possible fact to each parameterized type. We have no clue, however, as to how one can compute the best fact for t . This is the purpose of the next section: we present a procedure that infers the “best” fact f for a type t , that is, the smallest element f in the fact lattice such that $F[t \mapsto f]$ is correct with regard to t .

Fact inference

We introduce an inference judgement of the form $F \vdash \text{fact}(t) = f$, meaning that if F contains facts for other data types, then the smallest fact we can infer for t is f . For groups of mutually recursive data type definitions, the type-checker of Mezzo leverages this procedure and determines the smallest fact for each type, through the use of a fixed-point computation.

Implicitly, the fact inference procedure is parameterized over the set of formal parameters \vec{Y} of the type whose fact we are currently inferring. That is, the judgement should be written $\vdash_{\vec{Y}}$. We omit the subscript for readability.

The distinguished nature of the variables \vec{Y} , compared to other variables, is conveyed by the special fact that they enjoy.

Definition 8.10 (Fact for a parameter). *If Y is a parameter of the type whose fact is currently being inferred, then the fact for Y is:*

$$\text{parameter}(Y) = m Y \Rightarrow m$$

This means that we have control over Y : we can pick a mode for the formal parameter Y , which will in turn be the mode of the type Y .

One way to compute the best F is to take F_0 to be the fact environment which assigns constant(\perp) to each parameterized type, then perform a fixed point computation using the monotonous iter function.

$$\frac{\text{FACTS-INITIAL} \quad F_0 = \vec{t} \mapsto \perp \quad \text{FACTS-ITER} \quad F' = F[\vec{Y} \mapsto \text{parameter}(\vec{Y})] \quad F' \vdash \text{fact}(\vec{t}, \vec{f})}{\text{iter}(F) = F[\vec{t} \mapsto \vec{f}]}$$

<p>FACT-UNKNOWN $F \vdash \text{fact}(\text{unknown}) = \text{duplicable}$</p>	<p>FACT-DYNAMIC $F \vdash \text{fact}(\text{dynamic}) = \text{duplicable}$</p>	<p>FACT-SINGLETON $F \vdash \text{fact}(=x) = \text{duplicable}$</p>
<p>FACT-ARROW $F \vdash \text{fact}(t \rightarrow u) = \text{duplicable}$</p>	<p>FACT-VAR $\frac{}{F \vdash \text{fact}(X) = F(X)}$</p>	<p>FACT-APP $\frac{F(u) = f_0 \quad F \vdash \text{fact}(\vec{v}) = \vec{f} \quad f = \text{compose}(f_0, \vec{f})}{F \vdash \text{fact}(u \vec{v}) = f}$</p>
<p>FACT-AND $\frac{F' = \text{assume}(F, m X) \quad F' \vdash \text{fact}(t) = f}{F \vdash \text{fact}((t \mid m X)) = f}$</p>	<p>FACT-FORALL $\frac{F' = \text{extend}(F, \perp X) \quad F' \vdash \text{fact}(t) = f}{F \vdash \text{fact}(\forall(X : \kappa) t) = f}$</p>	<p>FACT-EXISTS $\frac{F' = \text{extend}(F, \text{affine } X) \quad F' \vdash \text{fact}(t) = f}{F \vdash \text{fact}(\exists(X : \kappa) t) = f}$</p>
<p>FACT-TUPLE $\frac{F \vdash \text{fact}(t_i) = f_i \quad f_i(\text{duplicable}) = h_i}{F \vdash \text{fact}((t_1, \dots, t_n)) = h_1 \wedge \dots \wedge h_n \Rightarrow \text{duplicable}}$</p>	<p>FACT-CONCRETE-X A is a constructor of X t is exclusive $\frac{}{F \vdash \text{fact}(A \{\vec{f} : \vec{t}\}) = \text{exclusive}}$</p>	
<p>FACT-CONCRETE-D A is a constructor of X $\frac{F \vdash \text{fact}(t_i) = f_i \quad f_i(\text{duplicable}) = h_i}{F \vdash \text{fact}(A \{\vec{f} : \vec{t}\}) = h_1 \wedge \dots \wedge h_n \Rightarrow \text{duplicable}}$</p>	<p>FACT-BAR $\frac{F \vdash \text{fact}(t) = f_1 \quad F \vdash \text{fact}(P) = f_2}{F \vdash \text{fact}((t \mid P)) = f_1 \sqcup (f_2 \sqcap \text{constant}(\text{exclusive}))}$</p>	
<p>FACT-EMPTY $F \vdash \text{fact}(\text{empty}) = \text{duplicable}$</p>	<p>FACT-ANCHORED $\frac{F \vdash \text{fact}(t) = h \Rightarrow \text{exclusive} \wedge \vec{v}}{F \vdash \text{fact}(x @ t) = \vec{v}}$</p>	<p>FACT-STAR $\frac{F \vdash \text{fact}(p) = f_1 \quad F \vdash \text{fact}(q) = f_2}{F \vdash \text{fact}(p * q) = f_1 \sqcup f_2}$</p>

Figure 8.12: Fact inference

The iter function relies on the rules defined in Figure 8.12. The final environment F obtained via this procedure is correct (Definition 8.9).

Lemma 8.11 (Existence and unicity). *Given a type t , there exists a unique fact f that satisfies $\text{fact}(f) = t$ as well as:*

$$\forall f'. \text{fact}(f') = t \Rightarrow f \leq f'$$

The fact environment F obtained via the fixed-point computation finds the unique, smallest fact for every type t .

Let us briefly comment the fact inference rules, which are exposed in Figure 8.12. For brevity, we write `duplicable` instead of `constant(duplicable)`.

The inference procedure is called via `FACTS-CORRECT` judgement; we infer a fact in an extended environment F' , where the “best” fact is assigned to the type’s parameters. Therefore, rule `FACT-VAR` consists in a mere lookup in the environment. Rule `FACT-AND` refines the current fact for a type variable using the “meet” operation on the lattice of facts. Rule `FACT-TUPLE` only has one non-trivial conclusion: a tuple is duplicable if all its components are. There are two rules for concrete data types. Rule `FACT-CONCRETE-X` tells that a concrete data type that “belongs” to a mutable data type is exclusive. Rule `FACT-CONCRETE-D` is for constructors that belong to an immutable data type; this rule is analogous to `FACT-TUPLE`. Rule `FACT-ANCHORED` removes any implication whose conclusion is X , as the “ p is exclusive” judgement has no meaning when p is at kind `perm`. Rule `FACT-BAR`, conversely, extends the fact for the permission with a exclusive constant fact, so that the resulting conjunction of a type and a permission can be seen as exclusive if the type itself is.

Rule `FACT-APP` is slightly more involved, and requires us to explain the `compose` function, which relates together the fact of the data type currently examined, and the pre-computed fact for another data type. Let us take

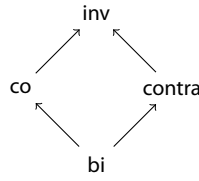


Figure 8.13: The variance lattice

an example before formally defining compose. We consider the following data type definition:

$$\text{data listpair } Y \ Y' = \text{ListPair } \{\text{listpair} : \text{list } (Y, Y')\}$$

The initial fact for “ Y ” (resp. “ Y' ”) is “parameter(Y)” (resp. “parameter(Y')”). Thus, the inferred fact for “ (Y, Y') ” is “duplicable $Y \wedge$ duplicable $Y' \Rightarrow$ duplicable”. Besides, assuming the environment F contains the inferred fact for list already, we have:

$$\forall Z. \text{fact}(\text{list } Z) = \text{duplicable } Z \Rightarrow \text{duplicable}$$

The value of the formal parameter “ Z ” for list is “ (Y, Y') ”. Therefore, we need “ (Y, Y') ” to be duplicable for “list (Y, Y') ” to be duplicable. Knowing that:

$$\text{fact}((Y, Y')) = \text{duplicable } Y \wedge \text{duplicable } Y' \Rightarrow \text{duplicable}$$

we find that “duplicable $Y \wedge$ duplicable Y' ” is a sufficient condition for “list (Y, Y') is duplicable” to hold. This is also a necessary condition. Therefore, we infer that, inside the definition of listpair, the fact for “list (Y, Y') ” is “duplicable $Y \wedge$ duplicable $Y' \Rightarrow$ duplicable”.

Definition 8.12 (Fact composition). *We consider a type application “ $X \vec{T}$ ”. Assume that $F(X) = \forall \vec{Z}. f_X$, and that $\text{fact}(T_i) = f_i$. Then:*

$$\text{compose}(F_X, \vec{f})(m) = f_1(F_X(m)(Z_1)) \wedge \dots \wedge f_n(F_X(m)(Z_n))$$

Taking our earlier example where $X = \text{list}$, we have $f_{\text{list}} = \forall Z. \text{duplicable } Z \Rightarrow \text{duplicable}$ and the fact for the argument of the type application is $f_1 = \text{duplicable } Y \wedge \text{duplicable } Y' \Rightarrow \text{duplicable}$.

For $m = \text{duplicable}$, we have: $F_{\text{list}}(\text{duplicable})(Z) = \text{duplicable}$. We then take $f_1(\text{duplicable}) = \text{duplicable } Y \wedge \text{duplicable } Y'$. Seeing that there is only one argument, the result of compose is thus $\text{duplicable } Y \wedge \text{duplicable } Y' \Rightarrow \text{duplicable}$.

Implementation

Fact inference is implemented using Fix [Potog].

8.6 Variance

Definition of variance

Variance is a standard notion in programming languages equipped with parameterized data types. In the context of ML with subtyping (such as in, say, OCaml), we say that a type T is covariant with respects to formal variable X if $u_1 \leq u_2 \Rightarrow [u_1/X]T \leq [u_2/X]T$. Other possible values for variance, besides covariant, are contravariant, invariant and bivariant. These values form a complete lattice (Figure 8.13) equipped with meet (\cap) and join (\cup) operations.

Variance is, just like “is duplicable”, a co-inductive judgement. The definition of variance relies on subtyping, and the subsumption relation relies on variance information (COAPP): the two are mutually recursive.

We introduce a variance environment V , which maps a data type D and an index i to one of the four possible variances (Figure 8.14), denoted by the meta-variable v . We prefix the subtyping and subsumption relations with a variance environment V , and write: $V \vdash P \leq Q$.

$$\begin{aligned}
V(D, i) = v &\triangleq \forall T \text{ unfolding of } D \vec{X}, v(T, X_i, V) \\
\text{co}(T, X, V) &\triangleq \forall u_1, u_2, V \vdash u_1 \leq u_2 \Rightarrow V \vdash [u_1/X]T \leq [u_2/X]T \\
\text{contra}(T, X, V) &\triangleq \forall u_1, u_2, V \vdash u_1 \leq u_2 \Rightarrow V \vdash [u_2/X]T \leq [u_1/X]T \\
\text{bi}(T, X, V) &\triangleq \forall u_1, u_2, V \vdash u_1 \leq u_2 \Rightarrow V \vdash [u_1/X]T \equiv [u_2/X]T \\
\text{inv}(T, X, V) &\triangleq \forall u_1, u_2, V \vdash u_1 \leq u_2 \Rightarrow \text{true}
\end{aligned}$$

Figure 8.14: The variance environment and the variance predicates

The informal notation for CoAPP can be replaced with a more formal one.

$$\frac{\text{CoAPP}_{\text{FORMAL}} \quad T_i \stackrel{V(D,i)}{\leq} T'_i}{x @ D \vec{T} \leq x @ D \vec{T}'}$$

Computation of variance

Just like with the “is duplicable” predicate, we do not want to recompute variance on-the-spot, and wish to have the information readily available for each data type. Even if we did, just like fact inference, for lack of a co-inductive hypothesis, we would fail to infer proper variance for data types.

Formally, we want the best variance V , that is, the smallest relation that satisfies rules of Figure 8.14. By smallest, we mean that it assigns the lowest possible v to each couple (D, i) .

This relation can easily be computed using a fix-point computation. This is performed using F. Pottier’s Fix library [Poto9]. Mapping the variance computation onto Fix’s interface has been covered by a series of blog posts [SP14a, SP14c, SP14b]. I wrote the part about variance.

Since this is a classic textbook analysis, I do not detail any further this part.

8.7 Signature ascription

Mezzo is equipped with a basic module system which offers qualified names and type abstraction. There are a few checks required to make sure an implementation satisfies the mandatory interface.

Duplicable exports Mezzo imposes the restriction that only duplicable items may be exported. To understand this restriction, let us imagine for a minute that we do not have the restriction, and let us see what kind of verifications would be needed to ensure soundness.

Exporting an affine item x from a module M is dangerous: since two modules N and N' may both re-export x under $n : : x$ and $n' : : x$, a module O which imports both N and N' may thus have two copies of a non-duplicable permission.

A way to ensure soundness would be to ensure that type-checking N leaves intact the interface of M , that is, does not consume or mutate anything from M . An early version of Mezzo enforced this: after checking a module N , Mezzo would check that for every module M that N depends on, M could still be checked against its interface.

This guarantees that parallel composition of modules is possible, and that type soundness does not depend on the initialization order of modules.

If we were to allow any sort of exports, then additional implementation difficulties would arise. Consider the following situation:

```

(* export.mz *)
data mutable t = T
val x = T
val y = x

(* export.mzi *)
abstract t

```



```
val x: t
val y: t
```

In order to properly reject this module, we need to take the final permission P obtained after type-checking the `mz` file, then take permissions out of P as we step through the various items in the `mzi` file, so as to make sure one can not obtain $x @ t$ twice. This is extra complexity.

Since no one ever cared to export non-duplicable items, we removed the feature from the language.

Checking values If the interface exports `val x: t`, then we must check that there exists, in the permission P available after type-checking the implementation, a permission $x @ t' \leq x @ t$. We must also check that t is duplicable.

Checking type definitions Any type *definition* (abstract, alias or data type) may be made abstract.

- the fact exported in the interface, if any, must be a consequence of the fact from the implementation;
- the variance advertised in the interface, if any, must be less precise than the one in the implementation.

Lacking any fact annotation, a type is assumed to be affine (the top fact). Similarly, lacking any variance annotation, a type is assumed to be invariant for all its parameters (the top variance).

In the case that either a data type or an alias definition is exported “as is”, the definitions must be syntactically equal in the interface and the implementation.

8.8 Differences between Simple Mezzo and Core Mezzo

I have presented the formalization of Simple Mezzo; this is the language that the rest of the dissertation uses to formally reason about Mezzo, and also happens to match what the type-checker manipulates internally. I mentioned earlier a more restricted version of Mezzo, named Core Mezzo. This restriction has been mechanically proved sound [BPP14a]. The present section lists the few differences between the two languages.

New formalization for adoption/abandon

Perhaps the biggest difference is that Core Mezzo features a new, more powerful formalization for adoption and abandon. We wish to port the current implementation to the new formalization.

New design The formalization uses three new types.

- $x @ \text{adoptable}$ guarantees that x has an address in the heap, and is duplicable;
- $x @ \text{unadopted}$ guarantees that no other object currently owns x (i.e., the hidden field of x contains the NULL pointer), and is affine;
- $x @ \text{adopts } t$ guarantees that x lives in the heap, and that every adoptee of x has type t ; it is affine too.

The run-time representation does not change; the types of `give` and `take` do, however, change.

```
val give: [a] (consumes x: a, y: adopts a | consumes x @ unadopted) -> ()
val take: [a] (x: adoptable, y: adopts a) -> (| x @ a * x @ unadopted)
```

The $x @ \text{adoptable}$ permission is akin to $x @ \text{dynamic}$ (even though it may be made available for immutable blocks as well). The $x @ \text{unadopted}$ permission carries, in essence, ownership of the hidden, mutable field: a separate permission still represents ownership of the regular fields of the block. Finally, the $x @ \text{adopts } t$ removes the need for an “adopts clause” with a distinguished status; the permission also embodies the ownership of all the adoptees of x .

```

type adoptable_by (y: value) =
  adoptable

type unadopted_by (y: value) =
  unadopted

val dedicate [y: value] (consumes x: unadopted) : (| x @ unadopted_by y) =
  ()

val share [y: value] (x: unadopted_by y) : (| x @ adoptable_by y) =
  ()

val give : [a] (consumes x: a, y: adopts a | consumes x @ unadopted_by y) -> ()
  = give

val take : [a] (x: adoptable_by y, y: adopts a) -> (| x @ a * x @ unadopted_by y)
  = take

```

Figure 8.15: Alternative give and take operators

Restricting adoption in a library This new formalization also allows the user to define restricted versions of the `give` and `take` operations (Figure 8.15), so as to provide greater static safety, hence addressing a common concern about adoption/abandon (§3.3, “Discussion”).

The types `adoptable_by` and `unadopted_by` are to be made abstract via a module interface. This allows the programmer to make sure some objects are “committed”, and will only ever be adopted by a single adopter for their entire lifetime, hence ruling out more potential mistakes.

Figure 8.16 demonstrates this, using our running example of a graph structure.

Implementing this restriction was impossible with the former design, since a constructor type embodied both the ownership of the hidden field and the regular fields. Now, having two separate permissions allows one to “dedicate” the hidden field, and make sure it will only ever be used for a single adopter.

This new design also allows for writing more generic functions: a function that takes an argument of type `adopts (node a)` is applicable not just to a graph, but to any object that adopts nodes (§3.3).

The new design also allows us to get rid of “exclusive” which was, truly, only used for type-checking the `give` and `take` operations. The only slight possible drawback is that expressing nesting (§4.1) becomes more awkward, as we no longer can express the “this object is uniquely-owned” idiom. One could possibly use `x @ unadopted` for that purpose, but this ties together nesting and adoption/abandon, which are, in theory, orthogonal. What we would do is probably generate another permission when allocating a new block, of the form `x @ nesting`, meaning that `x` has a unique owner and hence can nest other objects.

Syntactic sugar A drawback of the new mechanism is that it requires the user to write more types. The graph example above shows that the user must provide the extra `adopts` type themselves; this is currently cumbersome, as it requires a name introduction along with a type/permission conjunction. A possible solution would be to insert `adoptable` and `adopts` permissions in the default case. Another solution would be to introduce a type/type conjunction, denoted `&`; that way, the user would merely need to write `x @ (node a & unadopted)`, which alleviates the syntactic burden. Similarly, in a definition, the user could do `data t = (A | B) & adopts u`.

This is still very much a work in progress.

Other differences

Data types The current proof of Core Mezzo currently does not feature algebraic data types. (A former, less modular version of the proof did, however, feature non-parameterized algebraic data types. This proof used integers instead of user-defined tags, though.)

```

alias node_of (g : value) =
  adoptable_by g

alias nodes_of (g : value) =
  list (node_of g)

data mutable node (g : value) a =
  Node {
    payload: a;
    successors: nodes_of g
  }

data mutable graph a = (g: Graph {
  roots: nodes_of g
}) | g @ adopts (node g a)

val g : graph int =
  let g = Graph {
    roots = nil
  } in
  (* g @ Graph { roots: Nil } * g @ adopts ⊥ *)
  let n = Node {
    payload = 0;
    successors = nil
  } in
  (* n @ Node { payload: int; successors: Nil } * n @ unadopted * adoptable *)
  dedicate [g] n;
  (* n @ Node { payload: int; successors: Nil } * n @ unadopted_by g * adoptable *)
  assert g @ adopts (node g int);
  share n;
  (* n @ Node { payload: int; successors: Nil } * n @ unadopted_by g * adoptable_by g *)
  assert n @ node_of g;
  g.roots <- cons [node_of g] (n, nil);
  n.successors <- cons [node_of g] (n, nil);
  give (n, g);
  assert g @ graph int;
  g

```

Figure 8.16: The graph example, using the new formalization of adoption/abandon

Primitive references Core Mezzo currently features references as a primitive, for lack of data type definitions. Primitives are annotated with a mode (immutable or mutable). This would correspond to two separate definitions in Simple Mezzo, one for mutable references, and the other one for immutable references.

Other unimportant differences Core Mezzo also does not formalize n -ary tuples, recursive functions, field overload and separate compilation.

8.9 Reflecting on the design of Mezzo

The initial discussion on related works and languages (Chapter 2) was intentionally general. Now that we have seen the inner workings of Mezzo in detail, it seems like the right time for reflecting in greater detail on how Mezzo relates to other languages, type systems and verification tools that have been designed recently.

L^3 As I mentioned already (§2.2), L^3 [AFM07] shares many of the core concepts of Mezzo. In both languages, the core idea is to distinguish pointers to an object (the $=x$ type in Mezzo) from the token that embodies the ownership of the corresponding heap fragment (the $x @ t$ permission in Mezzo). L^3 differs from Mezzo in that it is a low-level language, and does not feature high-level constructs such as n -ary tuples and parameterized algebraic data types. In that regard, it is closer to Core Mezzo, the language used to perform our proof of soundness.

The Mezzo typing rules can be seen as simpler versions of some rules from L^3 : we do not need to mention separate typing contexts for duplicable and non-duplicable values, since the distinction is implicit via the “is duplicable” predicate. Also, our rule NEW does not require generating a fresh existential; in Mezzo, one can always use DECOMPOSEANY later on. (In L^3 , the NEW rule looks like the combination of these two rules.) Another point to note is that expression variables can appear in types, thus obviating the need for type-level names.

For the sake of simplicity, the authors of L^3 chose to make several constructs appear as first-class values; they hope that a suitable compiler will be able to remove the computationally-irrelevant parameters. In contrast, our permissions do not exist at runtime, even in the formalization; one can pack a type along with a permission using the type/permission conjunction. Similarly, the authors of L^3 rely on a swap primitive to implement reads and writes. This incurs extra run-time operations in the case of reading from uniquely-owned references.

Finally, the authors of L^3 design the language around a “pluggable” mechanism for handling aliasing; they expect additional rules to be provided for the soundness conditions related to thawing. The authors only provide a trivial mechanism, which says that thawing is safe as long as there is no other variable in scope.

Static regions Static regions have been extensively studied, as I mentioned already (§2.1). Initially seen as a technical device for inferring lifetimes and generating more efficient code, regions became gradually used for reasoning about ownership in the presence of aliasing. If I own a region r , then I may freely alias pointers that “live” in region r , as long as there is a unique *capability* for region r .

A recurring difficulty has been to allow taking an object out of a region: in a sense, a static region is invalidated if an object is taken out. Indeed, one cannot take two objects at the same time, unless they are statically proven to be non-aliases. Most solutions are thus similar to nesting [Boy10]: one can take an object out, but this is enacted in the type of the region. This means that adoption is permanent and one cannot permanently take an object out of a region. This is also the point of view adopted by Charguéraud and Pottier [CP08].

Fähndrich and Deline [DF01, FD02] use, in their Vault language, a compile-time token known as a “key”. At every program point is a held-key set. Allocating a new variable generates a new key; freeing the variable asserts that the key is held, and deletes it. Functions may consume keys or change the state of keys. Type-checking is performed in a flow-sensitive manner; loops must be annotated unless the invariant can be inferred using a heuristic procedure. The type system is similar to the Capability Calculus [CWM99]: keys embody the unique *capability* to access a given resource, while program variables may be freely aliased. Syntactic sugar for function pre- and post-conditions is translated into an internal representation of function types, which is reminiscent of the translation phase of Mezzo. Existential types along with inductive predicates allow one to encode data structures containing keys, such as lists and trees.

To deal with aliasing, the authors propose an *adoption* mechanism, which allows one to transfer ownership of the *adoptee* to its *adopter*. Once adoption has been performed, just like in our own adoption/abandon, the adoptee gets a nonlinear type, meaning the programmer is free to alias the object. The nonlinear type does mention the “key” (the *capability*) associated to the adopter.

Operationally, and unlike our own adoption/abandon mechanism, Vault keeps a pointer from the adopter to the adoptee (Mezzo maintains a pointer in the opposite direction). This is due to the lifetime semantics of adoptees: in the context of a systems language, having a garbage collector is not an option. The lifetime of the adoptees is thus tied to the lifetime of their adopter, and freeing an adopter also frees all of the adoptees.

The system offers no way of permanently taking an adoptee from its adopter: rather, a focus construct allows the programmer to temporarily regain linear ownership of the adoptee using a lexically-scoped alias. The authors mention a possible autofocus extension that offers more flexibility than the lexical scope and allows the system to infer calls to unfocus.

While the mechanics may look similar, we believe our adoption/abandon procedure to be more flexible: the dynamic type is available even before adoption takes place, which has proven useful in the earlier graph example (Figure 3.14). We can also regain ownership of an object permanently, which is, to the best of our knowledge, a fresh contribution. This comes, naturally, at the cost of extra run-time operations, and would certainly not be suited to a low-level programming language. Indeed, there is a run-time cost for take which may be unacceptable

in some low-level contexts; moreover, the fact that one can keep `x@ dynamic` forever mandates the use of a garbage-collector, which may also be a no-go in some contexts.

Spec# Similar concerns about ownership, pre- and post-conditions as well as aliasing appear in verification tools. Spec# [BLSo4, BFL⁺11] spans several years of development, and aims at providing a *sound* and *modular* analysis for real-world programs. Programmers are expected to annotate programs with functional specifications that are then checked using a mixture of dynamic and optional static checking. Mezzo does not offer the option of providing functional specifications; an advantage, however, is that we have only one layer, that is, the type system.

The authors of Spec# designed a *superset* of the C# language that authors of programs can gradually opt into. As they try to model an object-oriented, real-world language, they face some problems that we did not have to treat in Mezzo: inheriting from multiple interfaces with conflicting method specifications, or modeling the various usages of exceptions for instance.

Adding exceptions in Mezzo would be tedious: since Mezzo tracks side-effects, functions would need to mention all the exceptions they may throw, along with the post-condition associated with each exception. In the case that the function has multiple throw-sites for the same exception, this means that the post-conditions associated to the throw sites need to match. We believe that, for the time being, data types have served us well, and that the lack of exceptions has not been a showstopper.

Some other issues are shared with Mezzo. Methods mention the set of variables they may modify: without this, static modular verification would be impossible. This is similar to a function in Mezzo that demands permissions in its signature. Invariants can be temporarily relaxed using their lexically-scoped `expose o` construct; this render further calls that depend on `o` impossible. An analogous phenomenon appears, though implicitly, in Mezzo.

```
data mutable t =
| A { x: ... | pred x }
| B ...

val f (y: t): () =
  match y with
  | A ->
    (* invariant holds *)
    ...
    y.x <- ...; (* breaks invariant *)
    ...
    (* invariant must hold *)
  | B ->
    ...
```

Manipulating a concrete type is analogous to breaking the invariant held by the nominal type. Type annotations, however, guarantee in the example above that invariants may only be broken locally within `f`.

Spec# also features an ownership discipline inspired by Ownership Types, where an object owns its field in a hierarchical manner. Spec# does not need to incorporate escape hatches: one may just not opt in to the verification methodology for a certain piece of the code.

CaReSL More recently, the authors of CaReSL [TDB13] propose a program logic based on separation logic. CaReSL properly accounts for both higher-order predicates and higher-order functions, is able to perform refinement proofs, and can model racy programs using a compare-and-swap operation.

CaReSL has a more powerful logic than Mezzo, since it features, among other things, higher-kinded predicates, that is, predicates of the form $p(x)$ which refer to *program variables* of the language. One of their first examples is a higher-order foreach function. We may write the function in Mezzo (Figure 8.17). In Mezzo however, while we can write higher-kinded predicates parameterized over terms, we can only quantify on non-arrow kinds, that is, on permissions p and q , not $p\ x$ and $q\ x$.

The authors introduce “necessary propositions” which are akin to “pure” propositions in separation logic or, in Mezzo, duplicable permissions. They present an `mkSync` function which is the equivalent of our earlier `hide` example (Figure 3.7). They use a system of *states*, where each state corresponds to a predicate on the state of an

```

open list

val rec foreach [a, p: perm, q: perm, r: perm]
  (consumes l: list (x: a | p),
   f: (x: a | r * consumes p) -> (| r * q)
   | r
  ): (| l @ list (x: a | q)) =
  match l with
  | Cons ->
    f l.head;
    foreach (l.tail, f);
  | Nil ->
    ()
end

```

Figure 8.17: The `foreach` function from CaReSL, transcribed in Mezzo [TDB13]

object; a thread that claims ownership of a piece of shared data may move it into another state. They prove the correctness of their `mkSync` higher-order function using a state-based reasoning on locks.

CaReSL is a program logic: there is, in my understanding, no support for automated reasoning; also, the language they consider is a core calculus, meaning that the work feels more of a foundational basis for designing later on user-friendly systems. It shares several concerns with Mezzo, though.

Other related works Gordon *et al.* [GPP⁺12] present an extension of C# that offers new ownership qualifiers: `writable` (the equivalent of “exclusive” in Mezzo), `immutable` (the equivalent of “duplicable”), `readable` and `isolated`. The last two qualifiers refer respectively to mutable data that is accessible in a read-only fashion, and to a unique reference into a cluster of objects. These last two qualifiers have no equivalent in Mezzo. The key insight is that one can promote a `writable` object into a `unique` one, provided that the input typing context satisfies certain restrictions. This alleviates completely the need for alias tracking. The system does seem to offer a great deal of flexibility while requiring very little annotations; accounting for generics, though, seems to require extra technical machinery.

After the original paper on Separation Logic [Reyo2], several tools dedicated to program verification using separation logic emerged [BCO05a, NDQC07, NR11, HIOP13, BCO04]. These tools, for the most part, operate on imperative languages, with a limited feature set. Most notably, higher-order is rarely taken into account. The technical comparison between our own subtraction procedure and the various entailment algorithms from the literature is to be found in §11.6.

Implementing Mezzo

9 Normalizing permissions

9.1	Notations	129
9.2	Requirements for a good representation	129
9.3	Prefixes	130
9.4	Normal form for a permission	132
9.5	Treatment of equations	135
9.6	Data structures of the type-checker	138
9.7	A glance at the implementation	138

10 A type-checking algorithm

10.1	Typing rules <i>vs.</i> algorithm	145
10.2	Transformation into A-normal form	145
10.3	Helper operations and notations	146
10.4	Addition	146
10.5	Type-checking algorithm	147
10.6	About type inference	149
10.7	Non-determinism in the type-checker	150
10.8	Propagating the expected type	151
10.9	A glance at the implementation	151

11 The subtraction operation

11.1	Overview of subtraction	153
11.2	Subtraction examples	154
11.3	The subtraction operation	156
11.4	Implementing subtraction	163
11.5	A complete example	168
11.6	Relating subtraction to other works	169
11.7	A glance at the implementation	170
11.8	Future work	174

12 The merge operation

12.1	Illustrating the merge problem	175
12.2	Formalizing the merge operation	179
12.3	An algorithmic specification for merging	180
12.4	Implementation	187
12.5	Relation with other works	191
12.6	A glance at the implementation	191
12.7	Future work	193

We made progress since the early pages of this thesis. The reader, hopefully, became acquainted with the various challenges that await the young PhD student trying to design a better type system (Part I). Their curiosity piqued, the reader got a fairly good preview of what writing programs in Mezzo looks like (Part II). But what good is a system if not for the formal rules? Part III provided Greek letters galore. A major problem remains, though: after more than a hundred pages, we still have no idea how to actually type-check a Mezzo program. This last part tackles the problem of implementing a type-checker for Mezzo.

One first issue is that, given all the subsumption rules that exist (§8.2) for Mezzo, there are many equivalent ways to represent a permission. Chapter 9 introduces a normalized representation for permissions, which is suitable for manipulation by a type-checker, and tames the complexity and profusion of subsumption rules.

Having a normalized representation allows us to sketch an actual type-checking algorithm in Chapter 10. The algorithm is very high-level and works at the expression level. It clearly states the permissions that are taken and returned for each type-checking step, which is an improvement over the declarative rules we saw earlier (§8.3). This algorithm, however, relies on two key operations, which are detailed in the subsequent chapters.

The first key operation (Chapter 11) relates to frame inference and entailment. These are crucial issues in separation logic; Mezzo shares the same problems. However, in the context of functional languages, interesting variations surface: polymorphism, first-class functions bring extra difficulties related to quantifiers. The quantification over permissions is akin to second-order logic, which makes inferring polymorphic instantiations challenging. Finally, the fact that some permissions may be duplicable and some others may not be constitutes the icing on top of the cake.

The second key operation (Chapter 12) relates to the join problem over abstract shape domains. In essence, whenever control-flow diverges, the type-checker needs to compute the permission that is valid at the “join point”. Some languages chose to require annotations after every if block; Mezzo has an algorithm for inferring these. Again, the presence of a powerful subtyping relation, along with quantifiers and polymorphism, makes for an interesting problem.

Equipped with algorithms that can tackle these two key problems, the high-level type-checking algorithm is happy to proceed, and type-check Mezzo programs.

9. Normalizing permissions

The various algorithms that I am about to flesh out in the upcoming chapters naturally manipulate permissions. We thus need a representation for permissions that allows our algorithms to operate efficiently; the representation should also abstract away some features of the type system, such as the various ways of representing a permission; the representation should also be well-suited to a formal discussion: I intend, after all, to prove that the algorithms are, at the very least, correct.

In this chapter, I start by identifying the properties that a good representation should have, and the requirements that a type-checking algorithm places on such a representation. I then devise a *normalization* procedure, which allows us to rewrite a permission into a normal form and provides sufficient invariants for a type-checking algorithm to operate properly. This chapter is only concerned with the normal representation of a permission and the invariants the representation verifies: the actual type-checking algorithm is the topic of Chapter 10.

This chapter also justifies that the normalization procedure is correct using the subsumption rules of Mezzo (Figure 8.1), in the sense that it provides a deterministic procedure for applying them. Finally, I briefly mention how I implemented this representation in our prototype type-checker.

9.1 Notations

Internal syntax Expression snippets are, naturally, written using the surface syntax of Mezzo. Types, however, even if in monospace font, are now written in the internal syntax, until the end of the thesis, and unless specified otherwise.

Data type projection We define an additional notation for *data type projections*. If data type t is defined as :

$$\text{data mutable? } t(\vec{X} : \vec{\kappa}) = \dots | A \{\vec{f} : \vec{t}\} | \dots$$

then we define the *projection* of the *type application* $t \vec{u}$ on constructor A to be:

$$t \vec{u} / A = [\vec{u} / \vec{X}] (A \{\vec{f} : \vec{t}\})$$

that is, the concrete type for branch A where all formal parameters \vec{X} have been replaced with the effective parameters \vec{u} of the type application. This is a more concise notation than the sentence “ $A \{\vec{f} : \vec{t}\}$ is an unfolding of $t \vec{u}$ ” which we used earlier.

