Ecole des Mines de Paris
Thèse en vue de l'obtention du diplôme de docteur en informatique

Typage modulaire des multi-méthodes

Modular typing for multi-methods

Daniel Bonniot

Cette thèse a été soutenue le 18 novembre 2005
Le jury était constitué de

François Bourdoncle
Pierre Cointe
Roberto Di Cosmo
Jacques Garrigue (rapporteur)
Didier Rémy (directeur)
Manuel Serrano (rapporteur)

Résumé

Cette thèse présente un système de typage statique pour les langages à multi-méthodes avec la particularité de pouvoir être effectué modulairement, sans nécessiter la connaissance du programme entier. Nous montrons également comment concilier les multi-méthodes avec un langage noyau à la ML avec sous-typage, tout en préservant l'inférence de types pour le langage noyau, seul le type des méthodes devant être déclaré.

Notre présentation est elle aussi modulaire. Nous posons tout d'abord un système de types algébrique, qui comprend un langage noyau avec constantes et des types entièrement abstraits. Un langage peut être construit par choix des constantes et du langage des types ainsi que de leur relation d'ordre. Nous pouvons dès lors identifier des conditions sous lesquelles le langage résultant est statiquement sûr. Cela nous permet d'étudier plus facilement des extensions dans deux directions différentes: pour l'expressivité des constructions du langage et pour la richesse du langage de type, tout en partageant une partie de la preuve de sûreté.

Dans la première direction, nous formalisons nos multi-méthodes comme une façon de définir des constantes du langage. Dans la seconde, nous présentons un raffinement du système de types $ML_{\leq}$ en rajoutant des contraintes de kinding, qui permettent d'exprimer de type de méthodes "partiellement polymorphes", c'est à dire dont le type à une précision intermédiaire entre un celle d'un type monomorphe et celle d'un type polymorphe contraint classique, d'une manière modulaire.

Enfin, nous étudions le défi classique du "problème des expressions" pour valider la pertinence de nos propositions et les comparer aux solutions existantes. Nous donnons également un aperçu de la mise en oeuvre de l'ensemble de ces idées par la conception et l'implémentation d'un langage complet, Nice.

Summary

This thesis presents a static type system for languages with multi-methods. Typing can be performed modularly, with the knowledge of the whole program. We also show how to mix multi-methods with a core language *a la* ML with subtyping, while preserving type inference for the core language: only method types have to be declared.

Our presentation is modular as well. We first define an algebraic type system that includes a core language with constants and fully abstract types. A language can be built by chosing constants and a language of types with an ordering. We identify conditions that guarantee the resulting language is statically safe. This presentation makes it possible to study more easily extensions in two different directions: for the expressivity of the language constructs and for the preciseness of the type language, while sharing a part of the safety proof.

In the first direction, we formalize our multi-methods as a way to define constants for the core language. In the second one, we present an evolution of the $ML_{\leq}$ type system by adding kinding constraints that allow to express the type of "partially polymorphic" methods, that is, whose type's precision lies between that of a monomorphic type and that of a bounded polymorphic type, in a modular fashion.

Finally, we study the classical challenge of the "expression problem" to validate the interest of our propositions and compare them to existing ones. We also give an overview of the practical aspect of all those ideas with the design and implementation of a complete language, Nice.

Remerciements

Je remercie **Didier Rémy** pour son encadrement lors de cette thèse. Sa vaste connaissance du domaine et sa rigueur scientifique m'ont été indispensables pour réaliser ce travail.

Je suis très reconnaissant à **François Bourdoncle** de m'avoir fait découvrir le domaine de recherche des systèmes de types pour les langage à multi-méthodes. Par sa vision, il a su me communiquer l'enthousiasme qui a rendu cette recherche passionnante.

Je remercie vivement **Jacques Garrigue** et **Manuel Serrano** qui ont bien voulu être rapporteurs de cette thèse. Je leur suis très reconnaissant pour les encouragements et les remarques précises qu'ils ont pu faire sur ce travail.

Je remercie également **Pierre Cointe** et **Roberto Di Cosmo** qui ont eu la gentillesse de participer à mon jury.

Je suis grandement redevable de toute l'équipe **Cristal** de l'INRIA Rocquencourt, qui m'a accueilli, soutenu, et offert de nombreuses occasions d'élargir mes connaissances du domaines et d'affiner mes propositions.

Je remercie tout particulièrement **François Pottier** qui, à de multiples reprises, a bien voulu relire mes articles, et dont les remarques éclairantes ont eu une grande influence sur mon travail.

Contents

I Fondations **21**

1 Algebraic type system **23**
1.1 Type algebra . . . 23
1.1.1 Sub-algebras . . . 25
1.1.2 Example . . . 25
1.2 Core language . . . 26
1.2.1 Interpretation . . . 30
1.2.2 Progress . . . 31
1.3 Instantiating the algebraic type system . . . 32

2 Type algebras **33**
2.1 The Hindley-Milner type system . . . 33
2.1.1 Instantiation of the framework . . . 33
2.1.2 Simplification . . . 39
2.1.3 Example . . . 39
2.1.4 Beyond Core-ML . . . 40
2.2 $ML_{\leq}$ . . . 40
2.2.1 Type structure . . . 41
2.2.2 Constraints . . . 41
2.2.3 Constrained types . . . 44
2.2.4 Instantiation of the framework . . . 44

II Object-orientation **51**

3 Classes **53**
3.1 Object instantiation . . . 54
3.2 Field access in classes . . . 55

4 Generic functions **57**
4.1 Example . . . 57
4.2 Syntax . . . 58
4.3 Semantics . . . 58
4.4 Type-checking . . . 59

5 Super **63**
5.1 Super in class-based languages . . . 63
5.2 Super in multi-method languages . . . 63
5.2.1 Dylan . . . 63
5.2.2 Cecil . . . 65
5.3 Formalization . . . 65

5.3.1 Typing . . . 66
5.4 Example . . . 66

6 Kinds **69**
6.1 Introduction . . . 69
6.2 Typing homogeneous operations . . . 70
6.3 Partially polymorphic functions . . . 73
6.4 Using kinds to type partially polymorphic functions . . . 75
6.5 Closed-world formalization . . . 75

III Modularity **77**

7 Modular type algebras **79**
7.1 $ML_{\leq}$ . . . 79
7.1.1 Variants of $ML_{\leq}$ . . . 79
7.1.2 Original $ML_{\leq}$ . . . 80

8 Open generic functions **83**
8.1 Syntax and semantics . . . 84
8.2 Modular type-checking . . . 85
8.3 Early detection of errors . . . 86
8.4 Type inference for open generic functions . . . 87
8.5 $ML_{\leq}$ . . . 88
8.6 $ML_{\leq}$ multi-methods . . . 88
8.6.1 Syntax . . . 88
8.6.2 Type-checking . . . 89
8.6.3 Examples . . . 90
8.6.4 Semantics . . . 91

9 Super in a modular setting **93**
9.1 Formalization . . . 93
9.2 Consequences of the precocity rule . . . 94

10 Modular kinds **97**
10.1 The open world problem . . . 97
10.2 Open-world formalization . . . 98
10.3 Language . . . 102
10.4 Conclusion . . . 103

IV Practice **105**

11 Code generation **107**
11.1 Monomorphic bytecode language . . . 107
11.1.1 Type checking . . . 109
11.2 Monomorphic instances of polytypes . . . 110
11.2.1 Type Constructors . . . 110
11.2.2 Constrained polymorphic types . . . 111
11.3 Compilation . . . 112
11.3.1 Types . . . 112
11.3.2 Programs . . . 113

12 Typing kinds **121**
12.1 Constraint decomposition . . . 121
12.2 Core ML$_{\leq}$ . . . 121
12.3 Adding kinds . . . 122

13 The Nice language **125**
13.1 Syntax . . . 125
13.1.1 Classes . . . 125
13.1.2 Methods . . . 125
13.1.3 Kinds . . . 126
13.2 Type checking . . . 127
13.2.1 Option types . . . 127
13.3 Code generation . . . 128