9.2 Requirements for a good representation

The difficulties in type-checking Mezzo programs revolve around two main points: finding an algorithmic presentation where a type-checking step takes P and returns P' , and coming up with an efficient representation for

P. Without delving into the details, let us run over a few constraints that the algorithm and the underlying representation should satisfy. These will help us design a suitable representation for permissions. The next chapter will detail the actual algorithm.

- A permission may be represented using several equivalent ways: we have seen several times that one may *decompose* a permission $z @ (\text{int}, \text{int})$ into $z @ \exists(x, y : \text{value}) (=x, =y) * x @ \text{int} * y @ \text{int}$, and *recompose* it.
- Function types can also be represented in various equivalent ways: for instance,

$$\begin{aligned} & x @ \forall a, \forall (y : \text{value}). (=y \mid y @ a) \rightarrow () \\ \equiv & x @ \forall a, a \rightarrow () \end{aligned}$$

As a language of ML lineage, Mezzo deals heavily with first-class functions: any type-checking algorithm should thus be able to deal with either one of the two equivalent representations above without difficulty.

- Conjunctions of the form $x @ t_1 * x @ t_2$ may be inconsistent, redundant, or may simplify into another conjunction (using, for instance, `UNIFYTUPLE`).
- Quantifiers, both universal and existential, are pervasive in Mezzo. They are either provided by the user or introduced automatically by the translation from the surface syntax into the internal syntax (Chapter 7). Should these quantifiers be *opened*? Is there a particular constraint on the *order* for opening the binders?
- The presence of universal quantification over types or permissions amounts to second-order logic, thus making type-checking akin to proof search in this logic. Proper treatment of quantifiers is thus doubly important (§9.3).
- Mezzo features equations of the form $x = y$ where x and y are variables at kind `value`; the typing rules enforce the fact that one may use either x or y indifferently, whenever $x = y$ is present. We need special provisions to faithfully implement this behavior (§9.5).
- As we saw in the preceding section, several permissions for the same variable may be present in a conjunction; an algorithm will thus need to *explore* several possible branches when tasked with a problem to solve.
- Some permissions can be duplicated, while some others cannot: a type-checking operation may thus *consume* portions of the conjunction while leaving others intact.

9.3 Prefixes

A first issue we need to tackle is that of proper binding. Whenever we performed step-by-step type-checking of examples (Chapter 3), we wrote permissions where names were implicitly bound. That is, we never actually specified where x was bound when talking about a “current permission” $x @ t$. Moreover, many examples of permission rewriting carefully elide the issue of name binding.

```
(* x @ Cons { head: a; tail: list a } *)
(* x @ Cons { head = hd; tail = tl } * hd @ a * tl @ list a
```

Figure 9.1: A sample permission recombination

In the example from Figure 9.1, it is unclear how `hd` and `tl` are bound.

The type-checking rules are equipped with an environment K where existentially-quantified and universally-quantified variables are opened via \exists -elimination and \forall -introduction rules. These variables are all *rigid*. This representation is well-suited for declarative rules, but we need more binding structure for properly implementing a type-checker for Mezzo.

This section examines the question of name binding in a more thorough manner and devises a more powerful notion, called a *prefix*, that is better suited for the upcoming algorithmic specification of type-checking.

An example

While the present chapter is *not* about a type-checking algorithm, let me state nonetheless how an algorithm will want to deal with the classic snippet from Figure 9.2, which appeared repeatedly under one form or another in the section about lists (§3.2). This helps justify the design of the “prefix”.

```

1  (* x @ list a *)
2  match x with
3  | Cons ->
4    let tail = x.tail in
5    let l = length tail in
6    ...

```

Figure 9.2: Sample excerpt from the `list::length` function

In order to type-check the `Cons` branch, the type-checker expands permissions just like in the sample permission recombination from Figure 9.1.

Now that we have seen the actual subsumption rules of Mezzo, we know that this recombination corresponds to an application of the `DECOMPOSEBLOCK` subsumption rule. This rule introduces *existential* quantifiers. Formally, before line 4, we possess:

$$\exists(hd : \text{value}, tl : \text{value}). \text{Cons } \{ \text{head} = hd; \text{tail} = tl \} * hd @ a * tl @ \text{list } a$$

In order to type-check line 4 successfully, we want to add `tail = tl` into the environment. This is achieved by an instantiation of the type-checking rule `READ` (Figure 8.5), where t is chosen to be `=tl`, followed by an application of the rule `LET`, where t_1 is `=tl` as well.

We do need, however, to open the existential binders for this to succeed. Otherwise, adding `tail = tl` would not make sense, for `tl` would not be bound.

Rigid vs. flexible

The variable `tl` is existentially-quantified. In essence, it is not something that we can choose. Therefore, when we open the binder that corresponds to `tl`, we bind `tl` as a *rigid* variable, that is, a variable that will *not* instantiate to something else.

Conversely, in some situations, the type-checker will want to *infer* the value of a given variable. This happens, for instance, whenever performing polymorphic function calls (Figure 9.2, line 5). Even in the absence of user-provided polymorphic quantification, we saw (Chapter 7) that the translation from Mezzo to Simple Mezzo introduces universal quantifications at kind value.

```

(* length @ [b, x: value] (=x | x @ list b) -> (int | x @ list b) *)
(* length @ (=?x | ?x @ list ?b) -> (int | ?x @ list ?b) *)

```

Figure 9.3: Introducing flexible variables for polymorphic function call inference

The declarative rules assume that one always knows when to instantiate a universal quantifier ahead of time. In practice, though, we need to *guess* the proper parameter for the instantiation that will make type-checking the expression succeed. For this inference to happen, we use the standard technique of *flexible variables*: whenever we cannot guess immediately the suitable instantiation of a variable, we introduce a flexible variable whose value will be decided upon later, once we have enough information to properly infer the right instantiation.

In the example from Figure 9.3, the universally-quantified variables `b` and `x` are opened as flexible variables, which are denoted as `?b` and `?x` respectively. The algorithm then proceeds with matching the argument `tail` with `(=?x | ?x @ list ?b)`. It finally picks `?x = tl` and `?b = a`.

We introduce *prefixes*, which keep track of the order in which variables are bound, as well as the nature of the binding. The prefix keeps track of whether a variable is rigid or flexible, instantiated or not, and of the order the

$q ::=$	$\mathcal{R}(x : \kappa)$	binder
	$\mathcal{F}(x : \kappa)$	rigid variable
	$\mathcal{F}(x : \kappa = \tau)$	flexible variable
		instantiated flexible variable
$\mathcal{V} ::=$	$\text{nil} \mid q, \mathcal{V}$	a prefix is a list of binders

Figure 9.4: Syntax of prefixes

variables were introduced in. The syntax of prefixes is presented in Figure 9.4. The type variables are introduced with a kind κ , which, if missing, is assumed to be type.

Definition 9.1 (Prefixed permission). *A prefixed permission is written $\mathcal{V}.P$. All free variables in P are bound in \mathcal{V} .*

Therefore, whenever stepping through program points, the type-checker carries a *prefixed permission* $\mathcal{V}.P$. This not only clarifies the fuzzy discipline of binding that we used throughout the examples, but also allows for proper tracking of flexible variables.

A type-checking step not only transforms the permission into another, but also transform the prefix into another one.

Definition 9.2 (Prefix well-formedness). *A prefix is well-formed if, for all instantiated flexible variables*

$$\mathcal{V}_1; \mathcal{F}(X = t); \mathcal{V}_2$$

we have $\mathcal{V}_2 \# t$, $X \# t$ and $FV(t) \subseteq \mathcal{V}_1$.

Definition 9.3 (Prefix refinement). *A prefix \mathcal{V}' is more precise than \mathcal{V} , which we write $\mathcal{V}' \leq \mathcal{V}$ if \mathcal{V}' can be obtained from \mathcal{V} by:*

- instantiating flexible variables,
- adding flexible variables.

Wording differently, \mathcal{V}' is more precise than \mathcal{V} if it contains exactly all the rigid variables from \mathcal{V} .

Definition 9.4 (Prefix restriction). *If $\mathcal{V}' \leq \mathcal{V}$, we define $\text{restrict}(\mathcal{V}', \mathcal{V})$ to be a well-formed subset of \mathcal{V}' that contains all the rigid variables from \mathcal{V} .*

Our type-checker (§9.7) provides an efficient, deterministic implementation of this operation.

In the rest of the discussion, we always manipulate well-formed prefixes. The other notions of refinement and restriction will be used later on (Chapter 11).

9.4 Normal form for a permission

As we mentioned earlier (§9.2), a good representation should be able to deal with the various ways of representing a permission. We introduce a normalization procedure for this purpose.

Normalization works on prefixed permissions and comes in two variants: $\overset{\forall,*}{\rightsquigarrow}$, which applies all rules transitively, except for those marked with $\overset{\exists}{\rightsquigarrow}$, and the converse variant, named $\overset{\exists,*}{\rightsquigarrow}$, which applies all rules save for those marked with $\overset{\forall}{\rightsquigarrow}$. The former is called \forall -normalization, while the latter is called \exists -normalization.

The reason why we have two separate relations will be exhaustively commented during §11.3; in short, when trying to build a typing derivation, the *hypothesis* needs to be \exists -normalized, while the *goal* needs to be \forall -normalized. Performing this normalization is part of standard proof-search techniques and is a necessary step to obtain good success rates in our proof search procedure, called *subtraction*.

The normalization rules are presented in Figure 9.5. Normalization rules are taken to *apply* modulo equalities, commutativity and associativity. This matches closely the implementation, where equalities, commutativity and associativity are handled transparently. Normalization takes care of the following points.

$$\begin{array}{ll}
\mathcal{V}.x @ (x | P) & \rightsquigarrow \mathcal{V}.x @ t * P \quad (1) \\
\mathcal{V}.x @ (\vec{t}) & \rightsquigarrow \exists \mathcal{V}(\vec{y} : \text{value}) x @ (= \vec{y}) * \vec{y} @ \vec{t} \quad (2a) \\
\mathcal{V}.x @ A \{ \vec{f} : \vec{t} \} & \rightsquigarrow \exists \mathcal{V}(\vec{y} : \text{value}) x @ A \{ \vec{f} = \vec{y} \} * \vec{y} @ \vec{t} \quad (2b) \\
\mathcal{V}.x @ \exists (X : \kappa) t & \rightsquigarrow \exists \mathcal{V}(X : \kappa) x @ t \quad (3) \\
\mathcal{V}.x @ (= \vec{y}) * x @ (= \vec{y}') & \rightsquigarrow \mathcal{V}.x @ (= \vec{y}) * \vec{y} = \vec{y}' \quad (4a) \\
\mathcal{V}.x @ A \{ \vec{f} = \vec{y} \} * x @ A \{ \vec{f} = \vec{y}' \} & \rightsquigarrow \mathcal{V}.x @ A \{ \vec{f} = \vec{y} \} * \vec{y} = \vec{y}' \quad (4b) \\
\left\{ \begin{array}{l} A \text{ is a constructor of } t \\ \mathcal{V}.x @ A \{ \dots \} * x @ t \vec{t}' \end{array} \right. & \rightsquigarrow \mathcal{V}.x @ A \{ \dots \} * x @ t \vec{t}' / A \quad (5) \\
\mathcal{V}.x @ \forall (\vec{X} : \vec{\kappa}) t \rightarrow u & \rightsquigarrow \forall \mathcal{V}(\vec{X} : \vec{\kappa}) \forall (r : \text{value}) (=r | r @ t) \rightarrow u \quad (6a) \\
\left\{ \begin{array}{l} x @ \forall (\vec{X} : \vec{\kappa}) \forall (r : \text{value}) (=r | r @ t) \rightarrow u \\ \mathcal{V}.r @ t \rightsquigarrow^* Q \end{array} \right. & \rightsquigarrow \forall \mathcal{V}(\vec{X} : \vec{\kappa}) \forall (r : \text{value}) (=r | Q) \rightarrow u \quad (6b) \\
\mathcal{V}.x @ \forall (\vec{X} : \vec{\kappa}) \forall (r : \text{value}) (=r | \exists (\vec{X}' : \vec{\kappa}') Q) \rightarrow u & \rightsquigarrow \forall \mathcal{V}(\vec{X} : \vec{\kappa}) \forall (r : \text{value}) \forall (\vec{X}' : \vec{\kappa}') (=r | Q) \rightarrow u \quad (6c) \\
\mathcal{V}.x @ \forall (X : \kappa) t & \rightsquigarrow \forall \mathcal{V}(X : \kappa) x @ t \quad (7) \\
\left\{ \begin{array}{l} P \rightsquigarrow P' \text{ and } Q \rightsquigarrow Q' \\ \mathcal{V}.P * Q \end{array} \right. & \rightsquigarrow \mathcal{V}.P' * Q' \quad (8) \\
\left\{ \begin{array}{l} A \text{ is the only constructor of } t \\ \mathcal{V}.x @ t \vec{u} \end{array} \right. & \rightsquigarrow \mathcal{V}.x @ t \vec{u} / A \quad (9) \\
\mathcal{V}, \mathcal{F}(X : \kappa = \tau), \mathcal{V}'.P[X] & \rightsquigarrow \mathcal{V}, \mathcal{V}'.P[\tau] \quad (10) \\
\left\{ \begin{array}{l} t \text{ is exclusive} \\ \mathcal{V}.x @ t \end{array} \right. & \rightsquigarrow \mathcal{V}x @ t * x @ \text{dynamic} \quad (11)
\end{array}$$

Figure 9.5: Normalization

No nested permissions All permissions nested inside structural types using $(t | P)$ are floated out into the outer conjunction (1).

This is the direct application of MIXSTAR. Having this invariant simplifies the reasoning throughout the present thesis, as we don't have to consider the case where the permission we are looking for is "stashed" inside another one. It is also of practical interest, since it simplifies the implementation.

Expanded form (Only in \exists -normalization.) All structural permissions, that is, tuples (2a) and constructors (2b) are *expanded*, meaning that their fields are all singleton types. For instance, we always expand $x @ (\text{int}, \text{int})$ into

$$\exists (y : \text{value}, z : \text{value}). x @ (=y, =z) * y @ \text{int} * z @ \text{int}$$

These rules are only triggered if the fields are not all singleton types already; that way, these rules are not applied indefinitely.

Rules (2a) and (2b) are a consequence of DECOMPOSEBLOCK, DECOMPOSETUPLE and EXISTSATOMIC. They ensure that every single tuple or record field is given a name. This simplifies the type-checking of field accesses: type-checking $\text{let } x = y.f$ merely amounts to adding $z = x$ to the current permission, where z is the singleton type for the f field of y .

Existential variables (Only in \exists -normalization.) Existential variables are hoisted out (3) using EXISTSATOMIC.

The two rules (2a) and (2b) also reveal "hidden" existential quantifiers, which are then hoisted out when (3) kicks in. (Rules (2a) and (2b) may apply under a $*$ conjunction via (8), for instance, meaning that existential quantifiers may appear at any depth.) The reason why existential quantifiers must be hoisted is the topic of §11.3.

No redundant permissions Conjunctions such as $x @ \text{Nil} * x @ \text{list } a$, are redundant. We simplify these.

Rules (4a) and (4b) simplify conjunction of tuple types or constructor types that refer to the same variable by applying `UNIFYTUPLE` and `UNIFYBLOCK`. Rule (5) simplifies a conjunction of a constructor type and a matching nominal type using `UNIFYAPP`. These rules assume an expanded form, which can be attained by applying rules (2a) and (2b).

Let us illustrate these simplifications with more concrete examples. The following conjunction is consistent:

$$x @ \text{list } a * x @ \text{Cons } \{ \text{head} = h; \text{tail} = t \}$$

It can be simplified into:

$$x @ \text{Cons } \{ \text{head} = h; \text{tail} = t \} * h @ a * t @ \text{list } a$$

Another example is

$$x @ (=y_1, =y_2) * x @ (=z_1, =z_2)$$

which is simplified into

$$x @ (=y_1, =y_2) * y_1 = z_1 * y_2 = z_2$$

Function types (Only in \forall -normalization.) All function types have a domain that is a singleton type along with a permission, which is itself normalized. This results in universal quantifiers being hoisted out of function domains.

Rule (6a) ensures the domain of a function type is a singleton type along with a permission, rule (6b) ensures that the said permission is itself normalized, and rule (6c) hoists existential quantifiers resulting from (6b) into universal quantifiers over the whole function type (`COMMUTEARROW`). Rules (6a) and (6b) are a combination of `COARROW`, along with `DECOMPOSEANY`. Combined with rule (7), these make sure universal quantifiers can be hoisted out (`FORALLATOMIC`). Again, the subsequent sections will illustrate the need for these somehow technical rules.

Universal variables (Only in \forall -normalization.) Universal variables are hoisted out (7) via `FORALLATOMIC`.


Recursive The recursive normalization (8) is a consequence of `COSTAR`.

Simplify one-branch types Data types which possess a unique branch are expanded into the corresponding structural type (9) via `UNFOLD`, which allows for further normalizations, of course.

Substitute instantiated flexible variables We ensure that whenever X instantiates onto τ , all occurrences of X are replaced with τ (10). (The $P[X]$ notation captures *all* occurrences of X .) This defines the meaning of flexible variables. More details about this part appear in §9.5.

Ensure dynamic is always available Whenever a new exclusive permission appear in a permission, whether via an addition or via a flexible variable instantiation, we need to make sure that, should t be exclusive, $x @ \text{dynamic}$ also appears. This is the purpose of rule (11). Naturally, this rule is understood to apply only in the case where $x @ \text{dynamic}$ is not present already.

Lemma 9.5. *Assuming no recursive, one-branch data types are defined, both \exists -normalization and \forall -normalization are strongly normalizing.*

Proof. Mezzo does not allow equirecursive types via flexible variable instantiations, and does not allow recursive alias definitions either. Rule (10) hence terminates. The other rule that may not terminate is (9), when one-branch data types are defined *and* are recursive. We assume no such data types are defined. Mezzo currently does not enforce this¹. 

Conjecture 9.6. *Normalization is confluent, modulo the usual properties (associativity and commutativity), and modulo equalities (`EQUALSFOR EQUALS`)*

¹ Landin's knot (Figure 4.14) is an example of a legitimate program that uses a recursive, one-branch data type. The discussion in §11.4 explains the challenges in handling such cases.

The current implementation may, in some cases, figure out that a conjunction is inconsistent, because, say, one has two exclusive permissions for the same variable. The programmer can thus return any type, since the code fragment is unreachable. This is rarely leveraged by the Mezzo programmer, and has not been formalized yet.

At this stage, we wish to state some results about normalization. Our subtyping relation \leq , however, has no knowledge of instantiated flexible variables (this is an implementation tool); we thus need to perform all substitutions of instantiated flexible variables before stating subtyping results. This is done using rule (10) of normalization. I introduce a special notation for this, where $\overset{(10),*}{\rightsquigarrow}$ means repeated application of normalization rule (10) until it can no longer apply.

Definition 9.7 (Full substitution). *If $\mathcal{V}.P \overset{(10),*}{\rightsquigarrow} \mathcal{V}'.P'$, then the full substitution $\mathcal{V} \times P$ is defined to be $\mathcal{V}'.P'$.*

Theorem 9.8 (Correctness of normalization). *Normalization is correct with regard to the subsumption rules of Mezzo: once flexible variables have been substituted via rule (10), normalization yields a supertype of the original permission. That is, if $\mathcal{V}.P \overset{*}{\rightsquigarrow} \mathcal{V}'.P'$, then:*


$$\mathcal{V} \times P \leq \mathcal{V}' \times P'$$

A point to note is that both \mathcal{V}' and \mathcal{V}'' may contain rigid *and* flexible variables; when stating the result, all variables are taken to be rigid. Intuitively, the subtyping result holds regardless of future instantiation choices for the flexible variables.

I also need to state an important remark that will be used later on (Chapter 11). Intuitively, applying a non-reversible rule (e.g. FOLD) is a bad idea, as it will *lose* information about the current permission. We thus need to ensure that normalization does not weaken our assumptions.

Remark 9.9. *Normalization only ever applies reversible rules. That is, if $\mathcal{V}.P \overset{*}{\rightsquigarrow} \mathcal{V}'.P'$, then:*

$$\mathcal{V} \times P \equiv \mathcal{V}' \times P'$$

Proof. The only case that is not trivially reversible concerns rules (6a) and (6b) which rely on COARROW. This latter rule can be applied in the converse direction, though, since the transformations that operate on the domain are themselves reversible. 

Normal form for a prefixed permission

Definition 9.10 (Full normalization). *We say that a prefixed permission $\mathcal{V}.P$ is fully normalized if:*

- *P has been \exists -normalized, and*
- *any existential quantifiers have been opened as rigid variables.*

The type-checker algorithm (Chapter 10) as well as the subtraction (Chapter 11) and merge (Chapter 12) operations take, and return, fully normalized permissions. A fully normalized permission is thus the structure our algorithms work on.

Moving quantifiers into the prefix is an explicit operation that is performed at key steps by the type-checking algorithm. Normalization thus does not take care of this.

9.5 Treatment of equations

Equations have a special status in Mezzo; we have seen equations between program variables, of the form $x = y$. These get a special treatment in the type-checker. Due to the use of flexible variables, which may be instantiated, there is another kind of equations, of the form $\mathcal{F}(X = t)$. This section describes the way equations are handled and propagated in our representation of permissions.

Two kinds of equations

A prefixed permission $\mathcal{V}.P$ contains two kinds of equations.

- First, equations between *rigid* program variables, of the form $x = y$. These equations appear in the *current permission* P (“rigid-rigid equations”); they are a feature of our type system.
- Second, equations between a *flexible* variable and a type, of the form $\mathcal{F}(X : \kappa = \tau)$. These appear in the *prefix* \mathcal{V} of the current permission (“flexible equations”), and are an implementation technique that facilitates inference of polymorphic instantiations.

This latter variety of equations also covers $\mathcal{F}(X = y)$, that is, the case where κ is value and τ is a type variable at kind value. This case corresponds to the inference of a type variable at kind value.

The reason we need to distinguish between two kinds of equations is that not only do they correspond to different underlying mechanisms, but they also have different behaviors. Let us take a (slightly artificial) example to illustrate this.

```
val _ =
  let x = e1 in
  let y = e2 in
  let f (| x = y): () = () in
  let g [a] (z: a): a = z in
  let ret = g 1 in
```

While type-checking the *body* of f , we need to *assume* that $x = y$, since this permission is found in the *argument* of the function. Once we are done type-checking the body of f , we must *forget* this assumption, since it is known to hold only in the body of f .

Conversely, calling g introduces a flexible variable $\mathcal{F}(X)$, and returns a permission $z @ X$. In order to successfully type-check this function call, the type-checker may pick $\mathcal{F}(X = \text{int})$. This equation must be retained, as it must hold for the remainder of ret ’s scope.

Rigid-rigid equations

We now turn to equations of the form $x = y$, where x and y are rigid variables. For these equations, x and y are necessarily of kind value². One may recall that $x = y$ is syntactic sugar for $x @ =y$. These equations are thus regular permissions, and stored in the P part of the current conjunction $\mathcal{V}.P$.

Take, for instance, the typing rule for function bodies (FUNCTION). The final permission after type-checking the body is discarded: this means that rigid-rigid equations are discarded too. This corresponds, in the example above, to the equation $x = y$ that is discarded upon exiting the body of f .

These equations have a special meaning which is embodied by rule EQUALSFOREQUALS. The rule tells us that we can use both names interchangeably.

We implement this behavior by using an underlying persistent union-find data structure, meaning that our implementation handles transparently the fact that x and y really are the same thing. In the remainder of the discussion, we assume that these equations are handled under-the-hood, that is, that EQUALSFOREQUALS may be applied at any time, whenever needed. I used this fact already, when I mentioned that the normalization is taken to operate modulo EQUALSFOREQUALS.

One can note, before going further, that in $\mathcal{V}.P$, no scoping problem arises with equations between rigid variables. Indeed, the variables are always bound within the prefix \mathcal{V} ; since $x = y$ is a permission in P , both x and y are always well-scoped.

Flexible instantiations

The other variety of equations stems from instantiations of flexible variables stored in the prefix \mathcal{V} , of the form $\mathcal{F}(X = t)$.

² Mezzo does not have GADTs, meaning that the user cannot talk about a type equality between types at kind type.

The difficulty lies in ensuring that instantiations are legal, that is, that flexible variables instantiate onto something well-scoped. What we want, in essence, is for the prefix to be well-formed:

$$\mathcal{V}_1; \mathcal{F}(X = t); \mathcal{V}_2 \Rightarrow \mathcal{V}_2 \# t \quad \text{and} \quad X \# t$$

Here is an example of an illegal instantiation, where X is the program (type) variable x and t is the program (type) variable y .

```
val _ =
  let flex x: term in
  let y = () in
  assert x = y
```

The let-flex construct has been mentioned in the syntax reference (Chapter 6) and used once (Figure 3.11). It introduces a new flexible variable at the desired kind.

This example fails, rightly so, with the error message “could not prove equation: ?y = x”. The reason is, the flexible variable y was introduced *earlier* than the rigid variable x . Before attempting to prove the assertion, the current, prefixed permission is thus:

$$\mathcal{F}(y : \text{value}), \mathcal{R}(x : \text{value}). x @ ()$$

The type-checker cannot pick $x = y$ to prove the assertion, because it would be an ill-scoped equation with regard to the order of variables in the prefix. The instantiation above is thus illegal. Accepting it would be unsound for the type-checker.

We thus check that a flexible variable instantiation makes sense. This is done using levels [PRO5]. In the special case that a flexible variable instantiates onto another flexible variable, the operation is always legal, since one can, conceptually, always swap the two variables. This amounts to changing the level of the younger variable to be that of the older one; this is also what happens in the implementation.

Flexible variable substitution is part of normalization, through rule (10). It ensures that we never see an instantiated flexible variable, and always deal with the type it instantiated to.

Flexible variables are handled using another, different union-find data structure. Flexible variables are thus handled transparently, in the sense that the rule (10) is applied implicitly in the type checker’s representation.

Flexible equations

We now turn to the special case of permissions of the form $x = y$ (i.e. $x @ = y$) when at least x is flexible.

- When *gaining* the permission $x = y$, we want to enact this fact in the prefix, by changing $\mathcal{F}(x : \text{value})$ into $\mathcal{F}(x : \text{value} = y)$. Indeed, x is a flexible variable, and gaining the permission means that we just became aware of an instantiation choice. This choice is incomplete: we may actually want to pick $\mathcal{F}(x : \text{value} = z)$, so as to gain $x = z$ beyond $x = y$.
- When *trying to prove* the equation $x = y$, we may want to choose $\mathcal{F}(x : \text{value} = y)$ so as to satisfy the desired goal. Similarly, this is an incomplete choice; we may want $\mathcal{F}(x : \text{value} = z)$, as $x = y$ may be available elsewhere.

In essence, equations are always eagerly taken into account into our representation; equations have no first-class status, since the implementation relies on a union-find structure to materialize equations. Therefore, an equation is either immediately reflected by a unification operation, or discarded. This is a limitation of the current implementation.

Since this is a case of flexible variable instantiations, scoping issues arise. When gaining $x = y$, this permission may be inconsistent, because the prefix \mathcal{V} is of the form $\mathcal{F}(x : \text{value}), \mathcal{R}(y : \text{value})$. Symmetrically, a permission $x = y$ may be impossible to obtain because picking $\mathcal{F}(x = y)$ is illegal.

Mezzo could mark the environment as inconsistent in the first situation; right now, this is not implemented and permissions are either dropped or goals fail to be proved.

9.6 Data structures of the type-checker

The type checker of Mezzo is written in OCaml. Representing a current permission $\mathcal{V}.P$ is done using the following data structures.

Persistent union-find We use a union-find structure to keep track of equations between rigid variables, that is, permissions of the form $x = y$. The union-find is persistent, as it facilitates backtracking and branching on disjunctions. Moreover, the union-find maps the representative of an equivalence class x to a list of types, which represent the permissions available for x .

Floating permissions Permissions that are not of the form $x @ t$ are not attached to a program variable; they are abstract permissions $X \vec{T}$. We dub them “floating permissions”, and store them in a separate list³.

Flexible variables Flexible variables from the prefix are implemented using another union-find. We use a separate structure in order to *propagate* flexible variable instantiations: unlike rigid equations, flexible variable instantiations must be kept throughout the rest of the program. We allow inserting flexible variables at an arbitrary position in the prefix. The implementation uses a less precise, but easier-to-handle data structure than the list of binders described formally in the prefix. This has consequences on how the restrict function is implemented.

Levels In order to efficiently check the non-occurrence premise of a rule such as INSTANTIATION, we use levels [PRO5]. A level is a natural number associated to a rigid or flexible binding; comparing levels allows one to determine whether a variable can be instantiated to another one. This also allows for an efficient implementation of the “is local” check during merges (Chapter 12).

In essence, the level is bumped every time a new \mathcal{R} quantifier is introduced after a \mathcal{F} . Instantiating a flexible variable into a type t is only possible if the level of t (that is, the maximum level of the type variables in it) is not greater than the level of the flexible variable itself.

Lazy lists The type-checker sometimes needs to *explore* branches; we use streams, in the form of lazy lists, to represent a lazy set of solutions. We have heuristics (§11.4) that make sure the first choice in the list is, most of the time, a solution to the type-checking problem we were trying to solve. That way, we don’t uselessly force the computation of other paths in the exploration space.

9.7 A glance at the implementation

This section briefly describes the data structures used for representing a “current permission”. I talk a little bit about how equations between rigid variables and flexible variable instantiations are handled.

Handling of variables

Figure 9.6 presents selected definitions from the Mezzo type-checker. These definitions are located in `typing/TypeCore.ml`.

There is one single syntactic category, that is, one single datatype for all types at any kind. There is a separate category (in `typing/ExpressionsCore.ml`) for expressions.

Type variables are handled using a locally nameless binding-style, where variables are either *open* or *bound*. A bound variable (line 6) is represented using a DeBruijn index; an open variable (line 7) is represented by a globally-unique atom.

An open variable may be either rigid or flexible. This is described by the `var` type (line 12). Both the `point` type and the `flex_index` type represent globally-unique atoms, but they relate to different data structures.

The environment

Figure 9.7 presents selected bits from the definition of environment. The `env` type (line 5) represents the $\mathcal{V}.P$ “current permission” we described earlier.

³ Equations of the form $x @ t$ where x is flexible could be conceivably stored in this list as well; however, the inference mechanism does not know how to deal with these. These permissions are thus silently dropped. This yet another limitation of the current approach.

```

1  type flex_index = int
2  type db_index = int
3
4  type typ =
5      ...
6      | TyBound of db_index
7      | TyOpen of var
8      ...
9      | TyTuple of typ list
10     ...
11
12  and var =
13      | VRigid of point
14      | VFlexible of flex_index

```

Figure 9.6: Representation of variables

- The state field contains a persistent union-find data structure which corresponds to rigid bindings. The union-find maps an equivalence class to instances of the binding type. The `PersistentUnionFind` module offers the following (abridged) interface:

```

val find: point -> 'a state -> 'a
val union: point -> point -> 'a state -> 'a state

```

Per the `var` type (Figure 9.6), a rigid variable is a point; thus, a client of this code is required to go through the `find` function so as to obtain the binding associated to a rigid variable. This ensures that equalities are handled transparently: whenever two variables x and y are equal (i.e. we have $x = y$), the corresponding classes are union'd.

- The `flexible` field maps `flex_index`'s to flexible descriptors. Since the `flex_index` type is `int`, this is an `IntMap`. An earlier version of the type-checker used a list instead of a map; a map is required for implementing some operations on flexible variables which I present later.
- The `floating_permissions` field keeps track of permissions of the form $X \vec{T}$, which cannot be held in the state field since they are not anchored to any rigid variable.
- The `last_binding` field remembers whether the last binding was rigid or flexible; this is used in conjunction with `current_level` for properly implementing levels⁴.

Variable descriptors

Rigid variables are associated, via the `state` field, to a binding (line 31), which is made up of a `var_descr` (line 25), and a `rigid_descr` (line 25).

Flexible variables, conversely, are only associated to a `var_descr` if they are uninstantiated, or to a `type`, if they are instantiated (line 17). Worded differently, a flexible variable ceases to exist once instantiated, and from then on only serves as an indirection for a type.

Thus, the `var_descr` type (line 25) describes information that is relevant for any sort of variable: its kind, its level, and its fact, if any (variables at kind value have no fact). In the case of a plain variable (e.g. `a`) its fact will be a constant mode; in the case of a defined type (e.g. `list`), it will have an arrow kind and a more complex fact.

The `rigid_descr` type describes the information attached to a rigid variable. In the case of a variable at kind value, the information we store is the list of permissions available for it, that is, a list of types (line 40). In the case of a variable that corresponds to a type definition (e.g. `list`), we store its definition along with its variance (line 35).

⁴ An optimization would be to bump the level every time a rigid variable is bound; this would allow us to get rid of the `last_binding` field.

```

1   type level = int
2
3   type permissions = typ list
4
5   type env = {
6     ...
7     state: binding PersistentUnionFind.state;
8     flexible: flex_descr IntMap.t;
9
10    floating_permissions: typ list; (* At kind perm *)
11
12    last_binding: binding_type;
13    current_level: level;
14    ...
15  }
16
17  and flex_descr = {
18    structure: structure;
19  }
20
21  and structure =
22    | NotInstantiated of var_descr
23    | Instantiated of typ
24
25  and var_descr = {
26    kind: kind;
27    level: level;
28    fact: Fact.fact option;
29  }
30
31  and binding = var_descr * rigid_descr
32
33  and rigid_descr = DType of type_descr | DTerm of term_descr | DNone
34
35  and type_descr = {
36    definition: type_def;
37    variance: variance list;
38  }
39
40  and term_descr = {
41    permissions: typ list; (* At kind type *)
42  }

```

Figure 9.7: Environment and variable descriptors

Levels and flexible variables

Levels are an implementation technique to make sure that only legal instantiations are performed. In essence, we want to make sure the following prefix is ruled out:

$$\mathcal{F}(a = b)\mathcal{R}(b)\dots$$

The instantiation of a is not well-scoped, since b is introduced after it. Allowing this would be equivalent to validating the (false) formula:

$$\exists a, \forall b, a = b$$

Indeed, flexible and rigid variables can be understood as universal and existential quantifiers that quantify over the current permission.