14 The expression problem **129**
14.1 Base . . . 130
14.2 Data extension . . . 130
14.2.1 Linear extension . . . 130
14.2.2 Combining independent extensions . . . 131
14.3 Operations extensions . . . 131
14.3.1 Linear extensions . . . 131
14.3.2 Tree transformer extensions . . . 132
14.3.3 Combining independent extensions . . . 132
14.3.4 Binary methods . . . 132
14.4 Discussion . . . 134
14.5 Comparison with polymorphic variants . . . 134

15 Related work **137**
15.1 Core . . . 137
15.2 Modular multi-methods . . . 138
15.3 Kinds . . . 139

16 Conclusion **141**

Introduction

On peut définir la programmation comme l'activité humaine consistant à produire un programme à partir d'une spécification. Dans cette définition, un programme est une description formelle d'une tâche exécutable par un ordinateur. Une spécification est une description de plus haut niveau qui spécifie une tâche. Les spécifications peuvent être plus ou moins formelles, depuis une description vague comme "un programme pour lire des courriers électroniques" jusqu'à une description complète du comportement du programme. Nous insistons sur le fait que la programmation est réalisée par des êtres humains. Ont pourrait en effet être tenté d'appeler spécification une description formelle mais de haut niveau qui peut être exécutée par un ordinateur, par exemple quand un programme peut en être extrait automatiquement. Cependant, on peut dans ce cas voir la spécification elle-même comme un programme. Elle a été écrite sur la base d'un description préexistante de plus haut niveau, qui est ce que nous appellerons la spécification. Le fait que les programmes soient écrits par des humains est important car cela conditionne les critères à utiliser pour évaluer la conception des langages de programmation.

Il est bien connu que tous les langages de programmation généralistes sont équivalents, dans le sens où pour tout programme écrit dans l'un de ces langages et effectuant une certaine tâche, et pour tout autre langage, il existe un programme dans ce langage qui effectue la même tâche. Cependant, cela ne rend pas caduque l'idée d'améliorer les langage existants, ou d'en créer de nouveaux. Simplement, on doit se focaliser sur la conception de langage qui rendent plus facile la programmation pour les programmeurs. Nous identifions quatre critères qui vont dans ce sens.

1. Les programmes doivent satisfaire leur spécification. En particulier, l'exécution d'un programme ne devrait jamais atteindre un état invalide où il "plante" ou doit être interrompu avant d'avoir achevé sa tâche. Un langage qui a la propriété de sûreté statique permet de garantir qu'un certain programme ne va jamais conduire à une telle situation. De plus, un langage dont les programmes peuvent être annotés de propriétés vérifiables statiquement (types, assertions logiques, ...) permet plus facilement de vérifier que le programme satisfait la spécification, éventuellement dans une forme affaiblie. Plus ces propriétés sont riches, plus elles peuvent fidèlement exprimer la spécification.

2. Il doit être "simple" d'écrire des programmes. La simplicité étant difficile à formaliser, ce critère est difficile à évaluer. Typiquement, la possibilité d'écrire des programmes de façon concise (en pouvant omettre les opérations de bas niveau) et la facilité pour d'autres programmeurs de comprendre les programmes sont des signes de simplicité du langage. Quand une spécification formelle existe, la simplicité peut être caractérisée par un faible écart entre celle-ci et le programme.

3. Puisque les grands programmes sont écrits par des équipes de programmeurs, il doit être possible de les écrire modulairement. Cela veut dire que la tâche principale doit être décomposable en tâches plus petites, implémentés dans des parties de programme appelées modules. Chaque module doit être implémentable avec une connaissance minimale des autres modules dont il dépend.

4. Pour éviter du travail inutile, il doit être possible de partager des modules d'utilité générale entre différents programmes [26]. Ce critère découle du précédent mais le dépasse de deux façons. D'une part, la réutilisation est souvent difficile car chaque programme peut avoir besoin d'un comportement différent de ce que le module partagé propose, ou encore le programme contient des cas additionnels à

ceux traités par le module partagé. D'autre part, puisque ce module partagé est développé indépendamment des programmes client, on ne peut pas supposer qu'il est possible de modifier le module pour les besoins spécifiques du programme. Le langage doit donc offrir des mécanismes pour adapter et étendre les modules importés.

Notre objectif général dans cette thèse est d'identifier des situations où ces critères ne sont pas bien respectés par les langage de programmation actuels, de proposer des mécanismes permettant de traiter ces situations, de prouver que ces mécanismes ont des bonnes propriétés et d'illustrer comment ils peuvent être utilisés en pratique.

Dans le domaine de la recherche en langages de programmation, la base la plus généralement acceptée est le Meta-Langage (ML) de Robin Milner. La théorie de ce langage est bien maîtrisée, ce qui en fait une base idéale pour proposer des extensions. De plus, il a de bonne propriétés, comme la possibilité d'inférer les types et de manipuler les fonctions comme des valeurs de première classe. En conséquence, de nombreux langages de recherche actuels sont des extensions de ML. Toutefois, cette recherche a un impact réel mais limité sur les langages de programmation les plus utilisés dans l'industrie informatique. Basés traditionnellement sur le paradigme impératif, ils ont dans ces dernières années été étendus pour inclure le paradigme de l'orientation objet. Cette extension a été motivée par le besoin de mieux permettre la conception de systèmes à grande échelle et de fournir plus de richesse d'expression. Ces objectifs auraient pu être atteints en utilisant le paradigme fonctionnel de ML, quoique d'une manière différente. Cela n'a probablement pas eu lieu car l'orientation objet pouvait plus facilement être présentée comme une extension de la programmation impérative (le premier langage orienté objets très largement répandu, C++, est une extension pure du langage C). Il est de toutes façons intéressant de comparer comment les approches orientées objets et fonctionnelles propose de résoudre les même problèmes, et comment elles répondent à nos quatre critères. Cela peut en effet inspirer la création de nouveaux langages ou de nouveaux paradigmes qui répondent mieux à ces critères.

Multi-méthodes

Il est reconnue que la programmation comporte deux aspects: la définition d'opérations et la définition de structures de données [19]. Un paradigme de programmation doit donc comporter des façons d'exprimer ces deux aspects. Le paradigme fonctionnel utilise principalement les types sommes (unions) et produits comme structures de données, et les fonctions définies par pattern-matching pour les opérations. Le paradigme orienté objets offre les classes comme moyen de structurer les données, et les méthodes pour opérer sur celles-ci. Toutefois, ces deux paradigmes introduisent une asymétrie entre ces deux aspects. Dans un programme de style fonctionnel, les structures de données peuvent être définies indépendamment des fonctions, alors que l'écriture des fonctions par pattern matching requiert la connaissance de tous les cas du type de donnée concerné. À l'inverse, dans le paradigme orienté objets, les méthodes sont définies localement à une classe indépendamment des autres définitions de classes, alors que les classes doivent inclure la liste de toutes leurs méthodes.

Cette asymétrie est problématique lorsque l'on considère nos deux derniers critères de modularité et d'extensibilité. Pour poursuivre le dualisme ci-dessus, nous devons donc à la fois pouvoir définir de nouvelles opérations sur des structures de données existantes et définir de nouvelles structures rentrant dans le champ des opérations existantes. Dans le paradigme fonctionnel, la définition de nouvelles fonctions est triviale. Néanmoins, l'extension des types sommes n'est pas possible puisque cela rendrait invalide les fonctions existantes définies par pattern matching sur ces types: elles n'auraient pas de branche pour les nouveaux cas. À l'inverse, en programmation orientée objets, l'extension des structures de données revient à écrire de nouvelles classes, ce qui est au coeur du paradigme, alors que la définition de nouvelles méthodes pour les classes existantes n'est pas autorisée.