We thus assign to each variable a *level*, which is a natural integer. The level of a type is thus defined to be the maximum level that occurs inside of it. A flexible variable can only instantiate onto a type which has a level lower or equal than its own.

Here is how levels are assigned (as mentioned in the earlier footnote, optimizations are planned; this section describes the *current* state of things):

$$\underbrace{\mathcal{R} \dots \mathcal{R} \mathcal{F} \dots \mathcal{F}}_{\text{level}=0} \underbrace{\mathcal{R} \dots \mathcal{R} \mathcal{F} \dots \mathcal{F}}_{\text{level}=1} \dots$$

Subsequent variables of the same nature share the same level. The level is bumped whenever introducing a rigid variable after a flexible one. This prevents a flexible variable from instantiating onto a rigid that was introduced *later*.

This allows, however, a flexible variable to instantiate onto another flexible variable that is introduced later at the same level. If one wishes to instantiate a into b , one can conceptually rewrite $\mathcal{F}(a)\mathcal{F}(b)$ into $\mathcal{F}(b)\mathcal{F}(a)$ and pick $\mathcal{F}(b)\mathcal{F}(a = b)$. This corresponds to swapping two subsequent existential quantifiers in the interpretation of the current permission.

The current implementation is based on a *map* that assigns each flexible variable a level. This means that the order of flexible variables at the same level is not enforced: the conceptual swapping between $\mathcal{F}(a)$ and $\mathcal{F}(b)$ transcribes into no operation at run-time. The variables merely happen to be at the same level; the instantiation is thus legal. Conceptually, there always *exists* an ordering of the flexible variables that yields a well-formed prefix (the occurs-check⁵ guarantees that there are no equirecursive instantiations); this order is never computed, however. This has consequences for the implementation of *restrict*, which I present below.

The `current_level` and `last_binding` fields of the `env` type (Figure 9.7) contain enough information to properly bump the level whenever needed. This could be, however, optimized, as I mentioned earlier.

Inserting flexible variables An operation that we wish to support is inserting a flexible variable before another flexible variable. The reason for this will be apparent in Chapter 11; in short, we sometimes discover that a needs to instantiate to, say, (β, γ) with both β and γ fresh flexibles. Adding such a variable does not change the level configuration: the newly-added flexible variable will share the level of the flexible variable next to it.

Since the prefix is actually represented as a *map*, this merely amounts to allocating a fresh, globally-unique `flex_index`, and making sure that the `flexible` field of `env` maps this `flex_index` to a descriptor with the right level⁶.

Restricting the prefix Another operation that we need to implement is prefix restriction (Definition 9.4). We have \mathcal{V}' , a prefix that is more precise than \mathcal{V} (Definition 9.3). \mathcal{V}' has extra binders, either appended at the end (flexible), or inserted at existing levels (flexible). Here is an example, where extra binders are drawn in **red**.

$$\begin{array}{l} \mathcal{V} = \underbrace{\mathcal{R} \dots \mathcal{R} \mathcal{F} \dots \mathcal{F}}_{\text{level}=0} \quad \underbrace{\mathcal{R} \dots \mathcal{R} \mathcal{F} \dots \mathcal{F}}_{\text{level}=1} \quad \dots \\ \mathcal{V}' = \underbrace{\mathcal{R} \dots \mathcal{R} \mathcal{F} \dots \mathcal{F} \dots \mathcal{F}}_{\text{level}=0} \quad \underbrace{\mathcal{R} \dots \mathcal{R} \mathcal{F} \dots \mathcal{F} \mathcal{F}}_{\text{level}=1} \quad \underbrace{\mathcal{R} \dots \mathcal{R} \mathcal{F} \dots \mathcal{F}}_{\text{level}=2} \\ \text{restrict}(\mathcal{V}', \mathcal{V}) = \underbrace{\mathcal{R} \dots \mathcal{R} \mathcal{F} \dots \mathcal{F} \dots \mathcal{F}}_{\text{level}=0} \quad \underbrace{\mathcal{R} \dots \mathcal{R} \mathcal{F} \dots \mathcal{F} \mathcal{F}}_{\text{level}=1} \quad \dots \end{array}$$

According to Definition 9.4, we must exhibit a subset of \mathcal{V}' that contains all the rigid variables from \mathcal{V} . As I mentioned earlier, the current implementation of our prefixes uses a loose data structure which only tracks the level of flexible variables, not their exact order. Rather than do a topological sort of our flexible variables and decide which ones to keep and which ones to drop, the implementation chooses to keep all binders up to the highest level found in \mathcal{V} . This provides a correct implementation of *restrict*.

Subtle issues may arise. Consider the following situation:

$$\mathcal{F}(a)\mathcal{F}(b)\mathcal{R}\mathcal{F}(c)$$

⁵which has yet to be implemented by the present author...

⁶This operation was impossible with the previous representation of a prefix with a list: the `flex_index`'s were indices into the list, and adding a new element at an arbitrary position would mess up the indices.

The level of a and b is 0, the level of c is 1.

$$\mathcal{F}(a)\mathcal{F}(b)\mathcal{R}\mathcal{F}(c = (b, b))$$

We instantiate c onto the tuple type (b, b) . This now means that c is transparently substituted with (b, b) from now on. In particular, if the type-checker tries to unify a with $\text{TyVar } (\text{Vflexible } c)$, the level check will succeed since (b, b) is at level 0. If this is implemented carelessly, then we may pick $\mathcal{F}(a = c)$: this is, after all, one more indirection that eventually ends up onto the same type (b, b) .

$$\mathcal{F}(a = c)\mathcal{F}(b)\mathcal{R}\mathcal{F}(c = (b, b))$$

This causes a problem, however, because this prefix is ill-formed. (A practical consequence is that when we restrict this prefix at level 0, c is dropped, and havoc ensues.)

To solve this, the implementation performs path-compression for this prefix: it eliminates the $a \rightarrow c$ indirection, and makes sure that we manipulate instead the following well-scoped prefix:

$$\mathcal{F}(a = (b, b))\mathcal{F}(b)\mathcal{R}\mathcal{F}(c = (b, b))$$

The value that corresponds to the (b, b) type is, of course, shared.

Another option would be to *modify* the level of c to 0 and obtain:

$$\mathcal{F}(b)\mathcal{F}(c = (b, b))\mathcal{F}(a = c)\mathcal{R}$$

This would correspond to Rémy's style of implementing Hindley-Milner [PR05].

10. A type-checking algorithm

Typing rules were shown in Figure 8.5. As I mentioned earlier (§8.3), the declarative rules give no clue as to how one can implement a type-checking algorithm. Now that we are equipped with a proper concept of a “current permission” $\mathcal{V}.P$, I am ready to describe a set of rules that proceed, just like in the examples, in a forward manner and compute the available permission at each program point.

This flow-sensitive type-checking is unusual for a language in the tradition of ML; classic approaches focus around type inference techniques in the style of the original algorithm W. Due to the fine tracking of effects that Mezzo performs, even though unification-based approaches exist for tracking effects, a flow-sensitive approach seemed natural. In the current state of things, it seems hard to present type-checking in Mezzo as a unification-based procedure.

The algorithm I present is high-level, and elides two key procedures which are described in the subsequent chapters. The algorithm is similar, in essence, to the forward procedure that we describe in the examples (Part II).

In order to type-check e , we take an input, prefixed permission $\mathcal{V}.P$, and return a newer permission Q along with a newer prefix \mathcal{V}' . Unlike the type-checking rules, which return a type, the type-checking *algorithm* returns a permission. A type-checking step thus takes *as an input* a name for the subexpression to be type-checked¹. In the example below, this name is `ret`. Naming the sub-expression presents another advantage: it helps performing a transformation into A-normal form, which facilitates greatly the application of several type-checking rules (§8.3).

Our type-checking judgements are thus of the following form, where we *assume* `ret` to be rigidly bound in \mathcal{V} .

$$\mathcal{V}.P \vdash \text{ret} = e \dashv \mathcal{V}'.Q * \text{ret} @ t$$

A thing to note already is that a type-checking step may introduce new rigid variables. There is no particular relation between \mathcal{V}' and \mathcal{V} . The typing rules use several additional notations which are introduced before the detailed review of Figure 10.1 in §10.3.

In the present chapter, I assume we know how to perform two operations: subtraction and merging.

- Subtraction extracts a permission Q out of a permission P , while computing the parts that are consumed, thus yielding a remainder R . Subtraction assumes that $\mathcal{V}.P$ is fully normalized. Subtraction operates under a prefix, which it may *refine*. We write:

$$\mathcal{V}.(P \ominus Q) = \mathcal{V}'.R$$

Subtraction verifies $\mathcal{V}' \leq \mathcal{V}$, that is, it may introduce new flexible variables and instantiate some others; it does not, however, introduce new rigid variables. Subtraction also computes a subtyping relation; once

¹ The current implementation uses the opposite convention, and a type-checking step *returns* a pair of a fresh name and a prefixed permission. We wish to update the implementation to match the formal presentation.

<p>VARIABLE</p> $\frac{}{\mathcal{V}.P \vdash y = x \dashv \mathcal{V}.P * y = x}$	<p>ANNOT</p> $\frac{\mathcal{V}.P \ominus x @ t = \mathcal{V}'.P'}{\mathcal{V}.P \vdash y = x : t \dashv \mathcal{V}'.P' \otimes x @ t * y = x}$	<p>LET</p> $\frac{\mathcal{V}, \mathcal{R}(x : \text{value}).P \vdash x = e_1 \dashv \mathcal{V}'.P' \quad \mathcal{V}'.P' \vdash y = e_2 \dashv \mathcal{V}''.P''}{\mathcal{V}.P \vdash y = \text{let } x = e_1 \text{ in } e_2 \dashv \mathcal{V}''.P''}$
<p>TYPE-APPLICATION</p> $\frac{\mathcal{V}.P \stackrel{\dagger}{=} \mathcal{V}.P' * x @ \forall(X : \kappa) t}{\mathcal{V}.P \vdash y = x [u] \dashv \mathcal{V}.P' \otimes y @ [u/X]t}$	<p>APPLICATION</p> $\frac{\mathcal{V}, \mathcal{R}(y : \text{value}).P \stackrel{\dagger}{=} \mathcal{V}.P' * \forall(\vec{X} : \vec{\kappa}) f @ t \rightarrow u \quad \mathcal{V}, \mathcal{F}(\vec{X} : \vec{\kappa}).P' \ominus x @ t = \mathcal{V}'.P''}{\mathcal{V}.P \vdash y = f x \dashv \mathcal{V}'.P'' \otimes y @ u}$	
<p>TUPLE</p> $\mathcal{V}.P \vdash y = (\vec{x}) \dashv \mathcal{V}.P * y @ (\vec{=}\vec{x})$	<p>CONSTRUCTOR</p> $\mathcal{V}.P \vdash y = A \{\vec{f} = \vec{x}\} \dashv \mathcal{V}.P * y @ A \{\vec{f} = \vec{x}\} \text{ adopts } \perp$	
<p>FIELD-ACCESS</p> $\frac{\mathcal{V}.P \stackrel{\dagger}{=} \mathcal{V}.P' * x @ A \{\dots f = y \dots\}}{\mathcal{V}.P \vdash z = x.f \dashv \mathcal{V}.P * z = y}$	<p>FIELD-UPDATE</p> <p>A is a mutable constructor</p> $\frac{\mathcal{V}.P \stackrel{\dagger}{=} \mathcal{V}.P' * x @ A \{\dots f = y \dots\}}{\mathcal{V}.P \vdash y = x.f \leftarrow z \dashv \mathcal{V}.P' * x @ A \{\dots f = z \dots\} * y @ ()}$	
<p>TAG-UPDATE</p> <p>A is a mutable constructor B $\{\vec{f}_B\}$ is defined</p> $\frac{\mathcal{V}.P \stackrel{\dagger}{=} \mathcal{V}.P' * x @ A \{\vec{f}_A = \vec{x}\} \quad \#\vec{f}_A = \#\vec{f}_B}{\mathcal{V}.P \vdash \text{tag of } x \leftarrow B \dashv \mathcal{V}.P' * x @ B \{\vec{f}_B = \vec{x}\} * y @ ()}$	<p>MATCH</p> $\frac{\mathcal{V}_0.P \vdash y = \text{let } p_i = x \text{ in } e_i \dashv \mathcal{V}_i.P_i \quad \mathcal{V}_0[y_d] \vdash \mathcal{V}_1.P_1 \vee \dots \vee \mathcal{V}_n.P_n = \mathcal{V}_d.P_d}{\mathcal{V}_0.P \vdash y_d = \text{match } x \text{ with } \vec{p} \rightarrow \vec{e} \dashv \mathcal{V}_d.P_d}$	
<p>ASSERT</p> $\frac{\mathcal{V}.P \ominus Q = \mathcal{V}'.R}{\mathcal{V}.P \vdash y = \text{assert } Q \dashv \mathcal{V}'.R \otimes Q * y @ ()}$	<p>IFTHENELSE</p> $\frac{\mathcal{V}[y_d].P \vdash y_d = \text{match } x \text{ with False} \rightarrow e_2 \mid \text{True} \rightarrow e_1 \dashv \mathcal{V}_d.P_d}{\mathcal{V}.P \vdash y_d = \text{if } x \text{ then } e_1 \text{ else } e_2 \dashv \mathcal{V}_d.P_d}$	
<p>FUNCTION</p> $\frac{\mathcal{V}, \mathcal{R}(\vec{X} : \vec{\kappa}), \mathcal{R}(x : \text{value}).\vec{P} \otimes x @ t_1 \vdash z = e \dashv \mathcal{V}'.Q \quad \mathcal{V}'.Q \ominus z @ t_2 = \mathcal{V}''._}{\mathcal{V}.P \vdash y = \Lambda(\vec{X} : \vec{\kappa}) \lambda(x : t_1) : t_2. e \dashv \text{restrict}(\mathcal{V}'', \mathcal{V}), \mathcal{R}(y : \text{value}).P * y @ \forall(\vec{X} : \vec{\kappa}) t_1 \rightarrow t_2}$		
<p>GIVECONS</p> $\frac{P \stackrel{\dagger}{=} P' * y @ A \{\dots\} \text{ adopts } t * x @ t' \quad \mathcal{V}.t \ominus t' = \mathcal{V}._ \quad t' \text{ and } A \{\dots\} \text{ adopts } t \text{ are exclusive}}{\mathcal{V}.P \vdash z = \text{give } x \text{ to } y \dashv \mathcal{V}.P' * y @ A \{\dots\} \text{ adopts } t' * z @ ()}$	<p>GIVEAPP</p> $\frac{P \stackrel{\dagger}{=} P' * y @ X \vec{t} \quad X \vec{t} \text{ adopts } u \quad \mathcal{V}.P' \ominus x @ u = \mathcal{V}''.P'' \quad u \text{ and } X \vec{t} \text{ are exclusive}}{\mathcal{V}.P \vdash z = \text{give } x \text{ to } y \dashv \mathcal{V}''.P'' * y @ X \vec{t} * z @ ()}$	
<p>TAKE</p> $\frac{P \stackrel{\dagger}{=} P' * x @ \text{dynamic} * y @ t \quad t \text{ adopts } u}{\mathcal{V}.P \vdash \text{take } x \text{ from } y \dashv \mathcal{V}.P' * y @ t \otimes x @ u * z @ ()}$	<p>ADOPTS</p> $\frac{P \stackrel{\dagger}{=} P' * x @ \text{dynamic} * y @ t \quad t \text{ adopts } u}{\mathcal{V}.P \vdash x \text{ adopts } y \dashv \mathcal{V}.P' * y @ t * z @ \text{bool}}$	

Figure 10.1: The high-level type-checking algorithm

flexible variables have been substituted (\times appeared in Definition 9.7), the output of subtraction verifies the subsumption relation introduced earlier (§8.2).

$$\mathcal{V} \times P \leq \mathcal{V}' \times (Q * R)$$

Moreover, $\mathcal{V}'.R$ is fully normalized (Definition 9.10). Subtraction is the topic of Chapter 11.

An additional judgement which operates at kind type is defined:

$$\mathcal{V}.(P * t \ominus t') \triangleq \mathcal{V}, \mathcal{R}(z : \text{value}).(P * z @ t \ominus z @ t')$$

- Merging computes the disjunction of two normalized, prefixed permissions $\mathcal{V}_l.P_l$ (“left”) and $\mathcal{V}_r.P_r$ (“right”). It returns a prefixed, fully normalized permission that subsumes the two, written $\mathcal{V}_d.P_d$ (“destination”). Moreover, the merge operation needs to know the set of “old” variables that were bound before the disjunction, which I write \mathcal{V}_0 . The merge operation also distinguishes the return value associated to the disjunction. We assume the variable to be rigidly bound in \mathcal{V}_0 (that is, *before* the disjunction). It is passed as the *input* of the type-checking algorithm for the left and right expressions. I write:

$$\mathcal{V}_0[\text{ret}] \vdash \mathcal{V}_l.P_l \vee \mathcal{V}_r.P_r = \mathcal{V}_d.P_d$$

The merge operation satisfies the following properties:

- $\mathcal{V}_d \leq \mathcal{V}_0$, that is, it refines the original prefix: intuitively, the merge operation may make some instantiation decisions to provide better results;
- $\mathcal{V}_l \times P_l \leq \mathcal{V}_d \times P_d$, that is, the destination permission is a supertype of the left permission;
- $\mathcal{V}_r \times P_r \leq \mathcal{V}_d \times P_d$, that is, the destination permission also is a supertype of the right permission, that is, it subsumes both.

This presentation evades the issue of instantiation choices local to \mathcal{V}_l and \mathcal{V}_r ; this is discussed in Chapter 12, which is dedicated to the merge problem.

10.1 Typing rules vs. algorithm

There are several key differences between the specification of the algorithm (Figure 10.1), which is syntax-directed, and the declarative type-checking rules (Figure 8.5).

For starters, the form of the rules is not the same. In the algorithmic specification, rules take an input permission and return an output permission, along with a permission $\text{ret } @ t$ for the expression that was just type-checked.

Another difference lies in the treatment of binders. The implementation uses flexible variables, which naturally do not appear in the declarative rules; they are, after all, an implementation technique. Flexible variables may be added or instantiated when stepping through the program. The algorithmic specification takes care of this.

The type-checking rules also leave things unspecified: for instance, when reading from a field (READ), the only requirement is that the type of the field be duplicable. By using a normalized representation, the algorithmic specification ensures that in such a situation, a singleton type is always used.

Finally, and perhaps more importantly, the type-checking rules allow one to use FRAME and SUB at any time; the algorithmic specification applies permission subsumption at key, controlled steps: when normalizing, subtracting, or merging.

Some choices, however, are still left up to the actual implementation. Section §10.7 reviews the parts of the specification that are non-deterministic.

10.2 Transformation into A-normal form

The algorithmic specification often writes x or y (that is, program variables) where the syntax of expressions (Figure 7.7) allows for arbitrary expressions. This allows for a more concise presentation of the algorithm, since one can easily refer to the permission $x @ t$ rather than inlining the LET rule.

For instance, FIELD-ACCESS can only type-check $x.f$. This is not restrictive: if one wishes to type-check $e.f$, one can always type-check e first, thus obtaining a fresh name ret for e , then type-check $\text{ret}.f$.

This transformation into A-normal form, is conceptually, a pure AST-to-AST transformation. It just so happens that this is performed on-the-fly when type-checking Mezzo programs. The details of this transformation are omitted from the rules. One can understand this transformation as an extra set of rules, as I mentioned earlier (§8.3).

The only important point is that the transformation into A-normal form is performed according to the operational semantics of Mezzo. That is, the introduction of let-bindings materializes a sequence, that corresponds to the order of execution. Since types track effects, it is mandatory in order to ensure soundness that the sequencing be done consistently with the operational semantics. The discussion of individual rules re-emphasizes this point on the concrete case of TUPLE and CONSTRUCTOR.

10.3 Helper operations and notations

Before jumping into a description of the rules, I introduce a few helper notations as well as some extra procedures that the algorithm leverages.

Discarding the result We may want to assert that a subtraction succeeds, and bind the resulting prefix, while discarding the remainder. For that purpose, we write:

$$\mathcal{V}.P \ominus Q = \mathcal{V}'._$$

This amounts to asserting that $P \leq Q$. However, since the algorithm operates with a set of flexible variables, we need to keep the output prefix \mathcal{V}' since it may contain instantiation choices that make $P \leq Q$ possible.

Similarly, whenever we want to algorithmically check that $t \leq t'$, we can write:

$$\mathcal{V}.t \ominus t' = \mathcal{V}'._$$

Duplicable subset We sometimes need to keep only the duplicable parts of a permission. If $P \overset{\exists, *}{\rightsquigarrow} p_1 * \dots * p_n$, this merely consists in dropping all permissions p_i that do not satisfy the “is duplicable” predicate. We write \bar{P} .

Focusing We often need to assert that a prefixed permission $\mathcal{V}.P$ contains, among other things, a permission of a certain form. Roughly, we need to perform powerful pattern-matching on a permission P to find a permission of a certain form.

For instance, when type-checking the function call $f x$, we want to assert that P contains an arrow permission for f . We write:

$$\mathcal{V}.P \stackrel{!}{=} \mathcal{V}.P' * f@t \rightarrow u$$

The extra $!$ in a superscript denotes that a syntactic matching takes place to find a function type for f .

More generally, if we want to assert that $\mathcal{V}.P$ contains a certain permission \mathbf{p} , we write:

$$\mathcal{V}.P = \mathcal{V}.P' * \mathbf{p}$$

\mathbf{p} should be understood to be a “pattern”, that is, the desired form of the permission that we wish to find. The free variables in \mathbf{p} are taken to be *binders*, that is, in the example above, t and u may be referred to by other premises.

This “focusing” operation performs an implicit duplication step whenever possible: if \mathbf{p} is duplicable, then $P' = P$.

Having this convention simplifies the premises of many rules as we no longer need to state that, should the focused permission be duplicable, it is also returned in the output permission. This operation, however, remains non-deterministic; this is discussed extensively in §10.7.

10.4 Addition

Leveraging the normalization procedure, one can implement the *addition* of a permission Q to an already-normalized, prefixed permission $\mathcal{V}.P$. Additions take place when entering function bodies and when returning from function

calls.

$$\text{ADDITION} \frac{P * Q \overset{\exists, *}{\rightsquigarrow} \exists(\vec{X} : \vec{\kappa}) R}{\mathcal{V}.P \otimes Q = \mathcal{V}, \mathcal{R}(\vec{X} : \vec{\kappa}).R}$$

Addition consists in \exists -normalizing the new conjunction, and opening any existential binders as rigid variables.

The rule above admits a special-case for equations, which I mentioned in §9.5.


Lemma 10.1. *Addition preserves full normalization.*

10.5 Type-checking algorithm

The algorithm is presented in Figure 10.1. The details of the transformation into A-normal form are left out. Also, I omit some obvious well-kindedness premises, such as in TYPE-APPLICATION.

Lemma 10.2. *If $\mathcal{V}.P \vdash e \dashv \mathcal{V}'.P'$ and $\mathcal{V}.P$ is fully normalized, then $\mathcal{V}'.P'$ is fully normalized as well, i.e. the type-checking algorithm maintains full normalization.*

Proof. Recall that subtraction and merging return fully normalized permissions, and that \oplus and \otimes take care of renormalizing. We essentially need to renormalize whenever dealing with user-provided types (assertions, annotations on expressions or function definitions, “adopts” clauses). These types are written using the surface syntax and are not in the decomposed form.

VARIABLE adds an equation to an already-normalized permission; this offers no opportunities for further renormalization, meaning that the output permission is fully normalized (the same goes for LET, FIELD-ACCESS). ANNOT performs renormalization using \otimes (the same goes for TYPE-APPLICATION, APPLICATION). Rules TUPLE, CONSTRUCTOR add already-decomposed permissions that refer to a fresh variable, hence preserving full normalization. FIELD-UPDATE and TAG-UPDATE add the unit permission, which is already decomposed, to a fresh variable. They also preserve the decomposed property for their x variable. MATCH relies on the post-condition of the merge operation to ensure full normalization (the same goes for IFTHENELSE). ASSERT, just like ANNOT, renormalizes. FUNCTION adds a permission for a fresh variable, that cannot undergo \exists -normalization. GIVECONS and GIVEAPP preserve normalization for reasons similar to FIELD-UPDATE. TAKE renormalizes. ADOPTS adds the bool type to a fresh variable, meaning no further normalization can occur. 

Let us now review the rules.

VARIABLE embodies two key points. First, the program variable x , which formerly belonged to expressions, now also appears in types. Second, it does not consume any permission from the environment: the rule merely states that the variable x “is x ”. This allows us to decide later on, when trying to match $\mathcal{V}.P$ against an annotation (e.g. when returning from a function body), which $x @ t$ should be consumed from the environment.

ANNOT checks that the given expression satisfies an annotation. The rule, while seemingly trivial, is interesting. It relies on a subtraction to assert that $x @ t$ can be obtained from $\mathcal{V}.P$. This subtraction may need to perform flexible variable instantiations so as to succeed: the rule thus returns a permission prefixed with a new \mathcal{V}' . Another interesting point is that the type annotation t requires re-normalizing (via \otimes), since it is user-provided and certainly can be expanded. Finally, this operation may weaken $\mathcal{V}.P$; if P is $x @ \text{MNil}$, and if t is m!ist int , then the subtraction will return an empty remainder, meaning that the output permission for the rule is $x @ \text{m!ist int}$.

ASSERT is similar, except that it works on a permission, and returns the unit type.



Assertions are useful in conjunction with `let flex` constructs, to name existential type variables which, due to \exists -normalization, are “auto-unpacked”. We saw an example of this pattern in Figure 3.11. Here is an artificial, smaller example.

```
val f (consumes x: { t } t): () =
  let flex u in
    assert x @ u
```

Upon entering this function body, the user has no way to refer to the type t . Binding a flexible variable, however, allows them to assert that $x @ u$ holds, which results in the type-checking performing the desired instantiation so as to obtain:

$$\mathcal{R}(x : \text{value})\mathcal{R}(t)\mathcal{F}(u = t)$$

There are multiple choices for u here: u could be unknown or even $=x$. The type-checker, however, guarantees that, if possible, neither of these two are selected for the instantiation choice. This is, of course, rather unsatisfactory from a language-design point of view, and this is an area that we wish to improve in the future.

LET type-checks e_1 , and assigns the resulting type to the x name. In the case where the left-hand side of the binding is a more complex pattern, equations are added in the environment, so as to bind each name in the pattern to the corresponding internal name. We omit the details for this procedure: they are very similar to the rules from Figure 8.6. Finally, we type-check e_2 , and assign the resulting type to the initial name.



This LET rule is unusual, in the sense that x is not removed from the list of binders after type-checking the whole expression. That is, x does belong in \mathcal{V}'' .

The reason for this is that the permission $y @ t$ that results from type-checking the let-binding may mention x . That is, x may appear in t , meaning that it may actually be the case that x does escape! This is handled transparently: the programmer can no longer refer to x , since kind-checking disallows referring to x outside of its scope. After escaping, x remains in \mathcal{V} as a rigidly-bound, unreachable variable, at kind value.

APPLICATION finds a function type for f , performs a subtraction to ensure the argument has the right type, and performs an addition in the conclusion. There may be several suitable function types: this is discussed in §10.7. An additional difficulty stems from the fact that the function may be polymorphic, i.e. that there may be universal quantifiers. We introduce them as flexible variables: we need to guess these types so as to make the function call succeed. The difficulty, naturally, lies within the machinery of the subtraction, where flexible variables may be instantiated, subsumption rules applied to synthesize $x @ t'$, and framing performed on-the-fly to leave the rest of the current permission untouched.

TYPE-APPLICATION instantiates a (among possibly many) universally-quantified type. This is mostly used for polymorphic functions. One may also obtain polymorphic values; here is an example.

```
val x =
  let x = MNil in
  assert x @ [a] mlist a;
  x
```

The premise of the rule contains an implicit focusing operation, as it is non-obvious whether $\forall(X : \kappa) t$ is duplicable or not.

TUPLE and CONSTRUCTOR are very concise, since we assume an A-normal form. Let me stress again that in practice, tuples and constructors are type-checked left-to-right, consistently with the evaluation order of expressions in Mezzo. (The adopts \perp conclusion in CONSTRUCTOR is useless in the case that A is immutable, since the rules for adoption and abandon demand that the constructor A be exclusive.)

FIELD-ACCESS and FIELD-UPDATE leverage the normalized representation that we adopted earlier, and assume singleton types for the constructor's fields. This not only satisfies the premise of the actual type-checking rule, which requires that only a duplicable field may be read, but also preserves the \exists -normalized, expanded representation. In a nutshell, these two rules amount to adding and changing equations, respectively.

TAG-UPDATE describes precisely what happens when changing the tag of a constructor. The new constructor needs to have the exact same number of fields. The resulting type has the fields \vec{f}_B of B, but keeps the types \vec{t} that were there already. (The output type for x is decomposed, and there is no other constructor for x (because A is exclusive), so no further normalizations can apply: we don't need to renormalize in the conclusion.)

MATCH type-checks each branch as if it were a let-binding. The results are then merged into a single resulting environment. The way we perform this operation is the topic of Chapter 12.

IFTHENELSE piggybacks on the MATCH rule. This does not account for the dubious "polytypic" extension I mentioned earlier, where if-then-else expressions may work for any data type with two branches (§8.3).

FUNCTION is a little more involved: when type-checking the function body, we need to restrict ourselves to the duplicable parts of P (which we write \bar{P}), for function types are always duplicable in Mezzo. We also need to

introduce universally-quantified type parameters as *rigid* variables, and extend the permission with $x @ t_1$, that is, the permission that stands for the function’s argument. Once the body has been successfully type-checked, we perform a subtraction to ensure that the function has the correct return type.

There is no separate rule for Λ -abstractions and λ -abstractions: the `FUNCTION` rule combines the two together. This is a reasonable thing to do: Λ -abstractions are not exposed to the user, and the only place where they appear is when desugaring a function definition. This means that the `FUNCTION` rule can safely assume that whenever there is a Λ , a λ -abstraction is to be found under them. This makes the type-checking specification easier, since we leverage the type annotations on the λ -abstraction to determine what is the type we should return.

`GIVECONS` and `GIVEAPP` implement the `give` operation. In the case where the adopter sports a concrete type, we need to apply the covariance rule of the `adopts` clause to make sure that the type of x and the `adopts` clause of y match. The “is exclusive” check is also deferred to the last minute. The reason for this is to allow definitions such as:

```
data mutable container a =
  | Container { ... } adopts a
```

In that case, the type-checker cannot check at definition-time that `a` is exclusive; when performing `give` and `take` operations, though, we can check that the parameter `a` has been chosen to be exclusive. This pattern is extremely useful in practice.

We also assert that $t \leq t'$: this is important as flexible instantiations may be required for the rule to be applied successfully. Recall that the `CONSTRUCTOR` rule states that freshly-allocated blocks have an `adopts` clause initially set to \perp , that is, to $\forall a. a$. Performing a subtraction will properly instantiate \perp to whatever is suitable.

In case the adopter is the application of a nominal type $X \vec{t}$, we merely state that $X \vec{t}$ has an `adopts` u clause, and try to obtain the said u for x . In both cases, the return type is the unit type.

`TAKE` implements the `take` operation. By “ t adopts u ”, we mean that t is either a type application whose `adopts` clause turns out to be “adopts u ”, or that t is a constructor type with an “adopts u ” clause. Renormalization is required in the conclusion since u may be anything.

`ADOPTS` is similar.

Conjecture 10.3 (Correctness of the algorithm). *The algorithm implements faithfully the declarative rules for type-checking presented in Chapter 8. That is, if $\mathcal{V}.P \vdash y = e \dashv \mathcal{V}'.P'$, and if K is all the variables from \mathcal{V}' with their respective kinds, then:*

$$\mathcal{V}' \times (K, P \vdash y : (=y \mid P'))$$

10.6 About type inference

The type-checking algorithm introduces flexible variables in several situations; when type-checking function calls (`APPLICATION`), when type-checking a `let flex` construct (rule not shown); internal procedures (subtraction, merging) may also introduce flexible variables in the course of their action. This amounts to performing type inference, since the subtraction algorithm will try to make suitable instantiation choices for the flexible variables, so as to make the desired operation succeed.

The most salient use-case for flexible variable introduction is the function call: the type-checker introduces flexible variables and relies on the subtraction operation to come up with suitable instantiations to make the function call succeed. One advantage is that the user can write “`let x = length l in ...`” instead of “`let x = length [int] l in ...`”, which is, naturally, a huge gain in usability.



This is not quite the classic local inference technique. In the local inference style, flexible variables are discarded as soon as they have been instantiated suitably. Conversely, in the algorithm presented above, flexible variables may be kept for an indefinite amount of time. Consider the new function from the `lock` module, whose signature is:

```
val new: [p: perm] (| consumes p) -> lock p
```

Upon calling this function, a new flexible variable is introduced. Lacking any type annotation or `assert` statement, and assuming that the resulting `lock` is unused until the end of its scope, the flexible variable that corresponds to `p` will be kept uninstantiated until the end of the current function body or, if defining a top-level value, until the end of the current compilation unit.

The subtraction procedure is not complete, though: a drawback is that in some rare situations, inference will fail. For these rare cases, the user still has the option to resort to a manual type application (`TYPE-APPLICATION`), meaning that user provides the desired instantiation. The `APPLICATION` rule then no longer needs to instantiate this Λ -abstraction. Mezzo provides syntactic sugar for instantiating a Λ -abstraction under another Λ using its name, should the user need to instantiate only one of the type variables.

10.7 Non-determinism in the type-checker

The type-checking algorithm, while providing a more concrete specification suitable for implementation, remains however non-deterministic in several aspects.

- Focusing is a non-deterministic operation, as several matching types may be found. It sometimes happens, for instance, that several function types are available for the same variable f (`APPLICATION`) [GPP13]. In this case, the type-checker tries all possible function types; if exactly one succeeds, it commits to this one. If more than one succeed, it emits a warning and picks an arbitrary one.
- `GIVECONS` also relies on a certain t' that the algorithm is expected to guess. Our type-checker implementation works as follows. If the adopts clause for y has never been specified (i.e. is still \perp), the type-checker demands a type annotation to guess t' . Otherwise, if y already has an adopts clause t , it picks $t = t'$. Abandon operations are delicate and we do not wish for the user to rely on inference or implicit subtyping for these.
- `TYPE-APPLICATION` is also non-deterministic, as it focuses *one* universally-quantified permission, among possibly many. The type-checker picks an arbitrary permission in this case. This could be improved by performing the substitution on *all* matching types.
- For constructor operations (`FIELD-ACCESS`, `FIELD-UPDATE`, `TAG-UPDATE`), there may be several matching types. Having multiple constructor types for the same variable, however, means that the environment is inconsistent, as our normalization procedure simplifies consistent situations. We thus pick an arbitrary constructor permission for these rules.
- The subtraction operation itself is non-deterministic. An easy way to see this is to consider the simple subtraction problem below:

$$\mathcal{F}(a : \text{type}).(x @ \text{int} - x @ a)$$

This subtraction problem can be solved by picking $a = \text{int}$, but also $a = (=x)$, as the singleton type $=x$ is always available for x . Subtraction, internally, performs exploration; for type-checking, however, the algorithm does not keep track of all possible solutions returned by the subtraction algorithm and picks the “best” one (accordingly to the subtraction procedure).

- Finally, the merge procedure that `MATCH` relies on is also non-deterministic and may find several incomparable solutions. We detail these difficulties in Chapter 12.

The most important point to note, perhaps, is that while the subtraction procedure performs exploration, the type-checker itself does not backtrack. Consider for example a trivial call to the identity function:

```
val id [a] (x: a): a = x

val _ =
  let y = 0 in
  id y
```

The type-checker, when type-checking the call to `id`, will introduce a flexible variable for a , and subtract $y @ a$. The subtraction algorithm will return three possible values for a : `int`, `=y` and `unknown`. The type-checker of Mezzo, however, will only keep the best solution, that is, the first (`int`) and proceed, rather than try to type-check the rest of the definition with each possible value for a .

The main explanations for this behavior is that we want to limit computational costs. Both subtraction and merging are expensive operations, which perform exploration and backtracking already. If we were to keep track of

multiple solutions through each program point, large Mezzo programs would be harder to type-check efficiently, and would bring us even further from what we claim is still a type system. Moreover, errors would be even harder to explain to the user, as we would need to mention the exponential number of solutions that we explored.

10.8 Propagating the expected type

The Mezzo type-checker features a basic mechanism for propagating the expected type of an expression. The propagation starts at the function declaration level: the expected return type is propagated down through tuples, constructors, if-then-else's, matches and let-bindings. This means that only expressions in terminal position of function bodies receive the expected type. Type annotations on expressions are also propagated downwards. There is no propagation backwards, as in bidirectional type-checking; this is something that we wish to explore.

There are two main places in the Mezzo type-checker that leverage that expected type. When performing a polymorphic function call, after introducing flexible variables \vec{X} , the type-checker knows the return type of the function t_2 ; if an expected type t_e is also available, a subtraction $t_2 - t_e$ is performed. Oftentimes, this allows the type-checker to make better instantiation choices for the variables \vec{X} ; if these variables appear in the domain of the function as well, this augments chances that the function call will succeed.

The other situation where the type-checker leverages type annotations is when confronted with a merge operation. Having an expected type makes sure the algorithm provides *at least* whatever the user asked for: the expected type is first subtracted from each branch, and the remainders are merged. Concretely, if the expected type is t_e , and the type-checker is confronted with a disjunction $\mathcal{V}_l.P_l \vee \mathcal{V}_r.P_r$, then the type-checker first performs $\mathcal{V}_l.P_l - \text{ret } @ t_e = \mathcal{V}'_l.P'_l$, then $\mathcal{V}_r.P_r - \text{ret } @ t_e = \mathcal{V}'_r.P'_r$, and finally performs $\mathcal{V}_0, \mathcal{R}(\text{ret} : \text{value}) \vdash \mathcal{V}'_l.P'_l \vee \mathcal{V}'_r.P'_r = \mathcal{V}_d.P_d$. The net result is $\mathcal{V}_d.P_d \otimes \text{ret } @ t_e$. This behavior is consistent with that of assertions and type annotations, which specify only a *partial* type annotation. One cannot conceivably ask the user to annotate the entire universe of permissions that they expect to possess.

10.9 A glance at the implementation

The Permissions module (in `typing/Permissions.mli`) exports the following functions:

```

1 | type result = ((env * derivation), derivation) either
2 |
3 | val add_perm: env -> typ -> env
4 | val sub_perm: env -> typ -> result

```

The `add_perm` function implements the (always) renormalizing addition we saw earlier (§10.4). The `sub_perm` function corresponds to the subtraction operation, which we saw but haven't explained yet. A subtraction returns a `result`: it is either a success (Left case), meaning that the caller gets an environment along with a successful type-checking derivation, or a failure (Right case), meaning that the caller gets a failed derivation.

Derivations are, at the moment, used only for debugging purposes, but one could imagine a trusted (machine-checked) verifier that takes a derivation and validates its correctness with regard to the subsumption rules of the Mezzo proof of soundness.

The Merge module (in `typing/Merge.mli`) exports the following function:

```

1 | val merge_envs: env -> ?annot:typ -> env * var -> env * var -> env * var

```

The first `env` is the original one. Then follow two pairs of an environment and a variable: these are P_l with ret_l and P_r with ret_r , in MATCH. The function returns P_d with ret_d . In the case of a n -ary ($n > 2$) match expression, sub-environments for each branch are merged pairwise using a `left_fold`.

The type-checker is located in `typing/TypeChecker.ml` and leverages these modules to build the algorithm we described in this chapter. Implementation topics that were not covered in this chapter are: performing substitutions and opening binders, reporting proper error messages, proper let-bindings with patterns.

The type-checker module offers the following interface.

```

1 | val check_declaration_group :
2 |   env ->

```

```
3 | ExpressionsCore.definitions ->  
4 | ExpressionsCore.toplevel_item list ->  
5 | env * ExpressionsCore.toplevel_item list * (Variable.name * var) list
```

The function takes a group of mutually recursive definitions along with an environment; it also takes the following toplevel items (type or value definitions). It type-checks the definitions, adds the relevant permissions in the environment, and properly opens the corresponding binders in the following toplevel items.

11.1. The subtraction operation

We now turn to *subtraction*, one of the two basic building blocks of our type-checking algorithm, which I have mentioned at numerous occasions already.

11.1.1 Overview of subtraction

A subtraction takes the following form, and assumes that $\mathcal{V}.P$ is fully normalized.

$$\mathcal{V}.(P \ominus Q) = \mathcal{V}'.R$$

The output, prefixed permission $\mathcal{V}'.R$ satisfies the following properties.

- $\mathcal{V}' \leq \mathcal{V}$, that is, the output prefix is more precise (Definition 9.3). One can understand this subtraction problem as a question that we are asking: given the parameters of the input problem (the rigid variables in \mathcal{V}), can you find a suitable instantiation of the flexible variables which makes this subtraction problem succeed? Should the algorithm succeed, \mathcal{V}' contains instantiated flexible variables, along with possibly more flexible variables: it is thus more precise.
- $\mathcal{V}' \times P \leq \mathcal{V}' \times (Q * R)$, that is, subtraction applies subsumption rules so as to separate P into the desired part Q and a remainder R . Naturally, we want R to be as “big” as possible (that is, to contain as many permissions as possible or, in terms of \leq , to be as small as possible). One thing to note already is that writing $\mathcal{V}' \times P$ and $\mathcal{V}' \times Q$ makes sense: since $\mathcal{V}' \leq \mathcal{V}$, all variables from \mathcal{V} are also bound in \mathcal{V}' .

The type-checker calls into \ominus : this is the operation that is exposed by the `Permissions` module in the `Mezzo` source code.

This \ominus operation is called the *renormalizing subtraction*, because it is defined in terms of a more primitive “ $-$ ” operation, which will be called “plain subtraction”, or just “subtraction” from now on.

Plain subtraction takes the following form.

$$\mathcal{V}.(P - Q) = \mathcal{V}'.R$$

The renormalizing subtraction is defined in terms of plain subtraction as follows.

$$\frac{\text{RENORMALIZING-SUBTRACTION} \quad Q \overset{\forall, *}{\rightsquigarrow} \forall(\vec{X} : \vec{\kappa}).Q' \quad \mathcal{V}, \mathcal{R}(\vec{X} : \vec{\kappa}).P - Q' = \mathcal{V}'.R}{\mathcal{V}.P \ominus Q = \text{restrict}(\mathcal{V}', \mathcal{V}).R}$$

(I explain later on why universal quantifiers on the right-hand side of a subtraction are moved into the prefix as rigid bindings.) Renormalizing subtraction \forall -normalizes Q . As a consequence, new rigid binders are added into

the prefix. It then calls into plain subtraction, and discards the extra rigid binders from the result using `restrict`, hence ensuring that the $\mathcal{V}' \leq \mathcal{V}$ post-condition is met.

The reason why \ominus re-normalizes Q is that it is one of the pre-conditions of plain subtraction. Indeed, the pre- and post-conditions of plain subtraction are as follows:

- it requires that $\mathcal{V}.P$ be fully normalized,
- it requires that Q be \forall -normalized,
- it ensures that $\mathcal{V}'.R$ is fully normalized,
- it ensures that $\mathcal{V}' \times P \leq \mathcal{V}' \times (Q * R)$.

In essence, the pre- and post- conditions of plain subtraction are the same as full normalization, except that plain subtraction expects Q to be \forall -normalized. The correctness result for \ominus thus depends entirely on the correctness of “ $-$ ”, which is the topic of the subsequent sections.

Subtraction performs several tasks in order to compute \mathcal{V}' and R .

- It applies subsumption rules on-the-fly to match the desired goal Q . For instance, one may possess $x @ (=y, =z) * y @ \text{int} * z @ \text{int}$, and one may want to call $f x$ where f demands an argument of type (int, int) . Subsumption rules must be applied to transform the permission into the desired form.
- Subtraction also determines which part of the current permission is consumed by a function call and which part is left untouched (this is the R part). This a problem known as *frame inference* in separation logic [BCO05b]. Several algorithms have been proposed [BCO05a, DP08, NDQC07]. Calcagno *et al.* [CDOY09], when referring to this problem, formulate it as $\delta \vdash H * ?\text{frame}$. The problem, within the context of Mezzo, presents interesting difficulties compared to frame inference (ownership issues, comparison of function types, arbitrary universal and existential quantifiers, arbitrary user-defined inductive predicates, a rich set of subsumption rules, multiple choices for flexible variable choices ...).
- Subtraction performs inference of *flexible* variables, and instantiates them to find a solution to the desired problem. As we mentioned earlier, there may be several instantiation choices, and subtraction backtracks and explores multiple branches of the solution space.

These operations are hard to un-tangle. Inference, for instance, will result in a variable being instantiated into a duplicable or non-duplicable type, which will, in turn, change which parts of P are consumed.

As we saw in Chapter 10, the type-checking algorithm for Mezzo leverages subtraction in two places. The operation is used when exiting function bodies (`FUNCTION`), to check that the return type is satisfied (the remainder is then discarded), and when calling functions (`APPLICATION`), to compute the part of the current permission that is *consumed* by the function call, and the remainder (the frame). Less importantly, subtraction is also used when checking type annotations and permission assertions.

In this chapter, I explain subtraction using a couple examples. I then formally define subtraction and introduce a set of algorithmic rules suitable for implementing an algorithm. I then justify why subtraction is correct with regard to the subsumption rules of Mezzo.

11.2 Subtraction examples

I take the example of the regular `map` function: it is self-contained, and contains interesting sample subtractions (Figure 11.1).

Several subtractions take place in the `map` example. The first one corresponds to the type-checking algorithm trying to prove that the argument of a function call has the desired type (`APPLICATION`). This situation happens at line 11. The second example corresponds to the type-checker trying to prove that the function body satisfies the function’s post-condition (`FUNCTION`). This situation happens twice, at line 13 and line 9.

Function call

At line line 11, a call to `f` happens. The following variables are in scope: a, b , at kind type; these have been opened as rigid binders by the `FUNCTION` rule; the same goes for f and l , at kind value. New variables were bound in the `Cons` branch: h and t are let-bound, rigid variables at kind value. The prefix for line 11 is thus:

$$\mathcal{R}(a, b) \mathcal{R}(f, l, h, t : \text{value})$$

```

1  open list
2
3  val rec map [a, b] (
4    f: (consumes a -> b),
5    consumes l: list a
6  ): list b =
7  match l with
8  | Nil ->
9    l
10 | Cons { head = h; tail = t } ->
11   let h' = f h
12   and t' = map (f, t) in
13   Cons { head = h'; tail = t' }
14 end

```

Figure 11.1: The classic map function

The current permission is made up of a permission for f , along with a refined, expanded (via renormalizing addition) permission for l :

$$f @ a \rightarrow b * l @ \text{Cons} \{ \text{head} = h; \text{tail} = t \} * h @ a * t @ \text{list } a$$

Per the type-checking algorithm (Figure 10.1), we need to find a permission for f that is a function type; we happen to possess one, namely $f @ a \rightarrow b$. The function is called with argument h . We thus need (per rule APPLICATION) to obtain $h @ a$ from the current permission; in other words, we need to perform the following subtraction:

$$\begin{array}{l}
\mathcal{R}(a, b) \mathcal{R}(f, l, h, t : \text{value}). \\
f @ a \rightarrow b \\
* l @ \text{Cons} \{ \text{head} = h; \text{tail} = t \} \quad - \quad h @ a \\
* h @ a \\
* t @ \text{list } a
\end{array}$$

Let us sketch an algorithm for performing this subtraction. The goal is $h @ a$: we can syntactically match h in the goal with the corresponding permission for h in the hypothesis, that is, in the current permission. Before “taking $h @ a$ ”, we need to determine whether we can save a copy of the permission or not. In this case, the type a is abstract, meaning that lacking any mode hypothesis, it is affine (when instantiated, a could be anything!). The permission $h @ a$ is thus affine too: the subtraction *consumes* the permission and returns the following output. This final prefixed permission corresponds to $\mathcal{V}.R$ (the output prefix along with the remainder) in our earlier description of subtraction (§11.1).

$$\begin{array}{l}
\mathcal{R}(a, b) \mathcal{R}(f, l, h, t : \text{value}) \\
f @ a \rightarrow b \\
* l @ \text{Cons} \{ \text{head} = h; \text{tail} = t \} \\
* t @ \text{list } a
\end{array}$$

Function body

A more sophisticated subtraction takes place when type-checking the whole match expression. Because of the downward propagation of the expected type I mentioned earlier (§10.8), the type-checker needs to check that both in the Nil case (line 9) and the Cons case (line 13), the post-condition of the function is met.

In the Cons branch, this gives:

$$\begin{array}{l} \mathcal{R}(a, b) \mathcal{R}(f, l, h, t, h', t' : \text{value}). \\ f @ a \rightarrow b \\ * l @ \text{Cons} \{ \text{head} = h; \text{tail} = t \} \\ * h' @ b \\ * t' @ \text{list } b \\ * \text{ret} @ \text{Cons} \{ \text{head} = h'; \text{tail} = t' \} \end{array} \quad - \quad \text{ret} @ \text{list } b$$

The subtraction algorithm performs a syntactic matching, again, on `ret` in the goal. This time, however, the permission in the hypothesis does not syntactically match the permission in the goal. Solving this, however, is easy: `Cons` is a constructor of list, meaning that we can *strengthen* the goal and prove instead:

$$\begin{array}{l} \mathcal{R}(a, b) \mathcal{R}(f, l, h, t, h', t' : \text{value}). \\ f @ a \rightarrow b \\ * l @ \text{Cons} \{ \text{head} = h; \text{tail} = t \} \\ * h' @ b \\ * t' @ \text{list } b \\ * \text{ret} @ \text{Cons} \{ \text{head} = h'; \text{tail} = t' \} \end{array} \quad - \quad \text{ret} @ \text{Cons} \{ \text{head} : b; \text{tail} : \text{list } b \}$$

This goal can be further decomposed in three subgoals: showing that the constructors match, and showing that the head and tail fields possess the right types. The type-checker realizes that `ret`, in both the goal and the hypothesis, has the `Cons` tag. The `Cons` tag being duplicable, the hypothesis for `ret` remains, as well as the other two remaining subgoals.

$$\begin{array}{l} \mathcal{R}(a, b) \mathcal{R}(f, l, h, t, h', t' : \text{value}). \\ f @ a \rightarrow b \\ * l @ \text{Cons} \{ \text{head} = h; \text{tail} = t \} \\ * h' @ b \\ * t' @ \text{list } b \\ * \text{ret} @ \text{Cons} \{ \text{head} = h'; \text{tail} = t' \} \end{array} \quad - \quad \begin{array}{l} h' @ b \\ * t' @ \text{list } b \end{array}$$

The type-checker then tries to move forward by proving one of the sub-goals. The type-checker performs syntactic matching, again, and successfully proves the two remaining sub-goals using the hypotheses `h' @ b` and `t' @ list b`. These two permissions, however, are not duplicable, meaning that they disappear.

$$\begin{array}{l} \mathcal{R}(a, b) \mathcal{R}(f, l, h, t, h', t' : \text{value}). \\ f @ a \rightarrow b \\ * l @ \text{Cons} \{ \text{head} = h; \text{tail} = t \} \\ * \text{ret} @ \text{Cons} \{ \text{head} = h'; \text{tail} = t' \} \end{array} \quad - \quad \text{empty}$$

The subtraction is now finished, and the remainder `R` appears on the left-hand side. The prefix `ℳ'` also appears at the top of the subtraction.

In the `Nil` case, the subtraction is similar, albeit somewhat simpler:

$$\begin{array}{l} \mathcal{R}(a, b) \mathcal{R}(f, l : \text{value}). \\ f @ a \rightarrow b \\ * l @ \text{Nil} \\ * \text{ret} = l \end{array} \quad - \quad \text{ret} @ \text{list } b$$

The algorithm also strengthens the goal and transparently rewrites the hypothesis using the equation `ret = l` so as to succeed.

11.3 The subtraction operation

A proof search procedure

Subtraction is very similar to a proof search procedure. The subtraction examples I mentioned earlier (§11.2) are hand-crafted proof derivations in a certain logic, while the specification I'm about to describe is a procedure

for proof search in the same logic. A consequence is that we can reuse the vocabulary from existing work in logic [LMo7] and talk about negative and positive connectors, as well as synchronous and asynchronous phases. (§11.6 explores the connection with other, existing works in greater detail.)

Negative and positive connectors When considering:

$$\mathcal{V}.(P - Q) = \mathcal{V}'.R$$

one can think of P as the hypothesis, that is, the “current permission” that the type-checker carries through program points. Similarly, one can think of Q as the goal, that is, the thing that we wish to show. In \mathcal{V} , the rigid variables are part of the problem (hypothesis), while the flexible variables must be properly inferred in order for the problem to succeed (goal). $\mathcal{V}'.R$ represents the remaining hypotheses that were not consumed during the process.

The behavior of the connectors in Mezzo can be classified according to their polarity. Existential quantification, for instance, is a positive (also known as “synchronous”) connector, just like our products (tuples, constructors, conjunctions); when in the goal (that is, when performing a right introduction), it decomposes into a subgoal that require the proof search procedure to make a decision that may have later consequences (that is, this connector is not *right invertible*). Conversely, universal quantification, just like our function types, is a negative (also known as “asynchronous”) connector; when in the goal, it can be handled without the need for backtracking (that is, the connector *is* right invertible).

The negative and positive natures of quantifiers determine how these quantifiers are moved into the prefix. Existentially-quantified variables in the *hypothesis* are moved into the prefix as *rigid* variables: this is the re-normalizing addition (§10.4). Similarly, universally-quantified variables in the goal will be moved into the prefix as *rigid* variables: this is the re-normalizing subtraction (§11.1). This explains why the definitions of the renormalizing addition and subtraction make sense.

Synchronous and asynchronous phases Normalization only ever applies reversible rules (Remark 9.9): it corresponds to the *asynchronous* phase of the proof search procedure and only introduces rigid variables. Rules applied during the asynchronous phase have no influence on the rest of the derivation and can be applied in any order, without backtracking. Worded differently, \exists -normalization applies all the left-introduction rules for positive connectors while \forall -normalization applies all the right-introduction rules for negative connectors. These are all invertible rules.

The rules of subtraction, conversely, may strengthen the goal, introduce and/or instantiate flexible variables or consume permissions from the hypothesis. All of these are non-reversible operations: subtraction corresponds to the *synchronous* phase of proof search and is a backtracking procedure that performs exploration of the solution space. Making a decision during the synchronous phase may influence the outcome of the proof search.

Notations

Focused subtraction In the algorithmic presentation of the subtraction, we often wish to focus the subtraction on a particular variable x . (This is akin to focusing in a proof search procedure, where a subgoal is picked at the end of an asynchronous (“normalization”) phase). I write:

$$\mathcal{V}.(P * x @ t - x @ t') = \mathcal{V}'.R$$

This focusing step is analogous to the one used by the high-level type-checking algorithm (§10.3), and also implicitly duplicates (saves a copy) of $x @ t$ in P , if possible.

There is a slight pun related to the notation: the focalized judgement and the regular subtraction judgement share the same syntax; whenever $x @ t$ appears as in the judgement above, the reader should understand the subtraction to be focused on the variable x .

Kind-specific judgement The rules that are about to be presented present a subtraction at kind perm, either focused or “unfocused”. Other subtractions need to be performed, for instance when comparing parameters of a

type application. I push the pun further, and offer additional judgements at kind value and type. In the case that the kind of a type is unspecified, the proper judgement is taken to automatically apply.

$$\frac{\mathcal{V}, \mathcal{R}(z : \text{value}). P * z @ t - z @ t' = \mathcal{V}'.R}{\mathcal{V}. P * t - t' = \text{restrict}(\mathcal{V}', \mathcal{V}).R} \qquad \frac{\mathcal{V}. P - x @ y = \mathcal{V}'.R}{\mathcal{V}. P * x - y = \mathcal{V}'.R}$$

These two extra subtraction judgements do not introduce new rigid variables either in their output prefix. The \ominus judgement is extended in a similar way.

Subtraction modulo AC The subtraction rules are taken to operate modulo associativity, commutativity, and modulo equations. That is, the atomic permissions in P and Q may be reordered arbitrarily; if an equation $x = y$ is available, $\mathcal{V}.(P * x @ t - y @ t)$ is understood to rewrite transparently into $\mathcal{V}.(P * x @ t - x @ t)$. These features are handled transparently by the implementation; this is discussed in §11.4.

Propagating flexible instantiations The $\text{restrict}(\mathcal{V}', \mathcal{V})$ function was defined earlier (Definition 9.4). In essence, it allows to propagate the flexible instantiations from prefix \mathcal{V}' to prefix \mathcal{V} . This function is used throughout the present chapter whenever I wish to assert that a sub-operation is possible: we are not interested in the actual remainder, but *must* remember the instantiation choices that made this sub-operation possible.

Invariants

Remark 11.1 (Proper pre-conditions). *Subtraction always calls itself recursively with the right normalization pre-conditions.*

Proof. As I mentioned earlier, normalization does not descend into type application parameters, under quantifiers, and in codomains of arrows. For each of these situations, subtraction recursively calls itself using the circled versions $\textcircled{\otimes}$ and $\textcircled{\ominus}$ to make sure both sides are properly renormalized. $\textcircled{\text{m}}$

Lemma 11.2 (Prefix growth). *If $\mathcal{V}. P - Q = \mathcal{V}'.R$, then $\mathcal{V}' \leq \mathcal{V}$.*

Proof. The FORALL and EXISTS rules directly manipulate the prefix; however, they only introduce flexible bindings. Other rules recursively call onto $\textcircled{\ominus}$ or $-$. $\textcircled{\text{m}}$

A consequence of lemma 11.2 is that the subtraction does not introduce rigid variables, since the definition of \leq for prefixes only allows for extra *flexible* variables, not rigid ones.

Lemma 11.3 (Full normalization). *If $\mathcal{V}. P - Q = \mathcal{V}'.R$, where $\mathcal{V}. P$ is fully normalized and Q is \forall -normalized, then $\mathcal{V}'.R$ is fully normalized.*

Reviewing the rules

Algorithmic rules The rules of subtraction are presented in Figure 11.2. Just like in the type-checking judgement, an additional set of rules (Figure 11.3) clarifies what happens when variance comes into play. Most of these rules are syntax-directed; some, however, are non-deterministic and it is up to the actual implementation to implement strategies, or “heuristics” to determine what behavior it should adopt in such situations.

Most of these rules are *focused*, that is, they find matching variables x on both sides, and rely on the goal $x @ t'$ to find a corresponding $x @ t$ in the hypothesis. In this case, the $*$ connective on the left-hand side implicitly saves a copy of the permission that is focused (see “Notations” above).

Let me now review the rules.

APPORVAR is somewhat complex. The rule is triggered whenever the root of the application (the t variable) is the same on both sides.



In the case that the application has a non-empty list of parameters, the type t is necessarily rigid (and its variance known), as Mezzo disallows quantifying on variables with higher-order kinds. In the case that the application has an empty list of parameters (that is, in the case that we are comparing two variables), if one of them is flexible, the algorithm will unify the two, assuming this is legal. This is an inference decision; these decisions are discussed later on.

<p>APPORVAR</p> $\frac{\text{variance}(t) = \vec{v} \quad \mathcal{V}_i.\bar{P} \otimes u_i \overset{v_i}{\ominus} u'_i = \mathcal{V}_{i+1}._}{\mathcal{V}_0.P * x @ t \vec{u} - x @ t \vec{u}' = \mathcal{V}_n.P}$	<p>CONSAPP</p> $\frac{\mathcal{V}.P * x @ A \{\vec{f} = \vec{y}\} \ominus x @ t \vec{u} / A = \mathcal{V}'.R}{\mathcal{V}.P * x @ A \{\vec{f} = \vec{y}\} - x @ t \vec{u} = \mathcal{V}'.R}$	<p>FLOATINGAPPORVAR</p> $\frac{\text{variance}(p) = \vec{v} \quad \mathcal{V}_i.\bar{P} \otimes u_i \overset{v_i}{\ominus} u'_i = \mathcal{V}_{i+1}._}{\mathcal{V}.P * p \vec{u} - p \vec{u}' = \mathcal{V}.P}$
<p>TUPLE</p> $\frac{\mathcal{V}_i.P_i - y @ t'_i = \mathcal{V}_{i+1}.P_{i+1}}{\mathcal{V}_0.P_0 * x @ (= \vec{y}) - x @ (t') = \mathcal{V}_n.P_n}$	<p>CONSTRUCTOR</p> $\frac{\mathcal{V}_i.P_i - y @ t'_i = \mathcal{V}_{i+1}.P_{i+1}}{\mathcal{V}_0.P_0 * x @ A \{\vec{f} = \vec{y}\} - x @ A \{\vec{f} : t'\} = \mathcal{V}_n.P_n}$	
<p>FORALL-L</p> $\frac{\mathcal{V}, \mathcal{F}(X : \kappa).P \otimes P' - Q = \mathcal{V}'.R}{\mathcal{V}.P * \forall (X : \kappa) P' - Q = \mathcal{V}'.R}$	<p>FORALL-L-TYPE</p> $\frac{\mathcal{V}, \mathcal{F}(X : \kappa).P \otimes x @ t - Q = \mathcal{V}'.R}{\mathcal{V}.P * x @ \forall (X : \kappa) t - Q = \mathcal{V}'.R}$	<p>EXISTS-R</p> $\frac{\mathcal{V}, \mathcal{F}(X : \kappa).P \ominus Q = \mathcal{V}'.R}{\mathcal{V}.P - (\exists (X : \kappa) Q) = \mathcal{V}'.R}$
<p>EXISTS-R-TYPE</p> $\frac{\mathcal{V}, \mathcal{F}(X : \kappa).P \ominus x @ t' = \mathcal{V}'.R}{\mathcal{V}.P * x @ t - x @ \exists (X : \kappa) t' = \mathcal{V}'.R}$	<p>SINGLETON</p> $\mathcal{V}.P - x = x = \mathcal{V}.P$	<p>DYNAMIC</p> $\mathcal{V}.P * x @ \text{dynamic} - x @ \text{dynamic} = \mathcal{V}.P$
<p>UNKNOWN</p> $\mathcal{V}.P - x @ \text{unknown} = \mathcal{V}.P$	<p>EMPTY</p> $\mathcal{V}.P - \text{empty} = \mathcal{V}.P$	<p>STAR</p> $\frac{\mathcal{V}.P - Q = \mathcal{V}'.P' \quad \mathcal{V}'.P' - Q' = \mathcal{V}''.R}{\mathcal{V}.P - Q * Q' = \mathcal{V}''.R}$
<p>ARROW</p> $\frac{\mathcal{V}.\bar{P} \otimes Q \ominus r @ t_1 = \mathcal{V}'.P' \quad \mathcal{V}'.P' \otimes t_2 \ominus u_2 = \mathcal{V}''._}{\mathcal{V}.P * x @ t_1 \rightarrow t_2 - x @ (=r Q) \rightarrow u_2 = \mathcal{V}''.P}$	<p>FLEX-INTRO</p> $\frac{\mathcal{V}, \mathcal{F}(X : \kappa), \mathcal{V}'.P - Q = \mathcal{V}''.R}{\mathcal{V}, \mathcal{V}'.P - Q = \mathcal{V}''.R}$	
<p>FLEX-INST</p> $\frac{\mathcal{V}' \# T \quad X \# T \quad \mathcal{V} \vdash T : \kappa \quad \mathcal{V}, \mathcal{F}(X : \kappa = T), \mathcal{V}'.\text{empty} \otimes P \ominus Q = \mathcal{V}''.R}{\mathcal{V}, \mathcal{F}(X : \kappa), \mathcal{V}'.P - Q = \mathcal{V}''.R}$	<p>FLEX-SWAP</p> $\frac{\mathcal{V}, \mathcal{F}(X' : \kappa'), \mathcal{F}(X : \kappa), \mathcal{V}'.P - Q = \mathcal{V}''.R}{\mathcal{V}, \mathcal{F}(X : \kappa), \mathcal{F}(X' : \kappa'), \mathcal{V}'.P - Q = \mathcal{V}''.R}$	

Figure 11.2: The rules of subtraction

<p>VARIANCE-CONTRA</p> $\frac{\mathcal{V}.P * t' \ominus t = \mathcal{V}'._}{\mathcal{V}.P * t \overset{\text{contra}}{\ominus} t' = \mathcal{V}'._}$	<p>VARIANCE-CO</p> $\frac{\mathcal{V}.P * t \ominus t' = \mathcal{V}'._}{\mathcal{V}.P * t \overset{\text{co}}{\ominus} t' = \mathcal{V}'._}$	<p>VARIANCE-INV</p> $\frac{\mathcal{V}.P * t \ominus t' = \mathcal{V}'._ \quad \mathcal{V}'.P * t' \ominus t = \mathcal{V}''._}{\mathcal{V}.P * t \overset{\text{inv}}{\ominus} t' = \mathcal{V}'._}$	<p>VARIANCE-BI</p> $\mathcal{V}.P \otimes t \overset{\text{bi}}{\ominus} t' = \mathcal{V}._$
--	--	---	--

Figure 11.3: Variance-dependent subtraction

By kind-checking, the two applications have the same number of parameters and the parameters have the same kind. In the case that the type has no parameters, the situation is easy to grasp: we are left with a single type variable, such as in the example below.

$$\mathcal{R}(\text{int})\mathcal{R}(x : \text{value}).P * x @ \text{int} - x @ \text{int}$$

The general case is more complex. A first thing that we want to do is rely on the `COPYDUP` subsumption rule to *save* all possible duplicable permissions before recursively comparing the parameters of the type application. We thus compare the parameters with the hypothesis \bar{P} , that is, the restriction of P to its duplicable parts.



Comparing the parameters with P instead of \bar{P} would be unsound. Here is an example why:

$$\mathcal{V}.y @ \text{ref int} * x @ \text{list } (=y) - x @ \text{list } (\text{ref int})$$

The subtraction above must be forbidden. The permission on the left describes a list that contains pointers to y , along with a single permission for y . The permission on the right describes a list of distinct references. In the case that the list is longer than one, making this succeed would result in two exclusive permissions for the same variable y !

The judgement for comparing type parameters is taken to be the correct judgement depending on the kind: either the main judgement at kind `perm`, or one of the auxiliary judgements at kind `value` or `type I` introduced earlier.

Types are compared depending on their variance, using an auxiliary judgement parameterized by the variance v_i of the i -th parameter of t . This is similar to the type-checking judgement.

The remainder returned by the subtraction of parameters i is *not* re-used for subtracting parameters $i + 1$.



If we did keep the remainder after comparing two parameters, the algorithm would be unsound. Consider:

$$\mathcal{V}.x @ \text{list } (a \mid Q) - x @ \text{list } a$$

Keeping the remainder means that, after performing this subtraction, we obtain $\mathcal{V}.Q$ as a remainder. Clearly, if the list is empty, we should not assume Q to be available afterwards!

The flexible variable instantiations are also carried from subtraction i to subtraction $i + 1$. We want to compare all parameters using the original environment \mathcal{V}_0 : as we saw above, we are not interested in the remainder after comparing parameters; we merely want to assert that they are, truly, comparable. It is mandatory, though, that any instantiation of flexible variable that took place while comparing parameters i be retained for comparing parameters $i + 1$ all the way to the final remainder. The \ominus judgement takes care of applying `restrict`, meaning that the propagation of flexible variables is built-in, and that any rigid variables that were introduced in the course of comparing the parameters are discarded. This rule hence ensures the $\mathcal{V}' \leq \mathcal{V}$ post-condition.



Not propagating flexible variable instantiations would be unsound. Consider, for instance:

$$\mathcal{R}(x : \text{value}), \mathcal{F}a. x @ t a a - x @ t \text{int } ()$$

The subtraction above would succeed, which, unless t is bivariant in both parameters, is very clearly unsound: a cannot have two values simultaneously!

Finally, one should note that in order to satisfy the pre-conditions of subtraction, we re-normalize both the left and right sides.

`CONSAPP` triggers an application of the `FOLD` subsumption rule: in the event that the left-hand side is a constructor A of type t , we can *strengthen* our current goal so as to match the current constructor type. We re-use the projection notation (§9.1).

`FLOATINGAPPORVAR` deals with permissions that are not anchored to a particular variable, that is, abstract parameterized permissions. Just like `APPORVAR`, it encompasses the case where the argument list is empty, which is that of a type variable with kind `perm`.

Rules `TUPLE` and `CONSTRUCTOR` descend into structural permissions. The various well-formedness checks guarantee that, in the constructor case, the fields match. In the tuple case, however, an ill-typed program may

end up comparing tuples with conflicting lengths. The \exists -normalization pre-condition allows us to assume that all fields for the tuple on the left-hand side are singletons. We then recursively call ourselves, and just like we mentioned already, chain the premises.

The rules for quantifiers **FORALL-L**, **FORALL-L-TYPE**, **EXISTS-R** and **EXISTS-R-TYPE** are discussed in the next paragraph.

The next rules **DYNAMIC**, **UNKNOWN**, **EMPTY** are standard. The only interesting thing to note is that dynamic is made “automatically” available via normalization rule (11). It is important for the normalization to take care of this: if $x @ a$ becomes, via a flexible variable instantiation, $x @ t$ with t exclusive, we need to make $x @$ dynamic available as well.

Rule **STAR** is interesting in that it is highly non-deterministic, because of associativity and commutativity. That is, we may choose any order for scheduling atomic permissions to be subtracted. The next section covers implementation techniques for dealing with the problem of selecting the “next atomic permission” to be subtracted.

ARROW is perhaps the most powerful rule. We mentioned earlier (§8.2) that one could derive **COARROW2** from **HIDEDUPLICABLEPRECONDITION**. I now give the derivation, as **COARROW2** constitutes the theoretical justification for the subtraction rule.

If the two hypotheses $h_1 = (u_1 | P) \leq t_1$ and $h_2 = (t_2 | P) \leq u_2$ hold, and P is duplicable, then:

$$\begin{array}{c} P * x @ t_1 \rightarrow t_2 \\ P * x @ t_1 \rightarrow (t_2 | P) \\ P * x @ (u_1 | P) \rightarrow u_2 \\ x @ u_1 \rightarrow u_2 \end{array} \begin{array}{c} \text{COPYDUP} \\ \leq \\ \text{COARROW}, h_1, h_2 \\ \leq \\ \text{HIDE} \\ \leq \end{array}$$

Thanks to the \forall -normalization pre-condition for the right-hand side of the subtraction, we can assume u_1 is of the form $(=r | Q)$. This allows us to follow the first premise of **COARROW2** and compare (contra-variantly) the domains using the restriction of P to its duplicable parts, that is, \bar{P} . (An example that illustrates why this particular form for u_1 is important follows.)

The codomains are not required by normalization to be named: we use the subtraction judgement at kind type to compare them (co-variantly). Any flexible instantiations that were performed during these two operations are propagated to the output prefix \mathcal{V}'' . No permission was consumed or created, since function types are duplicable: we return the original permission P .

Interestingly, and contrary to **APPORVAR**, the remainder from comparing the domains is carried on to the comparison of codomains, even though the remainder may contain non-duplicable permissions. This corresponds to an η -expansion rule, and is sound thanks to **ETA** which is derivable using **FRAME** and **COARROW**. A complete example that leverages the full power of the rule is presented in §11.5.



One may wonder why $\bar{P} \otimes Q$ performs a re-normalization since, after all, the right-hand side $x @ (=r | Q) \rightarrow u_2$ is \forall -normalized, meaning that Q is itself \exists -normalized (Figure 9.5). The reason is, we want normalization rules (4a), (4b) and (5) to apply. Worded differently, that P and Q are normalized does not imply that $P * Q$ is normalized.

Treatment of quantifiers Quantifiers have to be introduced in the *right order*. To see why, let us consider a few examples before reviewing the quantifier-related rules.

$$\mathcal{R}(x : \text{value}). (x @ \exists a.a - x @ \exists b.b)$$

Let us imagine for a second that we do not have normalization, and have instead the natural **EXISTS-L** rule, which opens an \exists quantifier on the left as a rigid variable (\exists is a positive connector; when in the goal, it transforms into a universal quantifier, that is, a rigid variable).

$$\frac{\frac{\text{FAILURE}}{\mathcal{R}(x : \text{value}), \mathcal{F}b, \mathcal{R}a. (x @ a - x @ b)}}{\mathcal{R}(x : \text{value}), \mathcal{F}b. (x @ \exists a.a - x @ b)} \text{FLEXINSTANTIATION}}{\mathcal{R}(x : \text{value}). (x @ \exists a.a - x @ \exists b.b)} \text{EXISTS-L}$$

$$\frac{\mathcal{R}(x : \text{value}). (x @ \exists a.a - x @ \exists b.b)}{\mathcal{R}(x : \text{value}). (x @ \exists a.a - x @ \exists b.b)} \text{EXISTS-R}$$

The derivation above ends with a failure, as instantiating b into a would create an ill-scoped equation. Indeed, we made a wrong choice by applying EXISTS-R instead of EXISTS-L in the first place. This failure could be solved by always applying EXISTS-L (and FORALL-R) first. This is not sufficient, though. Let us consider a second example (kinds are omitted, for brevity).

We know for a fact that $t \rightarrow u \equiv \forall(x : \text{value}) (=x \mid x @ t) \rightarrow u$ (COMMUTEARROW, DECOMPOSEANY); we wish to perform this subtraction, that is, to make sure our algorithm can derive this fact. The subtraction ought to succeed: taking any x along with $x @ a$ really is the same thing as taking an argument of type a . We assume a naïve ARROW rule which merely compares the types of the domains, using a placeholder rigid binding z .

$$\frac{\frac{\text{FAILURE}}{\mathcal{R}a, \mathcal{R}f, \mathcal{F}x, \mathcal{R}z.z @ a - z @ (=x \mid x @ a)} \text{ARROW}}{\mathcal{R}a, \mathcal{R}f, \mathcal{F}x.f @ (=x \mid x @ a) \rightarrow () - f @ a \rightarrow ()} \text{FORALL-L}}{\mathcal{R}a, \mathcal{R}f.f @ \forall x.(=x \mid x @ a) \rightarrow () - f @ a \rightarrow ()} \text{FORALL-L}$$

Using the naïve rule, the algorithm is stuck as there is no opportunity to apply FORALL-R. We end up with an ill-scoped equality $\mathcal{F}x, \mathcal{R}z.z = x$ which the type-checker is unable to take into account, and discards (§9.5). Indeed, finding an x such that for any z , we have $x = z$ is impossible, however powerful the type-checker may be. The z quantifier was introduced *too late*. This subtraction thus fails as well.

We therefore need to *eagerly introduce rigid quantifiers* into the subtraction prefix. Looking at rule ARROW, we need to introduce the rigid variables from u_1 *before* the flexible variables induced by $\forall(\vec{X} : \vec{\kappa}) t_1 \rightarrow t_2$ (FORALL-L-TYPE). This is the reason why we *extrude* existential quantifiers from u_1 , then hoist them out of the arrow as *universal* quantifiers (Figure 9.5). This is the *raison d'être* of \forall -normalization. More generally, normalization (Chapter 9) corresponds to the standard asynchronous phase in the proof search literature. The pre-conditions of subtraction, which are that P be \exists -normalized and Q be \forall -normalized, embody this requirement.

These techniques for introducing quantifiers properly are not a guarantee of completeness; we need, however, to properly tackle this aspect in order to have better success rates for subtraction operations.

Subtraction therefore *never adds* rigid variables into the prefix because it never encounters them: normalization took care of introducing them already; this is guaranteed by the subtraction's normalization preconditions. We thus only need the four rules EXISTS-R, EXISTS-R-TYPE, FORALL-L and FORALL-L-TYPE.




The rule FORALL-L is only ever triggered, say, if a function returns a universally-quantified permission, such as $\forall(X : \kappa) p X$ where p is an abstract predicate. It is hard to introduce this flexible variable at “the right time”. Other rules are syntax-directed: they are focused on a variable x and one can figure out the next rule to be applied by matching on the head constructor of the type t . (We thus know when to introduce a flexible variable, such as in EXISTS-R-TYPE.) In the case of FORALL-L, no syntactic hint tells us when to introduce this flexible variable in the prefix.

Handling of flexible variables Figure 11.2 gives three rules for handling flexible variables. FLEX-INTRO says one can introduce a flexible variable at any position in the prefix (in practice, our implementation only allows this as long as levels are not modified, without loss of generality). FLEX-SWAP allows one to swap two flexible variables at the same level. FLEX-INST defines what happens when we make an instantiation choice for a flexible variable: all occurrences of the variable are replaced with T , and renormalizations on both sides occur.

These rules are very declarative, and allow the type-checker to instantiate arbitrarily, at any time, any flexible variable any way they want. This is, naturally, a loose specification, and an actual implementation will take care of implementing more specific rules for this (§11.7).

Correctness Perhaps the most important result of this section is that subtraction is correct.

Theorem 11.4 (Subtraction correctness). *If $\mathcal{V}.P - Q = \mathcal{V}'.R$, then $\mathcal{V}' \times P \leq \mathcal{V}' \times (Q * R)$. In other words, subtraction is a semi-algorithm for deciding subtyping and computing unused hypotheses.*

Proof. The result relies on reviewing the rules, and making sure that their specification corresponds to actual subtraction rules from §8.2. The only tricky rule is that for arrow types, which I commented earlier. Rules for flexible variables also deserve careful review, as they may create ill-scoped instantiation choices; the premises from FLEX-DEFAULT-L ensure this does not happen. 

$$\begin{array}{c}
\text{SUBSTEQUALS}_1 \\
\frac{P = P * x = y \quad \mathcal{V}.[y/x]P - Q = \mathcal{V}'.R}{\mathcal{V}.P - Q = \mathcal{V}'.R}
\end{array}
\qquad
\begin{array}{c}
\text{SUBSTEQUALS}_2 \\
\frac{P = P * x = y \quad \mathcal{V}.[x/y]P - Q = \mathcal{V}'.R}{\mathcal{V}.P - Q = \mathcal{V}'.R}
\end{array}$$

Figure 11.4: Equational rewriting in subtraction

$$\begin{array}{c}
\text{FLEX-TUPLE-L} \\
\frac{\mathcal{V}, \mathcal{F}(\vec{\beta} : \text{type})\mathcal{F}(a : \text{type} = (\vec{\beta})), \mathcal{V}'. \\ P \otimes x @ (\vec{t}) - x @ a = \mathcal{V}'' .R}{\mathcal{V}, \mathcal{F}(a : \text{type}), \mathcal{V}'.P * x @ (\vec{t}) - x @ a = \mathcal{V}'' .R}
\end{array}
\qquad
\begin{array}{c}
\text{FLEX-CONSTRUCTOR-L} \\
\frac{\mathcal{V}, \mathcal{F}(\vec{\beta} : \text{type})\mathcal{F}(a : \text{type} = A \{\vec{f} : \vec{\beta}\}), \mathcal{V}'. \\ P \otimes x @ A \{\vec{f} : \vec{t}\} - x @ a = \mathcal{V}'' .R}{\mathcal{V}, \mathcal{F}(a : \text{type}), \mathcal{V}'.P * x @ A \{\vec{f} : \vec{t}\} - x @ a = \mathcal{V}'' .R}
\end{array}$$

$$\begin{array}{c}
\text{FLEX-DEFAULT-L} \\
\frac{\mathcal{V}' \# t \quad a \# t \\ \mathcal{V}, \mathcal{F}(a : \text{type} = t), \mathcal{V}'.P \otimes x @ t \ominus x @ a = \mathcal{V}'' .R}{\mathcal{V}, \mathcal{F}(a : \text{type}), \mathcal{V}'.P * x @ t - x @ a = \mathcal{V}'' .R}
\end{array}$$

Figure 11.5: Instantiation of flexible variables (-R rules omitted)

11.4 Implementing subtraction

Naturally, the specification we just saw leaves a lot of room for implementation decisions. The present section reviews the choices left to the algorithm as well as some implementation strategies.

Equations

One thing that was unspecified by the algorithm is the treatment of equations. Actually, two rules were missing from Figure 11.2 and are shown in Figure 11.4. These rules mean that one can rewrite both the hypothesis and the goal using any equation available at hand.

As we saw earlier (§9.5), equations are handled transparently via our representation of permissions: the two rules are thus applied implicitly whenever needed.

Dealing with non-determinism

The subtraction algorithm leaves many things up to the actual implementation. I mentioned a few of them already; the present section reviews the various sources of non-determinism and the actual heuristics used by the type-checker.

Instantiation choices Figure 11.5 presents a set of more algorithmic rules for instantiating flexible variables. These rules are triggered by syntactically matching on whatever permission is available for x in the hypothesis. These rules apply whenever the flexible variable is on the right-hand side; a set of symmetrical rules (omitted) exist for whenever the flexible variable is on the left-hand side.

These rules are not fully general, and do not consider all possible solutions, as the exploration space is too large. Our algorithm is thus not complete. One should understand these rules as replacing the loosely-specified FLEX- rules from Figure 11.2 with a more actionable specification.

The two important rules are FLEX-TUPLE-L and FLEX-CONSTRUCTOR-L. Rule FLEX-DEFAULT-L is applied if no other rule matches. These two rules say that when trying to instantiate a flexible variable with a tuple type or a constructor type, one should first break down the flexible variable into a tuple or a constructor, whose fields are themselves flexible variables.

Doing so is correct. It amounts to introducing fresh flexible variables for the fields (FLEX-INTRO), then instantiating a suitably (FLEX-INST).

FLEX-DEFAULT-L is very general too, as we may have the following conjunction, which leads to multiple choices for applying the rule:

$$x @ t * x @ =x * x @ \text{unknown}$$

The last two permissions are always available for x : the algorithm is thus always faced with several choices for applying the rule. The algorithm implements a heuristic, which is to *favor* instantiating a flexible variable with t if such a non-singleton t exists. The algorithm backtracks, meaning it will also, eventually, consider other instantiation choices. It is crucial, however, that the choice that usually makes more sense be considered first: lacking a good heuristic, the type-checker would just explore too many solutions before finding the right one.



To see why the heuristics for FLEX-DEFAULT-L are tuned towards non-singleton types, consider the following subtraction problem (some intermediary steps omitted):

$$\begin{array}{l} \mathcal{R}(a : \text{type}, h, t : \text{value})\mathcal{F}(a : \text{type}). \\ \quad h @ a \\ * t @ \text{list } a \qquad \qquad \qquad - \quad l @ \text{Cons } \{\text{head} : a; \text{tail} : \text{list } a\} \\ * l @ \text{Cons } \{\text{head} = h; \text{tail} = t\} \end{array}$$

This is a standard subtraction problem. It pops up, for instance, when type-checking the `list::length` function, when the recursive call takes place.

The algorithm tries to recursively subtract the head fields, meaning that the following sub-operation takes place:

$$\begin{array}{l} \mathcal{R}(a : \text{type}, h, t : \text{value})\mathcal{F}(a : \text{type}). \\ \quad h @ a \qquad \qquad \qquad - \quad h @ a \\ * h @ =h \end{array}$$

Lacking a good heuristic for FLEX-DEFAULT-L, if one picks $a = =h$, the next step of the subtraction is, after performing substitution:

$$\begin{array}{l} \mathcal{R}(a : \text{type}, h, t : \text{value}). \\ \quad h @ a \\ * t @ \text{list } a \qquad \qquad \qquad - \quad l @ \text{Cons } \{\text{head} : =h; \text{tail} : \text{list } =h\} \\ * l @ \text{Cons } \{\text{head} = h; \text{tail} = t\} \end{array}$$

The algorithm then tries to recursively subtract the tail fields, which fails, as the following subtraction cannot succeed:

$$t @ \text{list } a - t @ \text{list } =h$$

(Avoiding instantiating flexible variables onto singleton types also helps avoiding the “local variable” problem in §12.4.)

To see why the rule FLEX-TUPLE-L is important, consider the following subtraction problem (some intermediary steps omitted):

$$\begin{array}{l} \mathcal{R}(a, b : \text{type}, h, t, x, y : \text{value})\mathcal{F}(a : \text{type}). \\ \quad h @ (=x, =y) \\ * x @ a \\ * y @ b \qquad \qquad \qquad - \quad l @ \text{Cons } \{\text{head} : a; \text{tail} : \text{list } a\} \\ * t @ \text{list } (a, b) \\ * l @ \text{Cons } \{\text{head} = h; \text{tail} = t\} \end{array}$$

This is a standard subtraction problem. It pops up, for instance, when type-checking the `list::assoc` function, when the recursive call takes place.

Without FLEX-TUPLE-L, the algorithm would pick $a = (=x, =y)$, which gives:

$$\begin{array}{l} \mathcal{R}(a, b : \text{type}, h, t, x, y : \text{value})\mathcal{F}(a : \text{type}). \\ \quad h @ (=x, =y) \\ * x @ a \\ * y @ b \qquad \qquad \qquad - \quad l @ \text{Cons } \{\text{head} : (=x, =y); \text{tail} : \text{list } (=x, =y)\} \\ * t @ \text{list } (a, b) \\ * l @ \text{Cons } \{\text{head} = h; \text{tail} = t\} \end{array}$$

The algorithm then tries to recursively subtract the tail fields, which fails, as the following subtraction cannot succeed:

$$t @ list (a, b) - t @ list (=x, =y)$$

Focusing strategies I mentioned earlier that the algorithm, when working, was focused on a single variable x . However, it is rarely the case that a subtraction problem is made up of a single permission. In the general case, one wants to subtract $Q = q_1 * \dots * q_n$. The specification allows for performing the smaller subtractions in any order. The implementation, naturally, has a strategy for working this out.

The algorithm first breaks down Q into a conjunction of basic permissions, which are either anchored permissions of the form $x @ t$, or floating permissions of the form $p \vec{q}$.

A floating permission $p \vec{q}$ can be subtracted if p is rigid. Indeed, the type-checker needs a way to identify the desired permission; in the case where p is flexible, the algorithm refuses to explore every possible instantiation choice, and this branch in the exploration space fails. However, if p is rigid, then the type-checker can easily look up all applications of p in the left-hand side, and try each one in turn using `FLOATINGAPP`.

An anchored permission $x @ t$ can be subtracted as long as x is rigid. Again, if x were flexible, there would be too many instantiation choices to consider; the type-checker hence fails in that situation. Assuming x is rigid, the type-checker can easily look up all permissions attached to x , and try each one in turn.

The algorithm keeps a work list of “pending” permissions $q_1 \dots q_n$, and repeatedly takes out permissions that can be successfully subtracted. The algorithm does not backtrack when there is a choice between several suitable permissions.

The algorithm has special cases for equations: for instance, subtracting $x @ =y$ where both x and y are flexible is possible, as it suffices to unify x and y . This is not complete, though, as we may want to instantiate both x and y onto a third variable.

Backtracking points There are two main reasons for backtracking.

- Several permissions are available for a single variable. This means that when subtracting $x @ t$, there are multiple choices for the permission to focus on the left-hand side. We cannot know in advance which permission should be used, meaning that the algorithm needs to *explore*.
- Subtraction corresponds to the synchronous phase of proof search, which applies non-invertible rules. These non-invertible rules may have an influence on the remainder of the derivation, meaning that when faced with a choice between multiple non-invertible rules to apply, the type-checker also needs to *explore*.

The first case was covered above: we saw that depending on which permission one focuses on, different instantiation choices would follow, hence influencing the outcome of the subtraction operation.

The second case is less obvious; let us see an example. Rules `FORALL-L` and `EXISTS-R` are non-reversible: these two are quantifier elimination rules. At first glance, it may seem like they can commute: both of them introduce flexible variables which, after all, can commute in the prefix. They may, however, uncover different types.

$$\mathcal{R}(z : \text{value}).z @ \exists a, \forall \beta.(a, \beta) \ominus z @ \exists a', \forall \beta'.(a', \beta')$$

Consider the subtraction above. Initially, renormalization occurs, which gives:

$$\mathcal{R}(z : \text{value})\mathcal{R}(a).z @ \forall \beta.(a, \beta) - z @ \exists a', \forall \beta'.(a', \beta')$$

Now, however, the algorithm is faced with a choice between the two elimination rules. Applying `FORALL-L` yields:

$$\mathcal{R}(z : \text{value})\mathcal{R}(a)\mathcal{F}(\beta).z @ (a, \beta) \ominus z @ \exists a' \forall \beta'.(a', \beta')$$

No renormalization happens. Next, only `EXISTS-R` can apply.

$$\mathcal{R}(z : \text{value})\mathcal{R}(a)\mathcal{F}(\beta)\mathcal{F}(a').z @ (a, \beta) \ominus z @ \forall \beta'.(a', \beta')$$

Renormalization gives:

$$\mathcal{R}(z : \text{value})\mathcal{R}(a)\mathcal{F}(\beta)\mathcal{F}(a')\mathcal{R}(\beta').z @ (a, \beta) \ominus z @ (a', \beta')$$

This is a failure, as we picking $\beta = \beta'$ would be ill-scoped. However, if we chose to apply EXISTS-R when faced with the earlier choice, we obtain:

$$\mathcal{R}(z : \text{value})\mathcal{R}(a)\mathcal{F}(a')\mathcal{R}(\beta')\mathcal{F}(\beta).z @ (a, \beta) - z @ (a', \beta')$$

Which admits a trivial solution.

The type-checker has no way to guess which one of these two rules should be applied, and explores both branches. We could probably improve this, and use a more sophisticated data structure, rather than plain old levels. This would possibly allow us to defer the choice and explore both branches in parallel. However, this would lead further away from a type system, probably into the domain of automated theorem proving. Out of the several hundred test cases that make up for the Mezzo test suite, only one of them exhibits this critical pair.

Termination issues Subtraction currently does not terminate in certain (rare) cases, due to the non-terminating nature of normalization in the presence of recursive, one-branch data types (discussion in the proof of Lemma 9.5).

Another source of non-termination is highly-recursive, advanced programming patterns such as Landin's knot (Figure 4.14). This example was discussed earlier (§4.6), and I mentioned that it was hard to type-check.

```

1  data patched (r: value) a b =
2    Patched { contents : (x : a | consumes r @ patched r a b) -> b }
3
4  val fix [ a, b ] (ff : (a -> b) -> (a -> b)) : a -> b =
5    let r = newref () in
6
7    let f (x : a | consumes r @ patched r a b) : b =
8      let self = r.contents in
9      ff self x
10   in
11
12   r.contents <- f;
13   tag of r <- Patched;
14   f

```

Figure 11.6: Landin's knot

The example is shown again in Figure 11.6. This code is *currently* accepted by the Mezzo type-checker. It is a piece of luck, though. Consider line 9 in Figure 4.14; the type-checker needs to perform the following sequence of operations to justify why the function call `ff self` is correct. I discuss some key steps of the subtraction, and I use the internal syntax.

The algorithm first needs to match the pre-condition of `ff`: we need the argument `self` to have the desired type $a \rightarrow b$. The subtractor is thus:

$$r @ \text{Patched} \{ \text{contents} = \text{self} \} * \text{self} @ (a \mid r @ \text{patched } r a b) \rightarrow b - \text{self} @ a \rightarrow b$$

The left-hand side shows the relevant bits of the current permission while the right-hand side of the subtraction shows the desired goal. Since the goal is an arrow, subtraction uses ARROW and, among other premises, compares the domains contra-variantly. This leads us onto the following sub-operation, where \dots denotes the set of duplicable permissions that remain available.

$$\dots * a \ominus (a \mid r @ \text{patched } r a b)$$

The goal is \forall -normalized via \ominus (this is a premise of ARROW); a consequence is that the patched type is expanded into a structural type via normalization rule (9).

$$\dots * a - (a \mid r @ \text{Patched} \{ \text{contents} = \text{self} \} * \text{self} @ (a \mid r @ \text{patched } r a b) \rightarrow b)$$

The type variables a subtract successfully. We are left with:

$$\dots - r @ \text{Patched} \{ \text{contents} = \text{self} \} * \text{self} @ (a \mid r @ \text{patched } r a b) \rightarrow b$$

The algorithm selects the permission for self as the following sub-goal:

$$\dots - \text{self} @ (a \mid r @ \text{patched } r a b) \rightarrow b$$

Recall that \dots represents duplicable permissions that were captured, meaning that we possess, among other things, a permission for self in \dots :

$$\dots * \text{self} @ (a \mid r @ \text{patched } r a b) \rightarrow b - \text{self} @ (a \mid r @ \text{patched } r a b) \rightarrow b \quad (*)$$

This is a function type, meaning that, again, the domains are compared:

$$\dots * (a \mid r @ \text{patched } r a b) - (a \mid r @ \text{patched } r a b)$$

This requires, among other things, to obtain the permission for r , since it is in the goal:

$$\dots - r @ \text{patched } r a b$$

If the type-checker expands this goal and keeps going, unfortunately, it falls into an infinite loop.

Expanding the goal is absolutely mandatory: I have argued relentlessly that permissions should be normalized in order to eagerly introduce rigid binders; this implies expanding one-branch data types in order to fetch any rigid variables that may be hidden underneath. This step cannot be discarded, or most realistic programs would fail to type-check.

There is hence a conflict between the rules of normalization and this particular example which somehow requires these rules not to be applied. This particular example *does* type-check, after I took the code and, being somewhat proficient as to the inner workings of the type-checker, fine-tuned it as follows...

The astute reader may have noticed that the name introduction construct in the definition of `patched` is useless, since x is not used anywhere. One practical advantage, however, is that the function type in the definition of `patched` now becomes syntactically equal to that of f . Indeed, due to desugaring, having a name introduction construct or not yields different internal types. The subtraction operation is equipped with a syntactic check: if two types are syntactically equal, other rules are skipped and the subtraction returns immediately. In this particular case, this means that step $(*)$ is an immediate success and does not recurse into the comparison of the domains, hence avoiding an infinite loop.

Other instances of this pattern have popped up over the course of the past few years, most of the time due to F. Pottier's extreme Mezzo programs. Solving these cases of non-termination would probably require a complete re-work of the type-checker routines.

Sources of incompleteness The subtraction algorithm is not complete: every flexible variable can potentially be instantiated into $\exists(a)(p : \text{perm})(a \mid p)$: at any time, one may pack permissions into a variable. Since p can be anything, ranging from empty to the entire set of available permissions, this means that there are just too many cases for an algorithm to consider them all. The algorithm is hence not complete.

Even if we were to actually split every a into $(a \mid p)$, doing this for every flexible variable would probably raise termination issues.

The general feeling is thus that subtraction, because there are so many possible instantiation choices, cannot, fundamentally, be complete. An argument in favor of this is that, having polymorphism at kind type, Mezzo embeds System F with subtyping, which is known to be undecidable [TU96]. Mezzo, however, does not feature the `DISTRIB` rule; System F with subtyping remains, however, undecidable, even without this rule [Chr98].


That being said, we conjecture that subtraction is complete whenever there are no user-provided quantifiers.

Conjecture 11.5 (Completeness of subtraction). *Subtraction is decidable for a fragment of Mezzo that excludes user-provided universal quantification, i.e. that excludes the syntactic construct $[a] \dots$ and $\{a\} \dots$, and that excludes recursive, one-branch data types.*

$$\begin{array}{c}
\text{SINGLETON} \\
\mathcal{R}(f, a, r)\mathcal{F}(x = r).r@a - r = r = \\
\mathcal{R}(f, a, r)\mathcal{F}(x = r).r@a \\
\hline
\text{FLEX-INST} \quad \mathcal{R}(f, a, r)\mathcal{F}(x).r@a - r@a = \\
\mathcal{R}(f, a, r)\mathcal{F}(x = r).r@a \\
\hline
\text{ARROW} \\
\mathcal{R}(f, a, r)\mathcal{F}(x).f@a = x \rightarrow =x - f@a (=r \mid r@a) \rightarrow a = \mathcal{R}(f, a, r)\mathcal{F}(x = r).\text{empty} \\
\hline
\text{FORALL-L} \\
\mathcal{R}(f, a, r).f@a \forall x.=x \rightarrow =x - f@a (=r \mid r@a) \rightarrow a = \mathcal{R}(f, a, r)\mathcal{F}(x = r).\text{empty} \\
\hline
\text{NORM} \\
\mathcal{R}(f).f@a \forall x.=x \rightarrow =x \ominus f@a \forall a.a \rightarrow a = \mathcal{R}(f, a, r).\text{empty}
\end{array}
\begin{array}{c}
\text{APPROVAR} \\
\mathcal{R}(f, a, r)\mathcal{F}(x = r)\mathcal{R}(z).r@a * z@a = x - r@a = \\
\mathcal{R}(f, a, r)\mathcal{F}(x = r)\mathcal{R}(z).z = x \\
\hline
\text{EQUATION} \\
\mathcal{R}(f, a, r)\mathcal{F}(x = r)\mathcal{R}(z).r@a * z@a = x - z@a = \\
\mathcal{R}(f, a, r)\mathcal{F}(x = r)\mathcal{R}(z).z = x
\end{array}$$

Figure 11.7: A complete subtraction example (kinds omitted for readability)

Proof. (Sketch.) The only possible quantifications left are on type variables at kind value, via the translation of the name binding construct. Moreover, these constructs are translated in a way that the quantified variables are always *reachable*, that is, are always connected to a *root* via a path that goes through structural types (tuples, constructors, and singletons). This is the concept of *unique value* I mentioned earlier. Starting from the root, the algorithm will thus always recursively perform the correct unifications thanks to the structural types.

We conjecture that eliminating “dangerous” data types allows not only normalization, but also subtraction to terminate. 

11.5 A complete example

Consider the code below, where comments denote the current permission.

```

fun (f: [x] (=x -> =x)): [a] (a -> a) =
  fun [a] (x: a): a =
    (* f @ [x] (=x -> =x) * x @ a *)
    f x
    (* ret @ =x * x @ a *)

```

The code allows transforming a function of type $\forall(x : \text{value}).=x \rightarrow =x$ into a function of type $\forall a.a \rightarrow a$. In essence, a singleton type does not carry any useful information; it merely reveals the existence of a value, and does not carry any ownership of the heap. Therefore, a function that takes a value and returns it, without the ability to do anything with it, necessarily preserves its argument untouched.

The code above performs an η -expansion, which manages to prove, using runtime code, that $\forall(x : \text{value}).=x \rightarrow =x \leq \forall a.a \rightarrow a$. Writing code for this purpose is unnecessary: just like the permission $x @ a$ is preserved across the function call by the frame rule in the *code*, the complex ARROW rule for subtraction allows to prove the subtyping fact above *without writing code*, by carrying leftover permissions from the domains into the codomains.

Indeed, the subtraction rule for arrows feeds the remainder from the domain subtraction into the codomain subtraction, hence performing an implicit η -expansion. Figure 11.7 contains the complete derivation for this example.

The key steps in the derivation are highlighted in **red**. The first step is renormalization, which rewrites the domain of the function in the goal. The next important step is the comparison of domains, which requires us to pick an instantiation choice $x = r$. The comparison of domains produces a *remainder* $r@a$ which is then used for comparing the codomains. Thanks to the equation $z@a = x$ in the hypothesis, we can rewrite the goal so as to prove the desired property.

As a side note, proving the converse subtyping relation (shown below) is trivial, as it is just a matter of instantiating a with $=x$.

$$\forall a.a \rightarrow a \leq \forall(x : \text{value}).=x \rightarrow =x$$

11.6 Relating subtraction to other works

Subtraction, depending on the reader’s background, may feel similar to *proof search* in logic, or *entailment* in separation logic. The present section tries to connect the procedure described in the present chapter with these two lines of work.

Proof search in logic

A natural candidate for connecting the type system of Mezzo with a well-defined logical system is linear logic [DCM10, LMo7]: indeed, many concepts from the literature on linear logic appear in one form or another in Mezzo.

The notion of duplicable vs. non-duplicable permission is reminiscent of the $!$ modality, which allows one to freely duplicate a hypothesis via the $!$ -contraction rule. Being duplicable, however, is not explicit, as in linear logic, but implicitly defined by the “is duplicable” predicate. Viewing permissions as *resources* that, just like in linear logic, one may or may not duplicate, remains a good way to connect the two systems.

In a subtraction problem $P - Q$, one may replace the “ $-$ ” symbol with “ \vdash ”: this makes it all the more apparent which of P and Q is the hypothesis and the goal. The way we eliminate quantifiers, either via normalization or during subtraction corresponds to the standard introduction and elimination rules for quantifiers.

Flexible variables are a well-known implementation technique for inferring the parameters of quantifier eliminations; it is used in ML-style type inference too.

The type system of Mezzo does not feature all of the rules from linear logic, though. There is no such thing as weakening a goal, for instance, via right contraction. Also, $*$ may be interpreted as either the additive or the multiplicative conjunction, depending on the operands; there is no built-in operator, however, for disjunction.

Alternating between subtraction steps and normalization phases is, as I mentioned earlier, reminiscent of the synchronous and asynchronous phases of linear logic. The asynchronous phase would correspond to normalization, also called goal-reduction in proof search techniques, while subtraction would correspond to synchronous steps, that is, the actual proof search.

Asynchronous steps have no consequence on the rest of the derivation: they may be applied in any order. In Mezzo, this corresponds to the various normalization rules, which reduce the hypothesis into a conjunction of *atomic* permissions. Asynchronous steps are applied at key points, and correspond to circled operators \otimes and \ominus . They do not introduce flexible variables, but only introduce rigid variables.

Synchronous steps have an influence on the rest of the derivation: eliminating a quantifier has consequences on whether the derivation will succeed or not, depending on how the quantifier is eliminated. This corresponds to the subtraction rules. These rules introduce flexible variables: they need to *guess* the right instantiations. These rules also backtrack, as they need to explore the search space, that is, perform the actual proof search.

When one is faced with a goal P that is made up of a conjunction $P_1 * \dots * P_n$, selecting P_i is akin to a *focusing* step. When one performs an `assert` statement, this is equivalent to an explicit *cut* where the user provides the intermediary result (the type-checker cannot try all possible cuts).

It should be noted that the use of flexible variables actually allows to *defer* the instantiation choices: whenever introducing an existential quantifier in the goal, we don’t always *have to* come up with a witness. The risk is then to later on instantiate a flexible variable with hypotheses that *did not exist* back when the quantifier was introduced. Using levels, however, guarantees that whenever settling on an instantiation choice, we pick one choice that is (in retrospect) legal.

Mezzo, being a type system, differs significantly from the MALL fragment of linear logic. For instance, there is no negation in Mezzo: our language of permission has no $\neg P$. Moreover, the subtyping relation cannot call functions, that is, $x @ t \rightarrow u * y @ t \not\leq \exists (z : \text{value}) u$. (We would need some sort of ghost function mechanism for this.)

The underlying logic of Mezzo is thus intuitionistic, rather than classic: we have no way, for instance, of proving Peirce’s law $((P \rightarrow Q) \rightarrow P) \rightarrow P$.

Data types encode disjunction; they are, however, tagged, meaning that the treatment of disjunctions is much easier: the type-checker does not need to explore both branches in parallel, and can discriminate on the tag.

Perhaps more importantly, the various, powerful subsumption rules render Mezzo more complex than a small, self-contained logic and make it harder to reason about completeness. The decomposition of structural permissions actually reveals hidden existential quantifiers, which is correctly accounted for in normalization. The same goes for the unfolding of data types with one branch. In a sense, we know from work in proof search techniques that these *need to* be performed if we want to be able to perform the proper instantiations.

We have no formal argument to offer, however, that could possibly justify why there are no other transformations that normalization has to perform, or why we haven't forgotten an extra rule in subtraction. In other words, a completeness result in the general case seems hard to attain. Besides, the syntactic rules for subsumption are proved correct with regard to the semantic notion of subtyping (from [BPP14a]); we do not know, however, that the syntactic subsumption rules are correct with regards to the semantic subtyping! We are thus left with our experience with the implementation which has been, in general, satisfactory.

Entailment in separation logic

Subtraction, when understood in the spirit of separation logic, performs two tasks at once: frame inference and entailment. Entailment, usually written $P \Vdash Q$, is a decision procedure for determining, whether in every memory configuration that satisfies P , then Q also holds.

Entailment in full separation logic is undecidable [Rey02]; in a sublanguage without quantifiers, it is, however, decidable [CYO01]. (This should not be taken as a guarantee that subtraction is undecidable in Mezzo: in particular, the fact that our sum types are tagged makes things much easier than with built-in list segment predicates, as an entailment procedure for these has to explore both cases of the anonymous sum.)

An early procedure for entailment in separation logic [BCO04] was proposed by Berdine *et al.* The authors only consider a small language, equipped with equalities and disequalities, along with points-to assertions and built-in ls (list segment) predicates. No quantifiers or user-defined predicates occur. They provide a proof system along with a decision procedure.

This proof system was implemented in the Smallfoot tool [BCO05a]; in the worst case, the tool performs an exponential exploration. It was not until 2011, though, that a polynomial-time algorithm was found for the same fragment [CHO⁺11].

This latter piece of work uses a graph-based reasoning, and sees the problem of entailment as a restriction of the (generally NP-complete) *graph homomorphism* problem. SeLogger [HIOP13] also adopts a graph-based approach and encodes the entailment problem as a graph homomorphism problem. The authors claim substantial improvements compared to SLP, another tool by Rybalchenko and Pérez.

SLP [NR11] offers an algorithm for deciding entailment in separation logic. The spatial formulas in SLP rely on the lseg (list segment) predicate. According to the authors, SLP provides a substantial gain in performance compared to Smallfoot. However, according to Haase *et al.*, SeLogger is also several orders of magnitude faster than SLP.

The same authors, in a later paper, use an SMT solver to decide entailment in separation logic [PR13]. This allows leveraging off-the-shelf tools; this also allows the logical formulas to use any theory supported by the solver. The only spatial assertion remains the list segment.

The interaction between general-purpose, first-order SMT solvers and higher-order logics such as separation logic is an active research area. Kassios *et al.* study the encoding of abstract predicates into Boogie [HLMS13]. A subsequent paper [JKM⁺14] describes a specialized intermediate language better suited for these logics.

Another related issue in separation logic is determining whether an assertion is satisfiable, that is, if there exists a memory configuration where the assertion is valid. This is the *satisfiability* issue. Satisfiability reduces to entailment, in that P unsatisfiable $\Leftrightarrow P \Vdash \text{false}$ (that is, P satisfiable $\Leftrightarrow \neg(P \Vdash \text{false})$). If an assertion corresponding to a program point is not satisfiable, it means that the code is unreachable. This can have important consequences on the remainder of the analysis. Decision procedures have been proposed; more recently, in the general case where one may have arbitrary user-defined predicates, and not just list segments [BFGP13]. The equivalent Mezzo concept would be an inconsistent conjunction; as I mentioned earlier, we hardly ever leverage this information.

Being able to account for user-defined inductive predicates is an improvement over just having the built-in list segment predicate. Nguyen *et al.* [NDQC07] offer a sound procedure for entailment in the presence of user-defined inductive predicates. No completeness result is offered, but termination is guaranteed.

11.7 A glance at the implementation

Where are the algorithms?

The `restrict` function is implemented in `typing/TypeCore.ml`; it is called `import_flex_instantiations_raw`. In `typing/Permissions.ml`, `import_flex_instantiations` performs proper renormalization of permissions after prop-

agating instantiation choices.

The normalization procedure that is used extensively in this section is, quite unfortunately, spread out over several modules.

- In `typing/Types.ml`, the `expand_if_one_branch` function implements normalization rule (9) (Figure 9.5).
- In `typing/Types.ml`, the `collect` function implements normalization rule (1) using a recursive traversal.
- In `typing/TypeCore.ml`, the `modulo_flex` function implements normalization rule (10); this function needs to be called every time someone matches on a type, quite unfortunately.
- In `typing/Permissions.ml`, `open_all_rigid_in` implements normalization rules (2a), (2b), (3) or (6a), (6b) (6c) and (7), depending on its side parameter. The function does not explicitly rewrite the type by floating out quantifiers, but opens them directly as rigid bindings.

Addition is implemented in `typing/Permissions.ml` by the `add_perm` and `add_type` functions. The former function first flattens the conjunction. Then, for each permission, it either:

- treats it immediately (equations);
- stores it in a separate list (abstract permission application);
- sends it off to `add_type` (anchored permission).

The `add_type` function uses all the functions above, then applies renormalization rules (4a), (4b) and (5) on-the-fly.

Taking the duplicable restriction \bar{P} of a permission P is implemented by the `keep_only_duplicable` function in the `Permissions` module.

The subtraction interface

The `Permissions` module in the type-checker implement the rules of subtraction via a set of mutually recursive functions `sub`, `sub_type` and `sub_perm`.

```
type result = ((env * derivation), derivation) either
val sub: env -> var -> typ -> result
val sub_type: env -> typ -> typ -> result
val sub_perm: env -> typ -> result
```

The first function takes an environment along with a variable x and a type t , and subtracts $x @ t$ from the “environment”. The `env` type is the one I described earlier in §9.7. The second function takes two types and performs a focused subtraction, that is, it only compares types at kind type, without the $x @ \dots$ part. The last function takes a permission, that is, a type at kind `perm`.

Results are either a success, meaning that the user gets another `env`, along with a (currently unused) successful derivation. In the event of a failure, the user gets a failed derivation that is suitable for debugging, and for isolating a potentially short or useful error message.

A design mistake The interface of the `sub_type` function is actually a design mistake: this early error makes it impossible to type-check some valid programs.

From a high-level perspective, the problem is as follows. When focusing on a subtraction $x @ t - x @ u$, we move on to a subtraction $t - u$ where the x parameter has been dropped. This is morally equivalent to $\mathcal{R}(z : \text{value}).z @ t - z @ u$, which is a strictly stronger goal than $\mathcal{R}(x : \text{value}).x @ t - x @ u$ in the case that $x \in t$ or $x \in u$. This leads to an incompleteness.

Figure 11.8 presents a fragment of valid Mezzo that the type-checker erroneously rejects for this precise reason. The code uses an advanced feature: the fact that one can name a branch of a data type using a name introduction construct. The `pred` type is artificial; it only allows me to defeat the syntactic check I mentioned earlier and make sure the subtraction fails in the way I describe below.

The type-checker calls “`sub env x t`” (`env` was described in §9.7); next, `sub` fetches the list of permissions for x , select B , then calls “`sub_type env B t`”. As usual, the type-checker strengthens the goal, and faces (sub-goal for `pred` omitted):

$$B - \exists(\text{self} : \text{value}) (= \text{self} \mid \text{self} @ B)$$

```

abstract pred (x: value): perm

data t =
  | A
  | (self: B | pred self)

val f (x: B | pred x): () =
  assert x @ t

```

Figure 11.8: A valid Mezzo program that can't be type-checked

The type-checker opens the flexible variable:

$$\mathcal{F}(\text{self} : \text{value}).B - (\text{self} \mid \text{self} @ B)$$

Lacking any other extra information, the type-checker is at a loss for picking a suitable value for `self`: it has lost the information that `self` should instantiate onto `x`. The algorithm “rigidifies” `self`, and fails.

This source of incompleteness does *not* lie in the set of rules presented in this chapter; rather, it is a consequence of the design I adopted in my code. A future version of the type-checker should probably remove the $t_1 - t_2$ operation and only offer an operation for $x @ t_1 - x @ t_2$ which remembers that the left-hand side of the anchored permission is `x`.

The derivation library

Sample code Here is an extract from the Permissions module; it focuses on the subtraction of a type/mode conjunction of the form $(m X \mid t)$.

```

let sub_type env t1 t2 =
  ...
  match t1, t2 with ->
  ...
  | _, TyAnd (c, t'2) ->
    try_proof env (JSubType (t1, t2)) "And-R" begin
      sub_type env t1 t'2 >>= fun env ->
        sub_constraint env c >>=
          qed
    end
end

```

This code sample is fairly representative of the coding style that prevails in the core of the type-checker. The code is written in a monadic style, using a library of combinators for building typing derivations. The call to `try_proof` takes an environment, the judgement that we are attempting to prove, the name of the rule that we are applying, along with the premises of the rule. It produces a `result`. In this case, there are two premises: one for the mode constraint, one for the type itself. We chain them using the bind (`>>=`) operator, and terminate the proof by writing `qed`. The particular order of operations matters: the code first subtracts $t_1 - t'2$, *then* checks that the mode constraint is satisfied. Consider $\mathcal{F}(a).\text{int} - (a \mid \text{duplicable } a)$. We first need to make sure `a` is instantiated onto `int` before checking that `duplicable a` holds. Picking the converse order of operations would lead to a failure, as the type-checker is unable to deal with `duplicable a` while `a` is flexible.

This monadic style allows for a close correspondence between the code and the type-checking rules. While it does not provide the same correctness guarantees as a completely certified type-checker, it nonetheless allows for easier manual inspection of the code. Several bugs were found when switching the core of the type-checker to this style. Also, the combinators take care of building a derivation automatically, which is then helpful for debugging.

Building derivations The interface for derivations is shown in Figure 11.7. The interface is quite brittle and deserves further changes. In particular, there is some redundancy between the various types that deserves further

```

1  (* Good: a snapshot of the [env] at the time the [rule] was applied to prove
2  *   the [judgement]
3  * Bad: a snapshot of the [env] at the time we tried to prove the
4  *   [judgement], along with all the [rule]s that failed *)
5  type derivation =
6  | Good of env * judgement * rule
7  | Bad of env * judgement * rule list
8
9  and rule =
10 rule_instance * derivation list
11
12 and rule_instance = string
13
14 and judgement =
15 | JSubType of typ * typ
16 | JSubPerm of typ
17 | ...
18
19 (* Either all [Left] (a set of successful derivations) or a single [Right]
20 * (a single bad derivation with all the failed rules). *)
21 type result = ((env * derivation, derivation) either) LazyList.t
22
23 val no_proof : env -> judgement -> result
24
25 val apply_axiom :
26   env -> judgement -> rule_instance -> env -> result
27
28 val par : env -> judgement -> rule_instance -> result list -> result
29
30 (* The [state] type: chaining the premises of a rule. *)
31 type state
32
33 val try_proof : env -> judgement -> rule_instance -> state -> result
34
35 val ( >>= ) : result -> (env -> state) -> state
36
37 val qed : env -> state
38
39 val fail : state

```

Figure 11.9: The interface of the Derivations module

exploration. The general feeling is that the interface strives to express a combination of the backtracking computation monad, the failure monad, and the state monad.

The application of a typing rule is defined by the `rule` type (line 9), that is, a pair of the rule name (line 12) and a list of derivations, which stand for the premises of the rule. This models either a successful rule application (all the premises are Good derivations) or a failed rule application (the first few premises are Good, then we hit a Bad premise).

A derivation itself is either good or bad (line 5). A good derivation captures the `env` where it took place and the judgement it proves, along with the typing rule one used to prove the judgement. A bad derivation lists *all* the rules that the type-checker tried, and that failed.

Since we explore the solution space, the `result` type is a stream, that is, a lazy list. There are two cases (which are redundant with the definition of `derivation` – again, this deserves further simplifications).

- i) The stream is exclusively made up of `Left (env, Good _)` elements. This means that at least one solution exists to this subtraction problem. Each `Left` element holds the final environment `env` after the subtraction took place, along with a `Good` derivation. The derivation may be used for debugging purposes.
- ii) The stream is made up of one single `Right (Bad _)` element. Instead of representing failure via an empty stream, we wish to provide the user with a meaningful error message. Building up an error messages requires examining the derivation (not covered in this thesis) to identify possibly meaningful points of failure.



The module that performs subtractions (§10.9) hides the backtracking aspect and does not expose the lazy list representation; rather, it returns a single element from the list (the “best” one in case of success, the failed one otherwise).

Several rules may apply for proving a judgement. The judgement type (line 14) lists two representative cases: subtracting a permission from an environment (`JSubPerm`) and subtracting a type from another type (`JSubType`). For instance, when trying to prove $P - x @ t'$ (which is a `JSubPerm` judgement), one may try several of the permissions available for x in P , which all lead to a different rule being applied to try to prove the `JSubType` judgement. In the case that all of these rules fail, we obtain a `Bad` derivation listing all the failed rule applications.

The library provides combinators for constructing lazy lists of solutions. The simplest one is `no_proof` (line 23): it returns a `result` which corresponds to a judgement for which we have no rule. It is a lazy list made up of a single `Right (Bad _)` element. Symmetrically, `apply_axiom` (line 25) builds a `result` for a judgement we can prove without premises, that is, an axiom: we only need to state the name of the axiom. It is a lazy list made up of a single `Left (_, Good _)` element.

More sophisticated combinators are available: `par` (line 28) is useful in case there are several branches to explore for proving the same judgement. The function combines the list of streams into a single lazy stream of results. This means that if either one of the branches contains a `Left (_, Good _)` derivation, the resulting stream will return `Left (_, Good _)` derivations. Conversely, if all the items in the streams are `Right (Bad _)` derivations, the result is a single `Right (Bad _)` derivation that records *all* of the failed derivations. This is the invariant for the `result` type I mentioned earlier.

These combinators allow combining results just fine; however, we only saw how to construct derivations for *axioms*, that is, rules that take no premises. Building a derivation with premises is done using the `state` type, which helps chaining the premises of the rule we are trying to prove.

Once the premises are properly combined, the `try_proof` (line 33) primitive combines the premises with a rule so as to produce a `result`, which is `Left (_, Good _)` if all the premises are successful, or `Right (Bad _)` if a premise failed.

The successful state that holds no premises is `qed` (line 37). As a side note, it turns out that `apply_axiom env j r sub_env` really is an alias for `try_proof env j r (qed sub_env)`. The failed state that holds no premises is `fail` (line 39).

The `bind` operator (`>>=`) (line 35) allows one to manually thread premises. It takes a premise, that is, a `result`, and a function that computes the subsequent premises. If the `result` fails, the computation of the subsequent premises is skipped, as the rule application is a failure anyway. If the result is successful, the subsequent premises are computed, and the operator returns the list of all premises for the rule. When chaining premises, one must always finish with `qed` or `fail`.

11.8 Future work

The main concern with the current implementation is its reliance on a set of heuristics. These heuristics have been written with no clear intent in mind; rather, they have been fined-tuned as we came up with examples that ought to be type-checked but failed to be accepted.

Ideas for improvements include a better representation of a subtraction operation, which I briefly mentioned in the preceding section. Other ideas include a notion of polarity of flexible variables: typically, a variable in positive position will want to be “as small” as possible, while a variable in negative position will want to be “as big” as possible. We could tag flexible variables with the polarity they originated from, and make instantiation decisions accordingly.

12. The merge operation

When type-checking examples, such as that of the `map` function (Figure 11.1), our `match` statements were in terminal position, meaning that they benefited from the top-down propagation of the expected type (covered in §10.8). Checking that the function was well-typed amounted to checking that each branch was, individually, well-typed.

Not all `match` statements are in tail position, though. The union-find implementation from Figure 4.3, at line 50, uses a `match` statement in the middle of a sequence, meaning that one has to infer the post-condition (that is, the permission) after the `match` expression. The post-condition for the `match` expression must be a supertype of the two individual post-conditions from each branch: as I mentioned earlier, this is the *merge problem*. Naturally, empty is a supertype of all permissions; the difficulty lies in finding a “good” (small, according to \leq) permission.

The type-checking rules, such as `MATCH`, assume we “magically” know which t we should seek to obtain. The algorithmic specification of type-checking, though, assumes a procedure for computing a *disjunction* of permission, written:

$$\mathcal{V}_0[\text{ret}] \vdash \mathcal{V}_l.P_l \vee \mathcal{V}_r.P_r = \mathcal{V}_d.P_d$$

Performing this computation is the topic of the present chapter. I start off with a brief overview of the problem: through a set of examples, I illustrate the variety of difficulties that arise when tackling this issue. Next, I formalize the merge operation, by expressing it as a set of rewriting rules along with the application of subsumption rules. Finally, I present an algorithmic specification and discuss implementation issues.

12.1 Illustrating the merge problem

The merge problem arises when type-checking two constructs: `if-then-else` and `match`-expressions. The two are, in essence, the same thing, except that the former has two branches while the latter has an arbitrary number of branches.

A series of examples

Let me offer a few examples of `if-then-else` expressions that produce interesting cases and highlight the main difficulties related to the merge operation. The curious reader may want to launch `mezzo -html-errors -explain html:` with these special flags, the extra `explain` keyword instructs the type-checker to generate an HTML, clickable explanation of these merge errors. This dissertation only reproduces the static graphs, where affine permissions involved in assignment conflicts are in red.

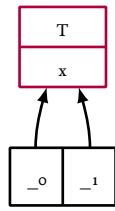
The code samples use `true` as a boolean expression: naturally, in the general case, the value of the boolean expression is not statically known.

First example Figure 12.1 presents the code snippet along with a graphical interpretation of what happens in the `then` branch and the `else` branch. In the `then` branch, the components of the tuple point to the *same*, uniquely-

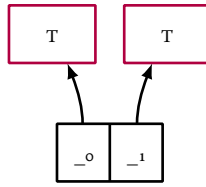
```
data mutable t = T
```

```
val z =
  if explain true then begin
    let x = T in
      (x, x)
  end else begin
    (T, T)
  end
```

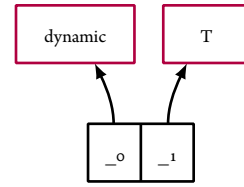
Code snippet



Then-branch



Else-branch



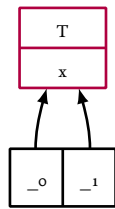
Possible result

Figure 12.1: First merge example

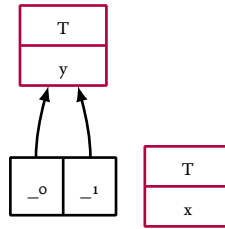
```
data mutable t = T
```

```
val z =
  let x = T in
  if explain true then begin
    (x, x)
  end else begin
    let y = T in
      (y, y)
    end
```

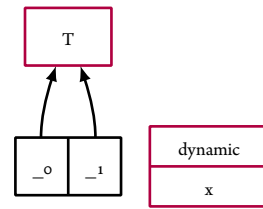
Code snippet



Then-branch



Else-branch

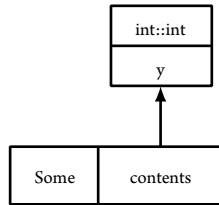


Possible result

Figure 12.2: Second merge example

```
val z =
  let y = 1 in
  if explain true then
    Some { contents = y }
  else
    None
```

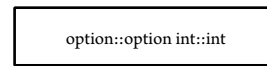
Code snippet



Then-branch



Else-branch



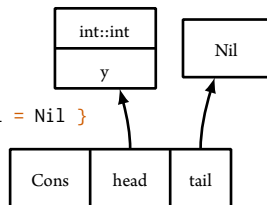
Possible result

Figure 12.3: Third merge example

```
open list
```

```
val z =
  let y = 1 in
  if explain true then
    Cons { head = y; tail = Nil }
  else
    Nil
```

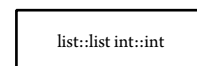
Code snippet



Then-branch



Else-branch



Possible result

Figure 12.4: Fourth merge example

owned T . In the `else` branch, the components of the tuple each point to a distinct T .

Solving this merge problem means assigning a tuple type to z . However, neither of the two descriptions subsumes the other one: one cannot pick $z @ (T, T)$ (this would be a breach of soundness should execution take the `then` branch), and one cannot pick $z @ (=x, =x) * x @ T$ (this would be a breach of soundness should execution take the `else` branch).

This merge problem can be solved by picking either “ $z @ (T, \text{dynamic})$ ” or “ $z @ (\text{dynamic}, T)$ ”. Lacking any other annotation, the type-checker has no way of deciding which solution should be preferred over the other, as they are incomparable. (If the type t were duplicable, “ $z @ (T, T)$ ” would be a principal solution.)

Second example In Figure 12.2, sharing occurs, but with a different scope depending on the branch.

This time, the type-checker can pick several solutions:

- picking “ $\{u\} (x @ T * z @ (=u, =u))$ ” preserves the permission for x while still retaining the fact the two components of the tuple are pointers to the same block in the heap (u is existentially quantified);
- picking “ $\{u\} x @ \text{dynamic} * z @ (=u, =u) * u @ T$ ” loses the permission for x in favor of a “better” one for z ;
- other, inferior solutions exist, such as “ $z @ (T, \text{dynamic})$ ” or “ $z @ (\text{dynamic}, T)$ ”, which would fail to preserve sharing information.

The first two solutions are incomparable, as none of them implies the other. The last two solutions, however, are derivable from the second one. Again, if t were to be duplicable, this situation would admit a principal solution, namely “ $x @ T * z @ (=u, =u) * u @ T$ ”.

Third example Merging is even more difficult when nominal types come into play (Figure 12.3).

Let us consider the case where y refers to duplicable data, and assume for the sake of example that we have “ $y @ \text{int}$ ”. This merge problem admits a principal solution, namely “ $z @ \text{option} (=y) * y @ \text{int}$ ”. One can derive “ $z @ \text{option} \text{int}$ ” using `COPYDUP`.

In the case that y refers to non-duplicable data, e.g. if we have “ $y @ \text{ref int}$ ”, then again, this problem does not admit a principal solution, as “ $z @ \text{option} (=y) * y @ \text{ref int}$ ” and “ $z @ \text{option} (\text{ref int})$ ” are not comparable¹.

Fourth example Lists pose similar difficulties (Figure 12.4).

Here, merging into “ $z @ \text{list} (=y) * y @ \dots$ ” is alright, because one can regain $z @ \text{list int}$ using `COPYDUP`. If we had $y @ \text{ref int}$ instead, merging into “ $z @ \text{list} (=y) * y @ \dots$ ” would be a terrible choice, as the user could only append y to the list afterwards!

Three key difficulties

Computing the merge of two permissions can be divided into three sub-problems. While the actual implementation does not solve the three sub-problems in isolation, merging is better understood as the combination of the following.

Graph traversal A permission, thanks to singleton types which encode points-to relationships, can be understood as a graph. The act of merging two permissions can be understood as reconciling two heap shapes, that is two graphs, into a common structure that subsumes the two. We do this by traversing the two graphs in parallel and computing a *mapping* onto the destination graph. This is similar to shape analysis [Riv11]. This sub-problem is treated efficiently and predictably using a *maximal sharing* semantics by our algorithm.

Subsumption As we saw, reconciling a `Nil` branch and a `Cons` branch requires applying subsumption rules on each side. Folding inductive predicates is thus a key aspect of the merge problem, which is highlighted by the third and fourth examples. What is specific to *Mezzo*, though, is that folding a concrete type into a nominal type requires one to *infer* the parameters of the type application. If the parameter is covariant, this amounts to recursively merging the parameters; in the general case, though, we need to rely on subtraction. We saw

¹ The system would need a rule for an η -expansion of data types. While we could in theory add this, we currently have no plans to add the feature. Besides, it would not help in the fourth example with lists.

in the previous chapter that inference often yields multiple solutions, and our algorithm applies heuristics. Hence, it is not complete.

Ownership Because of mutable data, the algorithm has to decide, as in the first example, where to “assign” a piece of mutable data. The algorithm can detect and warn the user about such cases, but picks an arbitrary solution. The first two examples would trigger such a warning. This problem is, in practice, of surprisingly small importance: for one thing, it rarely occurs in real-world code; besides, a natural method for detecting and warning about these situations appears near the end of the present chapter, meaning I don’t devote any special effort to this sub-problem. One could conceivably imagine more general approaches based on linear constraints, which would encode the fact that either one of two permissions is available but not both at the same time. The need for such advanced mechanisms has not surfaced yet in our usage of Mezzo.

Why insist on solving disjunctions?

A natural solution that comes to mind is simply to keep a symbolic disjunction and go on type-checking the rest of the code. That is, we would add a \vee connective to the grammar of permissions, and manipulate $\mathcal{V}_l.P_l \vee \mathcal{V}_r.P_r$. Any further type-checking operation would then be performed on each term of the disjunction, and upon reaching the end of a function body, we would merely assert that the post-condition is satisfied for each term of the disjunction.

This solution is not satisfactory for several reasons.

Combinatorial explosion If the function body that we’re type-checking contains several sequential if-then-else expressions, then the number of terms in the disjunction grows very fast.

Control-flow sensitivity Upon reaching the end of the control flow (end of a function body), some terms of the disjunction may satisfy the post-condition, while some others may not. Error messages would then need to mention which branches exactly the code has to take in order to trigger the error.

Predictability Some mistakes may only pop up very late in the type-checking process: if we were to keep disjunctions, then the user may very well return an integer in one branch, and a list in another branch, without hitting any difficulties until the very end. We want to catch these kinds of errors early.

Another mechanism we may want to leverage is a *backwards analysis*. One could, for instance, mark variables which are no longer used. In the third example, if y is no longer used afterwards, it makes no sense to preserve a permission for it. Conversely, if y is used afterwards, we may want to keep the permission for it. We have not explored this opportunity, since the set of heuristics we currently possess has proven satisfactory. We hope that more experiments with the language, especially with large imperative codebases, will tell whether a more sophisticated analysis is needed or not.

A last, unsatisfactory approach would be to require program authors to annotate with post-conditions every disjunction in the control-flow. It turns out this is already feasible. Consider the following snippet.

```
val z: t =
  if ... then
    e1
  else
    e2
```

Recall that type annotations in Mezzo, as I discussed already, only mention *part* of the current permission, not the entire set of permissions that the programmer expects to possess at this program point. The example above is type-checked as follows. The type-checker *first* subtracts $\text{ret } @ t$ from each branch, obtaining two remainders. The remainders are merged, thus giving a prefixed permission $\mathcal{V}.P$. The final permission is thus $\mathcal{V}.P * z @ t$. Worded differently, *at least* $z @ t$ is available; if other remaining permissions can be merged, they also become available.

We could thus get rid of the merge operation entirely if we asked program authors to first, annotate every disjunction and, second, to annotate them with the *entire set of permissions* that they need for the remainder of the current function body. This is, quite obviously, unrealistic, as programmers would need to mention all the variables that they use afterwards, all the functions from other modules that they intend to use, etc. We therefore need to devise a way to perform this merge operation.

12.2 Formalizing the merge operation

The merge operation computes the disjunction of several *fully normalized*, prefixed permissions $\mathcal{V}_i.P_i$. Each P_i is the result of type-checking the given branch using the type-checking algorithm from Chapter 10. The distinguished program variable `ret` was passed to the type-checking algorithm, meaning that the name of the return value in each branch is `ret`. This distinguished variable also stands for the return value of the whole disjunction, in P_d . The merge operation also needs to know the set of “old” variables that were in scope before the disjunction; the operation thus needs the prefix \mathcal{V}_0 that was used before type-checking each branch. The original permission P_0 is of no particular interest.

The merge operation is hence written as follows, where `ret` is assumed to be rigidly bound in \mathcal{V}_0 :

$$\mathcal{V}_0[\text{ret}] \vdash \mathcal{V}_1.P_1 \vee \dots \vee \mathcal{V}_n.P_n = \mathcal{V}_d.P_d$$

As mentioned earlier, the merge operation satisfies the following properties:

- $\mathcal{V}_d \leq \mathcal{V}_0$, that is, it refines the original prefix: intuitively, the merge operation may make some instantiation decisions to provide better results;
- $\mathcal{V}_l \times P_l \leq \mathcal{V}_d \times P_d$, that is, the destination permission is a supertype of the left permission;
- $\mathcal{V}_r \times P_r \leq \mathcal{V}_d \times P_d$, that is, the destination permission also is a supertype of the right permission, that is, it subsumes both.

An important pre-condition for the merge operation is that all variables in \mathcal{V}_l and \mathcal{V}_r be rigidly bound. I discuss this restriction in §12.4 as well as how to work around it in practice.

In the rest of this chapter, I only focus on binary disjunctions. We have no special treatment for n -ary disjunctions, and solve the general case by recursively merging conjunctions pairwise; we have no associativity or commutativity results for this n -ary merge operation.

Notations used in this chapter

Four permissions The permission available at the program point immediately before the disjunction is called the *original* permission and is written $\mathcal{V}_0.P_0$. The two branches in the control-flow are called the *left and right* permissions, noted $\mathcal{V}_l.P_l$ and $\mathcal{V}_r.P_r$ respectively. The result of the merge operation is the *destination* permission $\mathcal{V}_d.P_d$.

Roots We call any variable $(X : \kappa) \in \mathcal{V}_0$ a *root*; the special `ret` variable that stands for the name of the match expression is thus also a root. In other words, a *root* is a variable bound before the disjunction.

Local and old variables We say that any variable defined only in \mathcal{V}_l or \mathcal{V}_r is *local*. Variables defined in \mathcal{V}_0 (including `ret`) are said to be *old*.

Correctness of merging

The correctness condition of merging is:

$$\begin{cases} \mathcal{V}_l \times P_l \leq \mathcal{V}_d \times P_d \\ \mathcal{V}_r \times P_r \leq \mathcal{V}_d \times P_d \end{cases}$$

Additionally, merging also ensures that $\mathcal{V}_d \leq \mathcal{V}_0$, that is, merging refines the original prefix. There is no particular result we can offer that relates \mathcal{V}_l or \mathcal{V}_r to \mathcal{V}_0 : indeed, \mathcal{V}_l may contain extra *rigid* variables that were introduced in the course of type-checking the corresponding branches.

Our specification satisfies this correctness condition; it requires us, however, to talk about existential packing for rigid variables that only appear in \mathcal{V}_l or \mathcal{V}_r , as well as renaming issues that allow transforming the left and right permissions into the destination permission.

A formal merge example

The example below illustrates the rewriting steps and the explicit existential packings needed to show that the operation is correct. The example only involves duplicable data and two distinct heap shapes. It is similar to the first example in this chapter, except that it involves the duplicable type `int` so as to simplify the discussion.

$$\begin{array}{l} \mathcal{R}(\text{ret}, y_l : \text{value}). \quad \text{ret } @ (=y_l, =y_l) * y_l @ \text{int} \\ \vee \\ \mathcal{R}(\text{ret}, y_r, z_r : \text{value}). \text{ret } @ (=y_r, =z_r) * y_r @ \text{int} * z_r @ \text{int} \end{array}$$

These two permissions describe two different heap shapes. The left one describes a pair whose two components are the same integer, while the right one describes a pair whose two components are distinct integers. A shape description that subsumes both is the following one:

$$\mathcal{R}(\text{ret}, y_d, z_d : \text{value}). \text{ret } @ (=y_d, =z_d) * y_d @ \text{int} * z_d @ \text{int}$$

We do not wish, however, to introduce new rigid variables in the destination prefix; indeed, we wish to guarantee $\mathcal{V}_d \leq \mathcal{V}_0, \mathcal{R}(\text{ret} : \text{value})$. We therefore wish to return the following permission, where existential quantification over local variables from \mathcal{V}_l and \mathcal{V}_r allows for agreement.

$$\mathcal{R}(\text{ret} : \text{value}). \exists (y_d, z_d : \text{value}) \text{ret } @ (=y_d, =z_d) * y_d @ \text{int} * z_d @ \text{int}$$

The right permission can be trivially converted into the destination permission by packing all occurrences of y_r and z_r into y_d and z_d , respectively, and dropping the now-unused y_r and z_r from the prefix. We need to justify, however, why the left prefixed permission can be transformed into the destination prefixed permission. We start with the original permission for the left expression.

$$\mathcal{R}(\text{ret}, y_l : \text{value}). \text{ret } @ (=y_l, =y_l) * y_l @ \text{int}$$

The permission $y_l @ \text{int}$ is duplicable: we can thus create a copy of it.

$$\dots \leq \mathcal{R}(\text{ret}, y_l : \text{value}). \text{ret } @ (=y_l, =y_l) * y_l @ \text{int} * y_l @ \text{int}$$

We perform two existential packings. We pack occurrences of y_l that refer to the left component of the tuple as y_d ; we pack occurrences of y_l that refer to the right component of the tuple as z_d .

$$\dots \leq \mathcal{R}(\text{ret}, y_l : \text{value}). \exists (y_d, z_d : \text{value}). \text{ret } @ (=y_d, =z_d) * y_d @ \text{int} * z_d @ \text{int}$$

Finally, we drop the unused quantifier and obtain the desired permission

$$\dots \leq \mathcal{R}(\text{ret} : \text{value}). \exists (y_d, z_d : \text{value}). \text{ret } @ (=y_d, =z_d) * y_d @ \text{int} * z_d @ \text{int}$$

12.3 An algorithmic specification for merging

Overview of the algorithm

The algorithm can be divided in two sub-tasks. Each sub-task is covered in a separate subsection.

Building the mapping A key issue that the example above highlights is that we need to match variables together. That is, we need to compute which variable from the left “corresponds” to which variable from the right. This is the *mapping problem*, and corresponds to the first task for our algorithm. This is performed using a graph traversal, which is one of the aspects of the problem I alluded to earlier (§12.1).

Taking the example above, exploring the two permission graphs from `ret` leads us onto a tuple type on both sides, with matching lengths. Seeing the singleton types for the first component of the tuple, we infer y_l and y_r map onto y_d . Finally, examining the second component of the tuple, we conclude that y_l and z_r map onto z_d .

The mapping is a technical tool that allows us to build the destination permission.

The mapping directly gives the list of variables that need to be existentially-quantified. For instance, in the example above, we know from the mapping that the destination permission will be quantified using $\exists (y_d, z_d)$.

The algorithm thus does not perform α -conversions: it leaves the old variables untouched, and existentially packs local variables so as to obtain matching permissions on both sides.

$$\begin{array}{c}
\text{MAPPING-ANCHORED} \\
\frac{(x_l, x_r) \in \text{domain}(\psi) \quad \psi + t_l \vee t_r = \psi'}{\psi + x_l @ t_l \vee x_r @ t_r = \psi'} \\
\\
\text{MAPPING-TUPLE} \\
\frac{\psi_i + t_{l,i} \vee t_{r,i} = \psi_{i+1}}{\psi_0 + (\vec{t}_l) \vee (\vec{t}_r) = \psi_n} \\
\\
\text{MAPPING-CONCRETE} \\
\frac{\psi_i + t_{l,i} \vee t_{r,i} = \psi_{i+1}}{\psi_0 + A \{\vec{f}: \vec{t}_l\} \vee A \{\vec{f}: \vec{t}_r\} = \psi_n} \\
\\
\text{MAPPING-SINGLETON} \\
\frac{\psi + x_l \vee y_l = \psi'}{\psi + =x_l \vee =y_l = \psi'} \\
\\
\text{MAPPING-APP} \\
\frac{t \text{ is old} \quad \psi' + u_{l,i} \vee u_{r,i} = \psi'' \quad \text{variance}(t, i) = \text{co}}{\psi + t \vec{u}_l \vee t \vec{u}_r = \psi''} \\
\\
\text{MAPPING-VAR-NEW} \\
\frac{(X_l, X_r) \notin \text{domain}(\psi) \quad X_d \text{ fresh}}{\psi + X_l \vee X_r = \psi[(X_l, X_r) \mapsto X_d]} \\
\\
\text{MAPPING-VAR-EXISTS} \\
\frac{(X_l, X_r) \in \text{domain}(\psi)}{\psi + X_l \vee X_r = \psi} \\
\\
\text{MAPPING-STAR} \\
\frac{\psi + P \vee P' = \psi' \quad \psi' + Q \vee Q' = \psi''}{\psi + P * Q \vee P' * Q' = \psi''}
\end{array}$$

Figure 12.5: Computing the ψ function

Building the permission Once the existential quantification is known, the last sub-task consists in comparing permissions pairwise; pairs of permissions are rewritten using the matching; they are hence well-scoped under \mathcal{V}_d and the existential quantifiers. The matching gives both the packing witness (say, y_l) and the name of the variable it should pack onto (in this case, y_d or z_d).

Indeed, two “compatible” types are merged into a destination permission. For instance, comparing $(=y_l, =y_l)$ with $(=y_r, =z_r)$, both can be rewritten using the mapping into $(=y_d, =z_d)$, meaning that the two types trivially match: this is a successful merge.

The reason why the two types above match is that we chose the prefix accordingly. After rewriting, the prefix should give as many opportunities for merging as possible.

While the example above is relatively simple, in the general case, the rules for merging types pairwise will need to solve the other two sub-problems mentioned earlier, that is, the rules for merging types pairwise will need to apply subsumption rules (the Nil vs. Cons case), and decide how to assign exclusive permissions.

Building the mapping

Building the mapping is the first sub-task; it is a prerequisite for constructing the series of existential quantifiers and the destination permission.

Roots and paths The mapping, per the earlier example, is a set of triples (x_l, x_r, x_d) . To define what it means for a triple to be in the mapping, I need to introduce the notion of a path.

We assume permissions to be fully normalized, meaning that there is at most one tuple or constructor type per variable, and that tuple components and constructor fields are made up of singleton types. This also means that type/permission conjunctions have been flattened out.

A path starts from a root (§12.2), and goes through tuple components, constructor fields and type applications, ultimately reaching a singleton type $=x$. We say that x is the end of the path. Formally, a path is a series of field names, tuple indexes, and indexes for type applications.

I write $x.\pi = y$, meaning that starting from root x , following path π ends on y .



Because of strong updates, existing paths from the original environment may be modified in each environment. The mapping must thus be computed for all variables, not just those that are reachable from ret . Consider the snippet below:

```

val _ =
  let x = () in
    let y = () in
      let z = Ref { contents = y } in

```



```

if true then
  z.contents := x
else
  ()

```

If only computing the mapping starting from *ret*, we get:

$$\psi = \{(x, x, x); (y, y, y); (z, z, z); (ret, ret, ret)\}$$

If computing the mapping from all the roots, we get:

$$\psi = \{(x, x, x); (y, y, y); (z, z, z); (ret, ret, ret); (x, y, w)\}$$

where w is a fresh, existentially-quantified variable.

The ψ relation We write $\psi(x_l, x_r, x_d)$ to mean that this triple is in the mapping. The meaning of the relation is as follows: if $\psi(x_l, x_r, x_d)$, then there exists a path π and a root ρ such that in P_l (resp. P_r, P_d), $P_l, \rho.\pi = x_d$.

$\psi(x_l, x_r, x_d)$ means that these variables are found at the end of the same path in the left, right, and destination environment; or, that they correspond to the same symbolic heap location.

In the earlier example, a path starts from the *ret* root, and goes through the first component of the tuple, thus yielding a new relation $\psi(y_l, y_r, y_d)$. Similarly, another path starts from the *ret* root, and goes through the second component of the tuple, thus yielding $\psi(y_l, z_r, z_d)$.

Singleton types provide a natural naming mechanism that helps us match locations in the heap between the left, right, and destination permissions. It is therefore crucial that one always finds singleton types in tuples and constructors: this is the reason why we require the left and right permissions to be fully normalized.

Making the relation a function Consider the following merge.

$$\begin{aligned}
& \mathcal{R}(\text{ret}, x_l : \text{value}).\text{ret} @ (=x_l, =x_l) \\
\vee & \\
& \mathcal{R}(\text{ret}, x_r : \text{value}).\text{ret} @ (=x_r, =x_r) \\
= & \\
& \mathcal{R}(\text{ret} : \text{value}), \exists(x_d, y_d : \text{value}).\text{ret} @ (=x_d, =y_d)
\end{aligned}$$

This solution is sub-optimal, as it loses sharing information. It is still correct, however, and relies on the following mapping:

$$\psi = \{(x_l, x_r, x_d); (x_l, x_r, y_d)\}$$

We thus need to make ψ a function, that is, make sure ψ only associates at most one value for each pair (x_l, x_r) . We will from there on maintain that invariant and write $\psi(x_l, x_r) = x_d$ to express the fact that there is only one such x_d .

This corresponds to a “maximal sharing” semantics: whenever the same variable is shared on both sides, this will be reflected in the destination environment.

Interpreting the mapping The process of building the mapping can be understood as synthesizing a rewriting of the variables at kind value. That is, having $\psi(x_l, x_r, x_d)$ can be understood as the following state of an in-progress rewriting of the disjunction:

$$\exists(x_d : \text{value}).((x_l = x_d * \dots) \vee (x_r = x_d * \dots))$$

The “maximal sharing” semantics above can be understood as another rewriting; the suboptimal solution above would correspond to the following working state:

$$\exists(x_d, y_d : \text{value}).((x_l = x_d * x_l = y_d \dots) \vee (x_r = x_d * x_r = y_d * \dots))$$

Having a maximal sharing semantics amounts to rewriting y_d into x_d , so as to obtain a single variable. This is logically correct, since each branch implies $x_d = y_d$, meaning that the disjunction implies $x_d = y_d$.

If one wishes to proceed concisely, one can interpret the mapping as a rewriting guide: if $\psi(x_l, x_r) = x_d$, then one can *rewrite* x_l (resp. x_r) into x_d by packing x_d using x_l (resp. x_r) as a witness.

Extending the mapping to other kinds Adopting this rewriting-packing view, we aim for a unified approach and rewrite not just variables at kind value, but also at kinds type and perm. We no longer have an interpretation in terms of equations, as this interpretation only makes sense as kind value; we keep, however, the rewriting interpretation based on existential packing.

An advantage of this unified approach is that the set of rules that I’m about to introduce can treat type variables in a uniform manner, regardless of their kind. The rewriting becomes the identity for old, rigid variables; for local variables, it rewrites a locally-bound type variable into one of the existentially-quantified variables.

Another advantage is that merging locally-abstract types at kind type allows for better results.

```
(* f @ () -> { t } t *)
if ... then
  f ()
else
  f ()
```

In the example above, we wish for the return permission to contain, among other things, $\mathcal{R}(\text{ret} : \text{value}), \exists(t : \text{type}). \text{ret} @ (t, t)$. Mapping t_l and t_r into an existentially-quantified t_d yields a better result than the top type.

Computing the function The rules for computing the ψ function are presented in Figure 12.5. Recall that these rules only compute the *mapping*, that is, they match heap locations, embodied as singleton types, together, so as to compute triples of variables that correspond to the same *path* in the left, right, and destination environments. In essence, this corresponds to building a set of existential quantifiers $\exists(x_d^i : \text{value})$, while keeping the disjunction “as is”. The next phase (“pairwise merging of types”) takes care of applying subsumption rules and moving permissions out of the disjunction into the destination environment.

I write:

$$\psi + x_l @ t_l \vee x_r @ t_r = \psi'$$

meaning that starting from ψ , matching t_l with t_r returns an extended relation ψ' . We start with the initial environment ψ_0 defined as follows, where $\vec{\rho}$ are the roots, that is, the rigid variables from \mathcal{V}_0 or the distinguished variable ret :

$$\psi_0 = \{(\vec{\rho}, \vec{\rho}, \vec{\rho})\}$$

This means that ψ_0 also contains triples for rigid variables at kind type, since they belong to \mathcal{V}_0 as well. For instance, ψ_0 will most likely contain, among other things, the triple $(\text{int}, \text{int}, \text{int})$. This allows for a uniform treatment of rigid variables in rule `MAPPING-VAR-EXISTS`, rather than adding a special-case for “old” variables.

The final ψ function is defined as:

$$\psi = \psi_0 + P_l \vee P_r$$

The traversal descends into conjunctions (`MAPPING-STAR`) until it finds matching anchored permissions (`MAPPING-ANCHORED`). This means that the traversal starts from the initial set of roots. All structural paths are explored (`MAPPING-TUPLE`, `MAPPING-CONCRETE`, `MAPPING-SINGLETON`). Whenever the traversal hits two variables, of any kind, it either adds a new entry in the mapping (`MAPPING-VAR-NEW`) if this pair of variables has not been visited already; otherwise, it does nothing (`MAPPING-VAR-EXISTS`). This ensures that we do indeed compute a function. We chose to descend into type applications (`MAPPING-APP`). This creates more opportunities for matching local program variables, or local abstract types (at kind type or perm).

These rules are non-deterministic: as we match more variables together via `MAPPING-VAR-NEW`, more opportunities for applying `MAPPING-ANCHORED` pop up, meaning that the process is iterated until all possible matchings have been collected.

The rule `MAPPING-STAR` is also taken to operate, as usual, modulo associativity and commutativity.

An incomplete set of rules Several constructs are omitted: there is no `MAPPING-FUNCTION` rule for instance; quantifiers are also not covered. This means that if a local variable appears under a quantifier, no additional triple will be added in the matching.

Deciding how far we are willing to go to match variables together is a design issue, since it only affects completeness, not soundness; we chose to remain relatively simple.

- Existential quantifiers are automatically unpacked due to normalization. The only way for the matching rules to encounter an existential would be to hit an existentially-quantified parameter of a type application that happens to mention a local variable to boot:

$$\text{pred } (\exists(t : \text{type}) x_l @ t)$$

This is rare enough to justify that we don't descend into existentials while building the mapping.

- Universal quantifiers are only ever used (in practice) along with function types. The user would thus have to write a disjunction that returns a function in both cases, with each function referring to a local variable. Anecdotal evidence suggests this is enough of an extreme case that we can safely omit it.

Mapping is existential packing I explained informally earlier that the mapping implicitly computed a rewriting of rigid variables using existential packing. The present section explains this in greater detail.

The earlier, intuitive claim was that having $\psi(x_l, x_r) = x_d$ allowed one to rewrite x_l (resp. x_r) into x_d . Let us see why this is sound.

An important pre-condition for the merge operation (mentioned early in §12.2) is that all variables in \mathcal{V}_l and \mathcal{V}_r be *rigid*. This means that $\mathcal{R}(\vec{X}_l : \vec{\kappa}).P_l$ is equivalent to $\exists(\vec{X}_l : \vec{\kappa}).P_l$. We are thus computing the following disjunction:

$$(\exists(\vec{X}_l : \vec{\kappa}).P_l) \vee (\exists(\vec{X}_r : \vec{\kappa}).P_r)$$

One can then further existentially-pack *using the mapping*. If $\psi(x_l, x_r) = x_d$, then one can pack *some* occurrences of x_l as x_d , and *some* occurrences of x_r as x_d , hence obtaining:

$$(\exists(\vec{X}_l : \vec{\kappa}, \vec{x}_d : \vec{\kappa}).P_l) \vee (\exists(\vec{X}_r : \vec{\kappa}, \vec{x}_d : \vec{\kappa}).P_r)$$

This is sound, since we are always packing onto x_d using a *unique witness*, that is, x_l or x_r . No two x_d can be the same, since x_d is always taken to be fresh in MAPPING-VAR-NEW.

The following section is dedicated to packing and applying subsumption rules in such a way that we obtain the following, where $X_l \# P_d$ and $X_r \# P_d$:

$$(\exists(\vec{X}_l : \vec{\kappa}, \vec{x}_d : \vec{\kappa}).P_d * P'_l) \vee (\exists(\vec{X}_r : \vec{\kappa}, \vec{x}_d : \vec{\kappa}).P_d * P'_r)$$

Existential quantifiers commute, meaning that we can obtain:

$$(\exists(\vec{x}_d : \vec{\kappa}, \vec{X}_l : \vec{\kappa}).P_d * P'_l) \vee (\exists(\vec{x}_d : \vec{\kappa}, \vec{X}_r : \vec{\kappa}).P_d * P'_r)$$

Going further, we get:

$$(\exists(\vec{x}_d : \vec{\kappa}).P_d * \exists(\vec{X}_l : \vec{\kappa}).P'_l) \vee (\exists(\vec{x}_d : \vec{\kappa}).P_d * \exists(\vec{X}_r : \vec{\kappa}).P'_r)$$

The final step consists in dropping $\exists(\vec{X}_l : \vec{\kappa}).P'_l$ and $\exists(\vec{X}_r : \vec{\kappa}).P'_r$: the two sides of the disjunction become syntactically equal, meaning the disjunction is trivially solved.

This means that we can state the following lemma:

Lemma 12.1. *Rewriting rigid variables in P_l and P_r according to ψ is a correct (well-scoped) operation.*

This relies on the discussion above, and in particular on the fact that there are only existential quantifiers, meaning that they commute. Again, the assumption that no flexible variables exist in \mathcal{V}_l and \mathcal{V}_r (that is, that there are no universal quantifiers in-between the existentials) is restrictive, but this is discussed later on.

The final result of a merge operation is thus:

$$\mathcal{V}_0[\text{ret}] \vdash \mathcal{V}_l.P_l \vee \mathcal{V}_r.P_r = \mathcal{V}_0.\exists(\vec{X}_d : \vec{\kappa}).P_d$$

where $\vec{X}_d : \vec{\kappa}$ describes all the variables in the image of ψ that are not old, and P_d has been computed using ψ -based rewritings and according to the yet-unknown procedure described in the next section.

(Again, subsequent sections discuss improvements related to flexible variable instantiations.)

$$\begin{array}{c}
\text{MERGE-DEFAULT} \\
\frac{\mathcal{V}_l \# t_l \quad \mathcal{V}_r \# t_r \quad t_l = t_r = t}{t_l \vee t_r = t} \\
\\
\text{MERGE-ANCHORED} \\
\frac{\psi(x_l, x_r) = x_d \quad t_l \vee t_r = t_d}{x_l @ t_l \vee x_r @ t_r = x_d @ t_d} \\
\\
\text{MERGE-STAR} \\
\frac{P_l \vee P_r = P_d \quad Q_l \vee Q_r = Q_d}{P_l * Q_l \vee P_r * Q_r = P_d * Q_d} \\
\\
\text{MERGE-VAR} \\
\frac{\psi(X_l, X_r) = X_d}{X_l \vee X_r = X_d} \\
\\
\text{MERGE-TUPLE} \\
\frac{\vec{t}_l \vee \vec{t}_r = \vec{t}_d}{(\vec{t}_l) \vee (\vec{t}_r) = (\vec{t}_d)} \\
\\
\text{MERGE-CONSTRUCTOR} \\
\frac{\vec{t}_l \vee \vec{t}_r = \vec{t}_d}{A \{ \vec{f} : \vec{t}_l \} \vee A \{ \vec{f} : \vec{t}_r \} = A \{ \vec{f} : \vec{t}_d \}} \\
\\
\text{MERGE-SINGLETON} \\
\frac{t_l \vee t_r = t_d}{= t_l \vee = t_r = = t_d} \\
\\
\text{MERGE-APP} \\
\frac{t_l \vee t_r = t_d \quad \begin{cases} u_{i,l} \vee u_{i,r} = u_{i,d} & \text{if } \text{variance}(t_d, i) = \text{co} \\ u_{i,l} = u_{i,r} = u_{i,d} & \text{if } \mathcal{V}_l \# u_l \text{ and } \mathcal{V}_r \# u_r \\ \text{failure} & \text{otherwise} \end{cases}}{t_l \vec{u}_l \vee t_r \vec{u}_r = t_d \vec{u}_d} \\
\\
\text{MERGE-CONSCONS} \\
\frac{P_l * x_l @ A \{ \vec{f}_l = \vec{y}_l \} \leq x_l @ t \vec{u}_l \quad \psi(x_l, x_r) = x_d \quad t \vec{u}_l \vee t \vec{u}_r = t \vec{u}_d}{P_l * x_l @ A \{ \vec{f}_l = \vec{y}_l \} \vee P_r * x_r @ B \{ \vec{f}_r = \vec{y}_r \} = t \vec{u}} \\
\\
\text{MERGE-CONSAPP-L} \\
\frac{P_l * x_l @ A \{ \vec{f}_l = \vec{y}_l \} \leq x_l @ t \vec{u}_l \quad \psi(x_l, x_r) = x_d \quad t \vec{u}_l \vee t \vec{u}_r = t \vec{u}_d}{P_l * x_l @ A \{ \vec{f}_l = \vec{y}_l \} \vee x_r @ t \vec{u}_r = x_d @ t \vec{u}_d}
\end{array}$$

Figure 12.6: The merge operation

Merging types pairwise

We now turn to the final step. We know how to match variables together; we know what the destination prefix should be. All that is left to do is replace occurrences of variables from \mathcal{V}_l (resp. \mathcal{V}_r) with variables from the image of ψ . In order to determine which replacement should be performed, we perform the exact same graph traversal as for the mapping; we merge types pairwise, using the mapping to perform the proper rewritings.

Form of our pairwise merge judgement We write our merge rules as $t_l \vee t_r = t_d$. The scope issues have been taken care of; all we need to ensure is the following.

Lemma 12.2. *If $t_l \vee t_r = t_d$, then all free variables in t_d are either within the image of ψ , or old variables.*

This guarantees that once the two local permissions P_l and P_r have been merged into P_d , we can discard \mathcal{V}_l and \mathcal{V}_r and return $\exists(\vec{X}_d : \vec{\kappa}).P_d$ directly. The rules for merging types pairwise are presented in Figure 12.6.

Reviewing the rules **MERGE-DEFAULT** is a fallback rule that is used whenever no other rule has been found. It amounts to saying that types that are syntactically equal in the original prefix \mathcal{V}_0 can be safely merged “as is” in the resulting environment. This allows, for instance, function types to be merged, as long as their signature does not mention local variables.

MERGE-ANCHORED and **MERGE-STAR** implement a recursive descent on a composite permission. We recursively merge anchored permissions whose left-hand sides match according to ψ . There is some non-determinism in the way we split the permissions in two (**MERGE-STAR**); this has an incidence only in the case where there is a conflict between who gets to own a piece of mutable data (the third sub-problem mentioned in the introduction).

The rules for the recursive descent are taken to operate, just like in the previous chapter, modulo associativity, commutativity, and are also taken to implicitly save duplicable permissions whenever focusing on an anchored

$x @ t$. That is, merging $x @ \text{int}$ with another type does *not* consume it from P_l . Merging $x @ \text{ref int}$ does consume it, however.

MERGE-VAR is key: it determines how variables that have been visited previously during the mapping phase are translated so as to make sense under the existential quantifiers \vec{X}_d (Lemma 12.2). There are several things to note about this rule.

- Lacking any kind precondition, this rule applies to both variables at kind type and at kind perm: this rule also merges “floating”, abstract permissions which are not anchored. This is the reason why I insisted earlier that the mapping should cover variables at all kinds: we have a single rules for abstract permission variables, type variables, program variables.
- Reviewing this rule proves our earlier claim that merging types pairwise yields a type where all variables are either old or in \vec{X}_d .
- The premise of this rule is always satisfied, that is, $\psi(X_l, X_r)$ is always defined: the set of rules from Figure 12.5 and Figure 12.6 perform the exact same recursive traversal. (The implementation performs the two in one go.)

MERGE-TUPLE, MERGE-CONSTRUCTOR and MERGE-SINGLETON are the structural rules that rely on the ψ mapping to perform the graph transformation. The points-to relationships, embodied via singleton types in $\mathcal{V}_{l,r}$, are replaced with a different set of points-to relationships according to the computed mapping.

MERGE-APP, MERGE-CONSCONS, MERGE-CONSAPP-L (as well as the symmetrical MERGE-CONSAPP-R) tackle the problem of applying subsumption rules to make permissions match (the second sub-problem mentioned in the introduction).

MERGE-APP intentionally has no kind pre-condition, so as to serve either for anchored permissions, or for abstract, “floating” permission applications. This rule has a different behavior depending on the variance of the i -th parameter. If the parameter is co-variant, then one can recursively merge the types: merging is a covariant operation. If the parameter is contra-variant, one should compute the intersection of the types: the type-checker currently has no support for this, so we merely check that the types do not mention local variables and are syntactically equal. This is also the behavior for invariant parameters. For bivariate parameters, we could be smarter and pick \perp ; this has not proved useful, however. In case the parameters cannot be merged, this is a failure and the type applications cannot be merged.

MERGE-CONSCONS and MERGE-CONSAPP-L allow folding either both sides or just one side of the disjunction. This elides the question of finding out which type application exactly we should seek to fold onto. This question is treated in the next section which details our implementation; in short, we use a subtraction with flexible variables for the type parameters.

These two rules only mention t , not t_i ; we do not allow locally-defined data types. Therefore, if t appears on both sides, it is necessarily an old variable. In MERGE-CONSAPP-L, the \vec{u}_i parameters may mention local variables. If this happens, then MERGE-APP may fail to apply. Again, the subsequent section about implementation details how we avoid this situation.

The pairwise merging rules perform the same traversal as the rules for building the mapping. This means, in particular, that these rules do not try to merge quantified types or function types: if MERGE-DEFAULT does not apply, these types are bluntly dropped. Rules MERGE-DYNAMIC and MERGE-UNKNOWN are omitted. The fully normalized precondition rules out type/permission conjunctions.

Putting the pieces together

We finally have enough to describe the entire algorithm. Solving the merge problem $\mathcal{V}_l.P_l \vee \mathcal{V}_r.P_r$ can be done by:

- defining $\psi_0 = \{(\vec{\rho}, \vec{\rho}, \vec{\rho})\}$ for all ρ in \mathcal{V}_0
- computing $\psi = \psi_0 + P_l \vee P_r$
- defining $\vec{X}_d : \vec{\kappa} = \text{image}(\psi)$
- defining $P_d = P_l \vee P_r$
- returning $\mathcal{V}_0.\exists(\vec{X}_d : \vec{\kappa}).P_d$

Theorem 12.3 (Merge soundness). *The result of the merge operation satisfies $P_l \leq \exists(\vec{X}_d : \vec{\kappa}).P_d$ and $P_r \leq \exists(\vec{X}_d : \vec{\kappa}).P_d$.*

Proof. By Lemma 12.2, P_d is properly scoped under $\exists(\vec{X}_d : \vec{\kappa})$. By Lemma 12.1, the rewritings performed by the pairwise-merging of types are correct as well. Reviewing the rules from Figure 12.6, we find that these correspond to the application of subsumption rules from Figure 8.1. \square

The complexity of the type system of Mezzo makes it difficult to state any stronger results. Experience can offer, however, a few conjectures. The first one is related to the existence of principal solution. We know, from the earlier examples, that having parameterized algebraic data types can yield two incomparable solutions, such as $\text{ret } @ \text{ option } = y * y @ \text{ ref int}$ vs. $\text{ret } @ \text{ option ref int}$. Similarly, we have seen that conflicts over assignment of mutable data lead to incomparable solutions, such as $\text{ret } @ (T, \text{dynamic})$ vs. $\text{ret } @ (\text{dynamic}, T)$. We can thus only hope for a weaker result.

Definition 12.4 (Restricted merge problem). *A restricted merge problem is a merge problem where no parameterized data types are involved and no affine permissions are involved either.*

Conjecture 12.5 (Principality). *The restricted merge problem admits a principal solution.*

Conjecture 12.6 (Completeness). *Our algorithm can be made complete for the restricted merge problem.*

Our algorithm currently does not propagate *local* equations to the destination environment, for complexity reasons: we would need to test every pair of variables, either old, or from \vec{X}_d , to see if they became equal in both environments. (Our implementation does not support a more efficient method.) Should we forget the performance concerns and implement this, we believe our algorithm to be complete for the restricted merge problem.

12.4 Implementation

The current implementation of the merge algorithm contains various tradeoffs; this section details the implementation choices that we made. The algorithm is definitely not complete; the implementation relies on subtraction, which is not complete anyway. Moreover, non-principal situations occur, where one has to decide how to assign an affine permission.

That being said, the merge algorithm, while complex, is one of the less contentious pieces of our type-checker. We have had a surprisingly good experience with the current algorithm, and hardly ever needed to use explicit type annotations. This may be due to our style of writing structured code with small, self-contained functions. Further experiments with the language and, hopefully, different programming styles, will tell.

Conflicts over assignment of affine permissions

I have not discussed the third sub-problem originally mentioned in the introduction, which is that of conflicting choices for assigning mutable data. As I mentioned earlier, a natural criterion emerges for detecting these situations. The implementation features a procedure for emitting a warning in case this situation arises, as well as a heuristic that solves most cases, thus making sure the problem hardly ever happens in practice.

A conflict arises when:

- there is an affine permission $x_l @ t$;
- x_l appears more than once in the mapping, e.g. $\psi(x_l, x_r) = x_d$ and $\psi(x_l, y_r) = z_d$.

This is precisely our first example from this chapter.

The root of the problem lies in MERGE-STAR. The rule decides how to split permissions; assuming the mapping above, depending on which of $x_l @ t \vee x_r @ t$ and $x_l @ t \vee y_r @ t$ is considered first, there are two possible outcomes:

- if the former is considered first, using $\psi(x_l, x_r) = x_d$, we obtain $x_d @ t$ in P_d , and $x_l @ t$ and $x_r @ t$ disappear from P_l and P_r , respectively; then, we consider $x_l @ \text{unknown} \vee y_r @ t$ and are unable to merge this;
- conversely, if the latter merge is considered first, then the outcome for P_d is $z_d @ t$.

Worded differently, the order in which we perform operations determines who “gets” the affine permission.

Lacking any backwards analysis, we have no way to know “in advance” which of the two solutions should be selected. We thus use a combination of a heuristic and a warning.

A heuristic We use the following heuristic: the special return value `ret` is always visited first. The consequence is that between a variable x_d reachable from ret_d and a variable y_d *not* reachable from ret_d , x_d always gets the affine permission. The rationale is that constructing a new value will most certainly consume older permissions. Otherwise, the new value would be less useful. This allows the following example to succeed without any other annotations.

```
(* r @ list (ref int) *)
let l =
  if ... then
    r
  else
    Nil
in
...
```

Without that heuristic, we would first “save” $r @ \text{list ref int}$, meaning that we would be unable to find a permission for l .

This is a heuristic that has served us well in practice. Along with downward type propagation of function return types, we hardly ever annotate matches. (This could probably be further improved, for instance by visiting roots from the most recently bound to the oldest one. We haven’t needed a more sophisticated heuristic yet.) We still want to warn the user that we made an arbitrary decision, though.

Emitting a warning We want to emit a warning whenever a permission $x_l @ t$ could have served in two different merge operations.

We mark both x_l and x_r whenever “ $x_l @ t_l \vee x_r @ t_r$ ” succeeds with a non-duplicable t_l (and t_r). Later on, if we revisit x_l (resp. x_r) with a distinct y_r (resp. y_l), we know that the destination variable is a new one. This means that $x_l @ t_l$ (resp. $x_r @ t_r$) could have been used for another destination variable. We emit the warning.

This warning has subtle implications.

```
data mutable t1 = T1
data mutable t2 = T2

val z =
  let x1 = T1 in
  let x2 = T2 in
  if true then
    (x1, x1)
  else
    (x1, x2)
```

The resulting permission is $z @ (T1, \text{unknown})$. There is nothing we could’ve done better here, that is, this is a principal solution, yet the type-checker emits a warning when visiting the second component of the tuple. We strongly believe, though, that the warning is useful: the example above is definitely over-the-top, and the user most certainly meant something else.

Treatment of equations

The entire discussion leaves equations out of the picture. In the local environments, three type of equations may need to be merged: local-local equations, old-old equations, and local-old equations.

Local-local equations and local-old equations Local-local equations are of the form $x_l = y_l$. These equations are handled transparently by our implementation, meaning that in the eyes of the algorithm, there is only one name for both x_l and y_l .

Local-old equations are of the form $x_l = y_0$ where y_0 is in \mathcal{V}_0 , that is, where y_0 is an old variable. Again, this is handled transparently, as if only y_0 exists. Types that mention x_l are rewritten by the type-checker to only

mention y_0 , so as to prevent any scoping issues. If y_0 was an old, flexible variable, then the equation is ignored by the type-checker since it is ill-scoped.

Old-old equations Old-old equations that were already present before the disjunction in control-flow are retained and are handled transparently by the type-checker.

However, old-old equations may *appear* in the local permissions (either between rigid or flexible variables). For instance, we may have in both the left and right permissions $x_0 = y_0$, where both x_0 and y_0 are old variables. We should, in theory, merge these as well. From an implementation perspective, we do not “see” these permissions explicitly as the variables x_0 and y_0 will simply be merged into the same equivalence class on both sides of the disjunction. We can solve this by iterating over all pairs of old variables to see if they “became” equal in both sides, and unify them in the destination environment as well. This is not implemented, as we have yet to encounter a single case where we need this feature.

Working state of the algorithm

The rules are declarative. The implementation, however, maintains some state. It manipulates three sets of permissions P_l , P_r and P_d in parallel; whenever a rule is successfully applied, the left and right permissions are removed from their respective sets if they were affine, and a permission is added into P_d . Whenever a subtraction is performed to infer the parameters of a type application, permissions are consumed just like in any regular subtraction.

The main loop works by iterating over the pairs of variables in ψ , at kind value. For each pair of variables (x_l, x_r) , and for each pair of $x_l @ t_l$ and $x_r @ t_r$ available, the algorithm tries to find a suitable merging rule for t_l and t_r by syntactically matching on the two types.

The algorithm performs the two sub-tasks at the same time, that is, it both builds the mapping and merges types pairwise in one go.

Inference and flexible variables

Order of variables There is a technical issue related to the order of variables. In order to justify our earlier claim that our renaming is correct, we used the argument that existential quantifiers can freely commute, hence allowing use to not worry about the order in which variables are introduced.

This is a tricky issue: in the case that there may be uninstantiated flexible variables, these prevent the variables from being reordered freely. Indeed, a careless reordering may allow for more instantiation choices for a flexible variable, hence breaking soundness.

We assume the left and right permissions to be \exists -normalized which implies, among other things, that any instantiated flexible variable $\mathcal{F}(X = \tau : \kappa)$ has been substituted with τ everywhere it occurs and that the corresponding entry has been dropped from the prefix (rule (10) of Figure 9.5).

This leaves us with the problem of uninstantiated flexible variables. To solve this, we can turn all flexible variables into rigid ones. This is a correct operation, since it amounts to turning a $\forall(X : \kappa)$ quantifier in the hypothesis into $\exists(X : \kappa)$ (and kinds are inhabited).



We are, in essence, failing to translate a variable that is flexible on both sides into a flexible variable in the resulting environment. We believe this is not an excessive restriction. Anecdotal evidence suggests it hardly ever happens in practice that the user should have flexible variables on both sides and intends on instantiating them later on in the program; besides, the user can always annotate the disjunction with the expected type, thus choosing the proper instantiation choice for the flexible variable.



Another way to deal with the flexible variable problem would be to extrude all uninstantiated flexible variables from \mathcal{V}_l or \mathcal{V}_r prior to performing the merge operations. That way, we could still reorder arbitrarily the local variables from $\mathcal{V}_{l,r}$, while keeping flexible variables flexible. This would limit the instantiation choices and would require changing the level of the flexible variables, which is currently unimplemented.

An oracle for making flexible variables rigid Another way to deal with flexible variables is to instantiate them suitably. We can thus assume an oracle that tells us how to instantiate flexible variables (even local ones) *prior* to the merge operation we described. After substituting them, we can proceed with the procedure we described, which assumes no flexible variables.

$$\begin{array}{c}
\text{MERGE-FLEXIBLE-L} \\
\frac{\mathcal{F}(X : \kappa) \in \mathcal{V}_l \quad \mathcal{V}_r \# t_r}{X \vee t_r = t_r}
\end{array}
\qquad
\begin{array}{c}
\text{MERGE-FLEXIBLE-R} \\
\frac{\mathcal{F}(X : \kappa) \in \mathcal{V}_r \quad \mathcal{V}_l \# t_l}{t_l \vee X = t_l}
\end{array}$$

Figure 12.7: Extra rules for eliminating flexible variables

The actual implementation, however, has no oracle and performs this on the fly. It works as follows: the pairwise-merging of types is extended with the two symmetrical rules from Figure 12.7. Naturally, the prefix is updated and threaded through the various rules to keep track of the instantiation choices that we made.

In essence, whenever we can instantiate a flexible variable with a type that contains non-local variables, we do so, hence ensuring that the resulting type makes sense in the destination environment.

This does not cover the case where merging hits two local flexible variables: a correct behavior would be either to fail, or to allocate a fresh flexible variable in the destination prefix, right after \mathcal{V}_0 .

Inferring type application parameters The way `MERGE-CONSCONS` is written assumes we know how to *fold* concrete data types into their corresponding type application. That is, the rule assumes that when faced with a “Cons { . . . }”, we know what kind of list it is. This is actually a non-trivial problem, as we need to *infer* the parameters of the type application. Consider, for instance, a disjunction between, on the right-hand side, $P_r = x_r @ \text{Nil}$ and, on the left-hand side:

$$\begin{aligned}
P_l = & \\
& x_l @ \text{Cons} \{ \text{head} = h; \text{tail} = t \} \\
& * t @ \text{Nil} \\
& * h @ (=y, =z) \\
& * y @ \text{int} \\
& * z @ \text{int}
\end{aligned}$$

We want to infer that x_l should be folded onto list (int, int). We can express this inference problem in terms of subtraction. We want an answer to:

$$\mathcal{F}(a_l).(P_l - \text{list } a_l)$$

The reason why this is a difficult problem is that there may be several solutions. Indeed, when trying to figure out how to fold this type, any of the following solutions is acceptable:

- $x_l @ \text{list } \exists a.a,$
- $x_l @ \text{list } =h,$
- $x_l @ \text{list } (=y, =z),$
- $x_l @ \text{list } (\text{int}, =z),$
- $x_l @ \text{list } (=y, \text{int}),$
- $x_l @ \text{list } (\text{int}, \text{int}).$

Only the last solution is “reasonable”. Other solutions pose a problem: it is likely that the variables h , y and z are *local* to the branch, meaning that there is no chance the other branch could also fold onto the same type application.

The question of picking a “good” a_l is crucial: as we saw right above, bad choices will result in a failure to merge types (`MERGE-APP`). Our subtraction therefore implements a heuristic where we try to *never instantiate a flexible variable onto a type that contains singletons*. In our example, our heuristic picks $a_l = (\text{int}, \text{int})$. This heuristic was mentioned earlier (§11.7).

Once this is done, we perform the same operation on the other side: we perform $\text{Nil} - \exists a_r.\text{list } a_r$, and are left, as a result, with list a_r , meaning that the variable did not have to be instantiated in order for the operation to succeed.

We then recursively merge “list (int, int)” with “list a_r ”. Using `MERGE-FLEX-R`, our algorithm checks that “(int, int)” makes sense in the destination prefix (that is, it doesn’t contain local variables), and proceeds with instantiating “ $a_r = (\text{int}, \text{int})$ ”. The result of the merge operation is thus “ $x_d @ \text{list } (\text{int}, \text{int})$ ”.

In the general case, we implement the `MERGE-CONSCONS` rule by performing a subtraction on each side, of the form:

$$P - \exists(\vec{X} : \vec{\kappa}).t \vec{X}$$

where t is the type the constructor belongs to, and $\vec{\kappa}$ are the kinds of the type parameters. We then recursively merge the type applications using `MERGE-APP`. The same goes for `MERGE-CONSAPP` except that only one subtraction occurs.

Should flexible variable instantiations occur when comparing two type parameters (`MERGE-APP`), these instantiation choices are propagated to the next merge operation.



This is similar to subtraction: we should indeed ensure that “ $t a_1 a_1 \vee t \text{int bool}$ ” fails.

Floating permissions

Permissions that are not anchored to a program variable are stored in a separate list. We take all possible combinations pairwise and try to merge them, using rule `MERGE-APP` and `MERGE-VAR`.

Mode hypotheses

Mode hypotheses that may have been acquired in local environments are not merged. This situation has never occurred in practice; we could, however, compute a join operation on the fact lattice.

12.5 Relation with other works

Compared with subtraction, the merge issue seems harder to connect with other works. The issue is briefly mentioned in the Vault paper [DF01] as “computing type agreement at join points”. The issue seems to be easier to deal with in the context of Vault; from the paper, it seems like computing type agreement amounts to keeping the intersection of the two key sets at the join point.

Most papers on separation logic focus on the entailment procedure; SeLoger [HIOP13] writes, in passing, “these shape analysis tools mix traditional abstract interpretation techniques (*e.g.* custom abstract joins)”. The lack of emphasis on this particular issue seems to imply that the problem is, somewhat, easier; understanding why exactly is worth further investigation.

The merge procedure was remarked early on to be similar to the join operation on abstract shape domains. This has been extensively studied by Rival *et al.* [CR08, Riv11]; the ψ notation is inspired by these papers. Indeed, computing a shape join amounts to folding inductive predicates and matching triples of memory locations. In Mezzo, sums are tagged, meaning that folding inductive predicates is easier; the type parameters, though, require inference, which makes it more difficult to handle.

While the entailment problem can be expressed in terms of graph homomorphisms [HIOP13, CHO⁺11], we never needed to reason in terms of graphs for the subtraction algorithm. For the merge procedure, however, graphs are much more prevalent: intuitively, we want to reconcile heap shapes into a common description, that is, heap graphs. Simple examples, such as the first two I mentioned, can be understood as computing a graph homomorphism. In the presence of parameterized algebraic data types, however, the analogy falls short: it is well-suited to lists segments, but less so to the data types of ML (matching together two list segments corresponds to conflating two nodes in the graph – this is the nature of a homomorphism; the corresponding graph operation for arbitrary data types seems less simple).

12.6 A glance at the implementation

The `Merge` module in `typing/Merge.ml` exports just one function, which has been mentioned already (§10.9).

```
val merge_envs: env -> ?annot:typ -> env * var -> env * var -> env * var
```

The actual signature is slightly more complicated, because we return three lists of variables that were involved in affine permission assignment conflicts (one for each of P_l , P_r and P_d). This helps the graphical error reporting mechanisms highlight relevant portions of the permission graphs.

The actual implementation has quite a bit of a history, and would probably benefit from a massive cleanup. The three sub-tasks that were identified and isolated in the former section are all performed together. The algorithm is centered around a list of pending triples to visit (a work list). It also maintains a list of known triples (the ψ function).

```
let pending_jobs: (var * var * var) list ref = Lifo.create () in
...
let known_triples: (var * var * var) list ref = Lifo.create () in
```

Using `pending_jobs`, it performs a depth-first traversal of the two graphs in parallel.

```
let rec merge_vars
  ((left_env, left_var): env * var)
  ((right_env, right_var): env * var)
  ((dest_env, dest_var): env * var): env * env * env =
  ...
```

The `merge_vars` function takes the left and right variables, but also a pre-allocated destination variable. This is quite different from the algorithm specification. There are two cases: either this pair of variables has been visited already (it is in `known_triples`), meaning that we should just unify the pre-allocated `dest_var` with whatever variable was used earlier, or this pair of variables has not been visited (it is not in `known_triples`), meaning that we should just proceed with merging pair-wise.

```
type outcome = MergeWith of var | Proceed

let what_should_i_do
  (left_env, left_var)
  (right_env, right_var)
  (dest_env, dest_var): outcome
=
...
```

Once a triple of variables has been selected, in case the variables are x_l and x_r at kind value, we need to take all pairs of permissions $x_l @ t$ and $x_r @ t$ and match them pairwise. If the permissions are affine, they need to be discarded from their respective environments. This quadratic exploration is done using the auxiliary function `merge_lists`, which merges two lists of permissions. The `merge_vars` function then takes care of updating the lists of permissions for `left_var`, `right_var` and `dest_var` respectively.

```
and merge_lists
  (left_env, left_perms, processed_left_perms)
  (right_env, remaining_right_perms)
  ?dest_var
  (dest_env, dest_perms): env * (typ list) * env * (typ list) * env * (typ list) =
```

This function is implemented using a zipper on the left list (hence the two parameters). The optional `dest_var` argument is used in the presence of type annotations. The `merge_lists` function returns three lists along with three environments, as the various folding steps may have consumed local permissions and added permissions in the destination environment. The function finally calls `merge_type`, which performs the actual pairwise merging of types.

```
and merge_type
  ((left_env, left_perm): env * typ)
  ((right_env, right_perm): env * typ)
  ?(dest_var: var option)
  (dest_env: env): (env * env * env * typ) option
=
```

The function, naturally, returns three environments as the process may consume permissions in `left_env` and `right_env`, and will add new permissions in `dest_env`.

The function may fail and return `None`: the `Merge` module is written in a monadic style, using the `Option.bind` operator as (`>>=`). The `merge_type` function is written as a series of pattern-matchings on `left_perm` and `right_perm`: the pattern-matchings are evaluated lazily in sequence, as later matchings make assumptions that rely on earlier matchings having failed. Here is a representative sample of `merge_type`, which implements the `MERGE-SINGLETON` rule.

```
| TySingleton left_t, TySingleton right_t ->
  let r = merge_type (left_env, left_t) (right_env, right_t) dest_env in
  r >>= fun (left_env, right_env, dest_env, dest_t) ->
    Some (left_env, right_env, dest_env, TySingleton dest_t)
```

Finally, the main loop of the algorithm is as follows:

```
let state = ref (left_env, right_env, dest_env) in
while not (Lifo.empty pending_jobs) do
  let left_env, right_env, dest_env = !state in
  let left_var, right_var, dest_var = Lifo.pop pending_jobs in
  let left_env, right_env, dest_env =
    merge_vars
      (left_env, left_var)
      (right_env, right_var)
      (dest_env, dest_var)
  in
  state := (left_env, right_env, dest_env);
done;

!state
```

We pop jobs repeatedly (this is a depth-first traversal) from the work queue; we call into the `merge_var` for each triple.

12.7 Future work

The merge operation relies on the heuristics of subtraction. Currently, the subtraction has no knowledge of whether a variable is local to a branch of the merge operation, or is an old variable. Having this information would help make instantiation decisions: for instance, leftover permissions for local variables may want to be captured via flexible instantiations, when folding concrete types. (This would help with the inference difficulty for rich booleans mentioned in §4.4).

Part V

Conclusion

13 Summary; perspectives

13.1 A brief history of Mezzo	197
13.2 Looking back on the design of Mezzo	198
13.3 Type system <i>vs.</i> program logic	198
13.4 Perspectives	199

List of Figures

Bibliography

13. Summary; perspectives

The initial goals I outlined in the introduction (§1.4) revolved around one main axis: designing a programming language that provides stronger guarantees.

13.1 A brief history of Mezzo

Initial goals The initial premise with Mezzo was to design a high-level language that could be used by an end-user. A big constraint was thus to keep the complexity budget reasonably low, and to make the language somehow palatable. Our target audience was seasoned ML programmers; that is, users who are familiar with functional programming, but not necessarily gurus when it comes to reasoning about ownership in their programs.

Another natural constraint was to design a language in the tradition of ML. As related work showed, there was already a rich history of advanced type systems. These systems were, however, mostly focused around either low-level or object-oriented languages. Applying these ideas in the context of ML thus constituted an exciting research opportunity.

A last constraint was to design a type system, not a program logic. Several works already had explored the “type-check first, verify later” motto. We wanted to try a fresh approach where everything would be presented as a type system.

A first (failed) attempt We initially focused on a low-level calculus that featured primitive blocks, integer tags, reduction inside of types, and recursive types using μ combinators. The grand idea was to later on design a surface language that would desugar onto that core calculus.

It turned out hard to proceed this way. Lacking any vision for what the surface language should feel like, progress on the core calculus stalled.

Things gradually became easier when we started thinking first about what the language should feel like; things unfolded naturally from then on. Eventually, type-checking was performed on Simple Mezzo, which is relatively close to the surface language, while the proof of soundness, conversely, did reason on a core calculus.

Design by example Writing examples, even though the language didn’t exist yet, turned out to be highly beneficial. One may explain the design of Mezzo by looking back at the examples.

Initially, we took inspiration from static regions. Few examples could be written successfully using just static regions, however: a big limitation indeed is that only one element may be taken out at a time. Dynamic regions were more flexible, and seemed to be a better fit in almost all the situations. We dropped static regions from the feature list of the language. We refined the dynamic test mechanism and came up with a sketch of the adoption/abandon mechanism.

We also came up early with two variants of data types: exclusively-owned and duplicable ones. Out of habit, we wrote both implementations and interfaces for our examples. This made us realize that we needed to reveal

facts about data types. Amusingly, the first algorithm designed for Mezzo was thus fact inference, although it was initially a much more ad-hoc piece of code that didn't rely on a lattice.

Trying to write the type-checking rules led us, naturally, into keeping track of equations to handle local aliasing. We realized that this could be expressed using singleton types.

We picked a name for the language ("kreMLin", along with the motto "in Soviet Russia, ML type-check you" never took off), and I set out to implement the prototype type-checker.

13.2 Looking back on the design of Mezzo

Better understanding programs I believe the language speaks for itself: the numerous examples, libraries, programming patterns we have seen throughout the present dissertation are evidence that one does indeed gain greater confidence by writing their programs in Mezzo.

One way to experience this is to try writing some sample Mezzo code: this requires a brand new discipline. The ML programmer is not accustomed to the ownership motto, and it does require some effort to state invariants via the permission mechanism. My personal experience is that it is highly beneficial: by stating clearly the invariants one desires, the structure of the resulting program is more rigorous. The type system oftentimes makes the programmer realize that their mental model was weak, or maybe imprecise. This is particularly true in the case of data structures: a recent experiment with semi-persistent arrays showed that the mental model of the two members of the lab (including the present author) was, at best, confused.

Another way to experience this is to read some Mezzo libraries. Having functions state clearly their pre- and post-conditions is a real relief when jumping in a codebase, trying to figure out the invariants.

State change, concurrency Interestingly enough, while the state change aspect was a big initial motivation, it turned out to be less prevalent than we thought. We *do* have state changes in Mezzo, and several code examples used the feature in the present dissertation. It is not anecdotal. Nonetheless, we never really felt the need to put state change as a first-class concept; rather, it is a consequence of our pervasive use of data types and of our mutability discipline. It seems that state change fits naturally in the language without having to single it out as a separate idiom.

Writing better concurrent code was also one of the initial motivations. The ownership discipline helps a lot in that regard. While this is no silver bullet (one may still deadlock, after all), it turns to be great that the type-checker makes sure no two threads will compete for the same permission unless they use a lock.

A potential limitation is that racy, correct programs cannot be expressed in Mezzo. I would even go so far as to argue that this is perhaps not the best language for that purpose.

Escape hatches It turns out we often resorted to adoption/abandon in our code; actually, as soon as one ventures out of the realm of trees and singly-linked lists, it becomes pretty obvious that one should switch to dynamic tests. This can be seen in two opposite ways. On the one hand, it may be a strength of the language that we should have such a convenient escape hatch, hence making sure the programmer is never "stuck" and can always write the programs they wish. On the other hand, not relying on adoption/abandon demands a great deal of expertise (the work on iterators is one such example), and it may imply that we need some extra mechanisms to accommodate for a greater variety of ownership patterns.

Nesting represents an alternative programming pattern; experience shows, however, that the programmer still has to write a few manual checks.

The borrowing pattern has been mentioned several times over the course of this dissertation; we still don't have a good type-theoretic feature to easily express this usage. Should we wish to push Mezzo further and make it a language that actual people use, we would probably need to add more features into the type system. Fractional permissions naturally come to mind, too. Experience from other languages seems to convey the idea that there is no one-size-fits-all "region mechanism" that could account for the variety of patterns.

13.3 Type system vs. program logic

Throughout the whole Mezzo adventure, it seemed like there was always a tension between having a type system and having a program logic. We *did* want to have just a type system, if only to explore what seemed like a less-trodden path. The result, however, is a little bit far from the usual idea of a type system.

Having a type system has numerous advantages, which I have stressed repeatedly. It is part of the development cycle; it builds upon a mental model that programmers (hopefully) maintain at any time when developing their programs. It offers a unified approach instead of the combination of a type system, and a program logic. Switching to an advanced type system is less of a conceptual gap than switching to a proof system. It allows for more programming idioms: a proof system on top of the ML type system does not relax the restrictions of the ML type-checking discipline.

The drawbacks that we uncovered are many, though: the type system of Mezzo loses many of the important properties one expects, such as principality, decidability, predictability... Even grasping the base concepts of the type system demands a high level of proficiency. The underlying theory is complex: even after some efforts, I probably still remain the only person capable of fully deciphering an error message from Mezzo. The flow-sensitive approach to type-checking makes it all the more similar to a program logic. The fact that the type system should apply heuristics and sometimes fail is, also, unusual.

Modern program verification tools seem to provide many of the advantages of a type system (responsiveness, an intuitive specification language, integration with the development process) without having the constraints of pretending to be one. Time will tell, certainly, which one of the two paths yields the greatest benefits.

13.4 Perspectives

Mezzo as a foundation for program proof An initial idea we had was to have an additional step “fill in the blanks”: whenever the user relied on dynamic tests, an additional layer would generate proof obligations to show that the dynamic tests were unnecessary. Piggybacking on a tool such as Why3, we envisioned a two-layered language, where casual users would remain within the realm of type-checking, while guru users would make the switch and also perform program verification. Such a prospect never materialized, though, and remains a good topic for exploration. Our modest experiments with integer arithmetic were inconclusive; I believe, however, that a system designed perhaps slightly differently may be able to keep the benefits of having a type system and reuse that first coarse verification to lighten the proof obligations.

Solving via external tools The feeling that I have after writing all these tailored algorithms is that they feel pretty much like a proof search mechanism in a theory. The subtraction and merging algorithms are fine-tuned for the type system of Mezzo; a generic SMT-solver may fare worse.

A line of work that I believe is worth exploring is a system where this “program logic” aspect is recognized and integrated from the start. Expressing all the type-checking rules as proof obligations may be a more natural way to go rather than express everything as typing rules. A type-checker would then prove easy obligations using specialized algorithms; an SMT-solver would take care of the rest. This requires, naturally, tremendous theoretical and engineering effort to provide the proper feedback in a timely and user-readable manner. Probably worth another PhD.

A research experiment The final story is that Mezzo was a research experiment; we never really had the manpower or the will to develop it into a full-fledged programming language *à la* Rust. Therefore, we never had much of an experience when it comes to real-world programs. Error messages would need some work; we would need to sort out the story for binding external libraries; we would need to add extra type-theoretic mechanisms for the programmer’s convenience. These mechanisms would not be novel, hence not fit for publication. All of these hypothetical, engineering-oriented items would be needed to make Mezzo a real language.

Guidelines for designing a new language As a piece of final advice, I would recommend to any fresh, enthusiastic PhD student trying to design a new language: first, to find a “killer application” that the language should make possible; second, to write many examples from the start, so as to gather as much feedback as possible.

List of Figures

1.1	A story about a real programmer	3
1.2	Assembly listing for the Apollo Guidance Computer – lunar landing module	3
1.3	Donald Knuth on programming languages [Knu74]	4
1.4	Tony Hoare on programming languages [Hoa81]	4
1.5	John Reynolds on typed vs. untyped languages [Rey85]	5
1.6	Mezzo vs. ML: potato, potato	8
1.7	A buggy Java program	10
2.1	A history of linear type systems, regions and capabilities (F. Pottier)	14
2.2	Sample typing rules for linear lambda calculus	15
2.3	The key rules of separation logic	18
2.4	The zoology of permissions in Plural	20
3.1	Using a write-once reference.	25
3.2	Implementation of write-once references	26
3.3	Interface of write-once references	28
3.4	Signature of the <code>thread</code> module	28
3.5	Ill-typed code.	29
3.6	Fixing the racy program, using a lock	30
3.7	Hiding internal state via a lock, generic version	30
3.8	Definition of lists and list concatenation	32
3.9	List concatenation in tail-recursive style	32
3.10	A minimal implementation of stacks, with a higher-order iteration function	36
3.11	Borrowing an element from a container in direct style	39
3.12	Alternative version of the <code>else</code> branch from Figure 3.11	41
3.13	A failed attempt to construct a cyclic graph	42
3.14	Graphs, a cyclic graph, and depth-first search, using adoption and abandon	43
4.1	Axiomatization of nesting in Mezzo	49
4.2	A union-find data structure implemented with nesting (interface)	51
4.3	A union-find data structure implemented with nesting	52
4.4	Adoption/abandon as a library	55
4.5	Affine closures as a library	56
4.6	The type of rich booleans	57
4.7	A program involving conjunction rejected by the type-checker	58
4.8	Axiomatization of locks	59
4.9	Axiomatization of conditions	60
4.10	Specialized implementation of a channel (interface)	60
4.11	Specialized implementation of a channel	61
4.12	Implementation of a server using <code>mychannel</code>	62

4.13	The interface of channels	62
4.14	Landin’s knot	63
5.1	Using the identity to encode coercions	68
5.2	Manually applying a coercion	71
5.3	Lifting a coercion on lists	71
5.4	An object-oriented encoding of a counter	73
5.5	Encoding Mitchell <i>et al.</i> ’s movable example in Mezzo	75
7.1	Syntax of types in Mezzo and Simple Mezzo	88
7.2	Name collection function	90
7.3	Types and permissions: well-kindedness (auxiliary judgement)	90
7.4	Types and permissions: well-kindedness	91
7.5	Types definitions: well-kindedness	91
7.6	Expressions: well-kindedness	91
7.7	Syntax of expressions	94
7.8	Type-to-pattern function	95
7.9	Types and permissions: first translation phase (auxiliary judgement)	95
7.10	Types and permissions: first translation phase	95
7.11	Types and permissions: second translation phase (only one rule shown)	96
7.12	Expressions: translation	97
8.1	Permission subsumption	100
8.2	Some sample derivable subsumption rules	101
8.3	Variance-dependent subsumption judgement	101
8.4	Subtyping judgement	101
8.5	Typing rules	104
8.6	Auxiliary typing rules for pattern matching	105
8.7	The hierarchy of <i>modes</i>	108
8.8	Syntax of facts	109
8.9	Definition of the “exclusive” judgement (some rules omitted)	110
8.10	Definition of the “duplicable” judgement	110
8.11	Extra typing rules for dealing with mode constraint/type conjunctions	112
8.12	Fact inference	115
8.13	The variance lattice	116
8.14	The variance environment and the variance predicates	117
8.15	Alternative give and take operators	119
8.16	The graph example, using the new formalization of adoption/abandon	120
8.17	The foreach function from CaReSL, transcribed in Mezzo [TDB13]	123
9.1	A sample permission recombination	130
9.2	Sample excerpt from the <code>list::length</code> function	131
9.3	Introducing flexible variables for polymorphic function call inference	131
9.4	Syntax of prefixes	132
9.5	Normalization	133
9.6	Representation of variables	139
9.7	Environment and variable descriptors	140
10.1	The high-level type-checking algorithm	144
11.1	The classic map function	155
11.2	The rules of subtraction	159
11.3	Variance-dependent subtraction	159
11.4	Equational rewriting in subtraction	163
11.5	Instantiation of flexible variables (-R rules omitted)	163
11.6	Landin’s knot	166

11.7	A complete subtraction example (kinds omitted for readability)	168
11.8	A valid Mezzo program that can't be type-checked	172
11.9	The interface of the Derivations module	173
12.1	First merge example	176
12.2	Second merge example	176
12.3	Third merge example	176
12.4	Fourth merge example	176
12.5	Computing the ψ function	181
12.6	The merge operation	185
12.7	Extra rules for eliminating flexible variables	190

Bibliography

- [AFKT03] Alex Aiken, Jeffrey S Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. *ACM SIGPLAN Notices*, 38(5):129–140, 2003.
- [AFM07] Amal Ahmed, Matthew Fluet, and Greg Morrisett. L^3 : A linear language with locations. *Fundamenta Informaticæ*, 77(4):397–449, 2007.
- [Ald10] Jonathan Aldrich. Resource-based programming in Plaid. *Fun Ideas and Thoughts*, 2010.
- [BA07] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 301–320, 2007.
- [Bak90] Henry G. Baker. Unify and conquer (garbage, updating, aliasing, ...) in functional languages. In *ACM Symposium on Lisp and Functional Programming (LFP)*, pages 218–226, 1990.
- [Bat14] Batteries included. *BatList*, 2014.
- [BBA11] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Checking concurrent typestate with access permissions in Plural: A retrospective. In Peri L. Tarr and Alexander L. Wolf, editors, *Engineering of Software*, pages 35–48. Springer, 2011.
- [BCO04] Josh Berdine, Cristiano Calcagno, and Peter W. O’Hearn. A decidable fragment of separation logic. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 3328 of *Lecture Notes in Computer Science*, pages 97–109. Springer, 2004.
- [BCO05a] Josh Berdine, Cristiano Calcagno, and Peter W. O’Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.
- [BCO05b] Josh Berdine, Cristiano Calcagno, and Peter W. O’Hearn. Symbolic execution with separation logic. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2005.
- [BFGP13] James Brotherston, Carsten Fuhs, Nikos Gorogiannis, and Juan Navarro Perez. A decision procedure for satisfiability in separation logic with inductive predicates. *RN*, 13:15, 2013.
- [BFL⁺11] Mike Barnett, Manuel Fähndrich, K Rustan M Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the spec# experience. *Communications of the ACM*, 54(6):81–91, 2011.
- [BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.
- [Boy03] John Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium (SAS)*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

- [Boy10] John Tang Boyland. Semantics of fractional permissions with nesting. *ACM Transactions on Programming Languages and Systems*, 32(6), 2010.
- [BPP14a] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. The design and formalization of Mezzo, a permission-based programming language. Submitted for publication, July 2014.
- [BPP14b] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. Type soundness and race freedom for Mezzo. In *Proceedings of the 12th International Symposium on Functional and Logic Programming (FLOPS 2014)*, volume 8475 of *Lecture Notes in Computer Science*, pages 253–269. Springer, 2014.
- [BRIT93] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML kit (version 1). Technical Report DIKU 93/14, Department of Computer Science, University of Copenhagen, 1993.
- [CCH⁺89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 273–280. ACM, 1989.
- [CDOY09] Cristiano Calcagno, Dino Distefano, Peter W. O’Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Principles of Programming Languages (POPL)*, pages 289–300, 2009.
- [Cha10] Arthur Charguéraud. *Characteristic Formulae for Mechanized Program Verification*. PhD thesis, Université Paris 7, 2010.
- [CHO⁺11] Byron Cook, Christoph Haase, Joël Ouaknine, Matthew Parkinson, and James Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR 2011—Concurrency Theory*, pages 235–249. Springer, 2011.
- [Chr98] Jacek Chrzaszcz. Polymorphic subtyping without distributivity. In *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science*, pages 346–355. Springer-Verlag, 1998.
- [CMM⁺09] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *International Conference on Functional Programming (ICFP)*, pages 79–90, 2009.
- [Coo91] William R Cook. Object-oriented programming versus abstract data types. In *Foundations of Object-Oriented Languages*, pages 151–178. Springer, 1991.
- [CÖSW13] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming*, volume 7850 of *Lecture Notes in Computer Science*, pages 15–58. Springer, 2013.
- [CP08] Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *International Conference on Functional Programming (ICFP)*, pages 213–224, 2008.
- [CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 48–64, 1998.
- [CR08] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Principles of Programming Languages (POPL)*, pages 247–260, 2008.
- [CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.
- [CWM99] Karl Cray, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Principles of Programming Languages (POPL)*, pages 262–275, 1999.
- [CYO01] Cristiano Calcagno, Hongseok Yang, and Peter W O’hearn. Computability and complexity results for a spatial assertion language for data structures. In *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, pages 108–119. Springer, 2001.

- [DCM10] Roberto Di Cosmo and Dale Miller. Linear logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2010 edition, 2010.
- [DFo1] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Programming Language Design and Implementation (PLDI)*, pages 59–69, 2001.
- [DFo4] Robert DeLine and Manuel Fähndrich. Tpestates for objects. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004.
- [DPo8] Dino Distefano and Matthew J. Parkinson. jStar: towards practical verification for Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 213–226, 2008.
- [FDo2] Manuel Fähndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In *Programming Language Design and Implementation (PLDI)*, pages 13–24, 2002.
- [Fil03] Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, 2003.
- [Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
- [GJLS87] David K. Gifford, Pierre Jouvelot, John M. Lucassen, and Mark A. Sheldon. FX-87 reference manual. Technical Report MIT/LCS/TR-407, Massachusetts Institute of Technology, 1987.
- [GJSO92] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O’Toole. Report on the FX-91 programming language. Technical Report MIT/LCS/TR-531, Massachusetts Institute of Technology, 1992.
- [GMJ⁺02] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Programming Language Design and Implementation (PLDI)*, pages 282–293, 2002.
- [goo] Google C++ style guide.
- [GPP⁺12] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 21–40, 2012.
- [GPP13] Armaël Guéneau, François Pottier, and Jonathan Protzenko. The ins and outs of iteration in Mezzo. HOPE 2013, July 2013.
- [HIOP13] Christoph Haase, Samin Ishtiaq, Joël Ouaknine, and Matthew J Parkinson. Seloger: A tool for graph-based reasoning in separation logic. In *Computer Aided Verification*, pages 790–795. Springer, 2013.
- [HLMS13] Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. Abstract read permissions: Fractional permissions without the fractions. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 7737 of *Lecture Notes in Computer Science*, pages 315–334. Springer, 2013.
- [Hoa81] Charles Antony Richard Hoare. The 1980 ACM turing award lecture. *Communications*, 1981.
- [HPo5] Robert Harper and Benjamin C. Pierce. Design considerations for ML-style module systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 8, pages 293–345. MIT Press, 2005.
- [Jen13] Jonas B Jensen. Techniques for model construction in separation logic. 2013.
- [JG91] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 303–310. ACM, 1991.

- [JKM⁺14] Uri Juhasz, Ioannis T Kassios, Peter Müller, Milos Novacek, Malte Schwerhoff, and Alexander J Summers. Viper: a verification infrastructure for permission-based reasoning. 2014.
- [JMG⁺02] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.
- [Knu74] Donald E Knuth. Structured programming with go to statements. *ACM Computing Surveys (CSUR)*, 6(4):261–301, 1974.
- [Lar89] James Richard Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, EECS Department, University of California, Berkeley, 1989. Technical Report UCB/CSD-89-502.
- [LDF⁺14] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system release 4.02. *Institut National de Recherche en Informatique et en Automatique*, 2014.
- [LM07] Chuck Liang and Dale Miller. Focusing and polarization in intuitionistic logic. In *Computer Science Logic*, pages 451–465. Springer, 2007.
- [Mat14] Niko Matsakis. PLT-redex model of rust, 2014.
- [Min98] Yasuhiko Minamide. A functional representation of data structures with a hole. In *Principles of Programming Languages (POPL)*, pages 75–84, 1998.
- [MSY11] Toshiyuki Maeda, Haruki Sato, and Akinori Yonezawa. Extended alias type system using separating implication. In *Types in Language Design and Implementation (TLDI)*, 2011.
- [MWCG99] Greg Morrisett, David Walker, Karl Cray, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, 1999.
- [NBAB12] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. In *Principles of Programming Languages (POPL)*, pages 557–570, 2012.
- [NDQC07] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 4349 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2007.
- [Nor09] Ulf Norell. Dependently typed programming in agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009.
- [NR11] Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *Programming Language Design and Implementation (PLDI)*, pages 556–566, 2011.
- [opa] OPAM, the package manager for OCaml.
- [Pos83] Ed Post. Real programmers don’t use pascal. *Datamation*, 29(7), 1983.
- [Poto7] François Pottier. Wandering through linear types, capabilities and regions. 2007.
- [Poto9] François Pottier. Lazy least fixed points in ML. Unpublished, December 2009.
- [PR05] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.
- [PR13] Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic modulo theories. In *Programming Languages and Systems*, pages 90–106. Springer, 2013.

- [Rey85] John C. Reynolds. Three approaches to type structure. In *Theory and Practice of Software Development (TAPSOFT)*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer, 1985.
- [Reyo2] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74, 2002.
- [Riv11] Xavier Rival. *Abstract Domains for the Static Analysis of Programs Manipulating Complex Data Structures*. Habilitation à diriger des recherches, École Normale Supérieure, 2011.
- [Ré89] Didier Rémy. Type checking records and variants in a natural extension of ML. In *Principles of Programming Languages (POPL)*, pages 77–88, 1989.
- [SCF⁺11] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthik Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *International Conference on Functional Programming (ICFP)*, pages 266–278, 2011.
- [SHM⁺06] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe manual memory management in Cyclone. *Science of Computer Programming*, 62(2):122–144, 2006.
- [SNS⁺11] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in plaid. In *ACM SIGPLAN Notices*, volume 46, pages 713–732. ACM, 2011.
- [SP14a] Gabriel Scherer and Jonathan Protzenko. A toy type language: parsing and pretty-printing, 2014.
- [SP14b] Gabriel Scherer and Jonathan Protzenko. A toy type language: using Fix to compute variance, 2014.
- [SP14c] Gabriel Scherer and Jonathan Protzenko. A toy type language: variance 101, 2014.
- [SWM00] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming (ESOP)*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381. Springer, 2000.
- [SY86] Robert E. Strom and Shaula Yemini. Tpestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.
- [TDB13] Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *International Conference on Functional Programming (ICFP)*, pages 377–390, 2013.
- [Theo6] The Coq development team. *The Coq Proof Assistant*, 2006.
- [The14] The Mozilla foundation. The Rust programming language, 2014.
- [TP10] Jesse A. Tov and Riccardo Pucella. Stateful contracts for affine types. In *European Symposium on Programming (ESOP)*, volume 6012 of *Lecture Notes in Computer Science*, pages 550–569. Springer, 2010.
- [TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ -calculus using a stack of regions. In *Principles of Programming Languages (POPL)*, pages 188–201, 1994.
- [TU96] J Tiurnyn and P Urzyczyn. The subtyping problem for second-order types is undecidable. In *Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 74–85. IEEE, 1996.
- [Tue10] Thomas Tuerk. Local reasoning about while-loops. Unpublished, 2010.
- [Wad90] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*. North Holland, 1990.
- [Wik14a] Wikipedia. List of programming languages (alphabetical), 2014. [Online; accessed April, 15th 2014].

- [Wik14b] Wikipedia. Singular they, 2014. [Online; accessed April, 16th 2014].
- [WM00] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Types in Compilation (TIC)*, volume 2071 of *Lecture Notes in Computer Science*, pages 177–206. Springer, 2000.