Le fait que chacun de ces paradigmes privilégie un aspect différent explique pourquoi ils sont vus comme antagonistes. Toutefois, comme le montre la question de la modularité, les deux choix ont des inconvénients. Il est donc intéressant de chercher à résoudre ce conflit. Cela suppose de faire en même temps des opérations et des définitions de structures de données des concepts de première classe, définissables indépendamment

l'un de l'autre. Cela nécessite l'introduction d'un troisième concept: l'implémentation d'une opération pour un certain type de données. Ce nouveau paradigme est apparu initialement dans le langage CLOS, avec l'usage d'une terminologie orientée objets (méthodes et classes). De telles méthodes sont alors appelées *multi-méthodes*. Par rapport à l'orientation objet traditionnelle, celle nouvelle approche ajoute un aspect de la programmation fonctionnelle: la possibilité d'ajouter de nouvelles opérations définies par "pattern-matching" sur plusieurs arguments. Toutefois, cette approche n'avait jusqu'ici jamais été présentée dans un contexte incluant d'autres aspects de la programmation fonctionnelle, comme le polymorphisme générique et l'inférence de types. De plus, un défi additionnel est d'effectuer la vérification de types de programmes avec multi-méthodes d'une façon modulaire. C'est l'un des objectifs principaux de cette thèse.

Kinds

Un autre aspect de notre travail est de montrer l'intérêt et de formaliser une extension de systèmes de types dans le but de mieux respecter notre premier critère de typage statique fort. Nous présentons deux situations qui se présentent dans la pratique avec l'interaction du polymorphisme et du sous-typage, et nous proposons une solution unique pour traiter ces deux cas. Le premier cas, déjà mentionné par [32], est le typage des méthodes homogènes, c'est à dire, des méthodes qui acceptent plusieurs types (mais pas tous) comme arguments, tout en n'acceptant de les mélanger dans le même appel. Un exemple typique est l'opérateur de comparaison less, qui peut être appliqué à deux chaînes de caractères, deux nombres entiers, deux dates, etc, mais pas à deux valeurs de différents types, non plus qu'aux valeurs de types n'ayant pas d'ordre naturel, comme le composants graphiques. Le second cas, introduit dans cette thèse, est apparu au cours de notre pratique de langages avec multi-méthodes et basés sur le système de types polymorphes contraints $ML_{\leq}$ [6]. Nous avons remarqué que de nombreuses méthodes sont partiellement polymorphiques: leurs types ont une précision intermédiaire entre un type monomorphe et un type polymorphe contraint. Nous proposons de traiter ces deux situation en introduisant une notion de *kind*, c'est à dire, une propriété que certains types possèdent. Un avantage supplémentaire de notre solution est qu'elle est *modulaire* dans le sens où il est possible d'ajouter dans une hiérarchie de classes existantes de nouvelles classes possédant un certain kind sans changer le type des opérations qui concernent ce kind.

Présentation modulaire

Pour formaliser nos réponses à ces défis, nous devons définir un langage complet les incorpore de façon cohérente, et prouver ses propriétés. Nous attachons une attention spéciale à la manière de présenter cette formalisation. Puisque le paradigme fonctionnel est mieux compris théoriquement que le paradigme orienté objets, notre approche est essentiellement de partir d'un noyau fonctionnel et d'ajouter l'orientation objet, c'est à dire les définitions de classes et de multi-méthodes. Toutefois, nous ne nous basons pas directement sur Core-ML. Une raison pour ceci est que le système de types de Hindley et Milner ne contient pas le sous-typage atomique, qui est nécessaire à notre approche de l'orientation objet. De plus, la combinaison du polymorphisme paramétrique et du sous-typage atomique donne lieu à de nouveaux défis de typage. Ceux-ci peuvent être relevés en enrichissant encore le système de types. Il est probable que de nouvelles extensions deviennent nécessaires par la suite. Ces extensions peuvent prendre la forme soit de nouvelles opérations dans la sémantique du langage, de nouveaux types, ou des deux en même temps. En conséquence, une présentation monolithique directe aurait deux inconvénients. D'une part à cause de sa taille: elle serait difficile à comprendre. Mais surtout, la conception d'extensions deviendrait de plus en plus complexe puisque chacune nécessiterait une nouvelle présentation du système entier et une nouvelle preuve de correction.

Au contraire, nous choisissons de rechercher une présentation aussi modulaire que possible. Ainsi, dans la première partie (chapitre 1), nous présentons un *système de types algébriques*. Il comprend un langage noyau avec constantes, et ces types sont purement abstraits. Ce noyau permet de construire un langage complet en choisissant les types concrets et les constants d'expressions, qui doivent vérifier certaines propriétés garantissant la correction du langage. Celle construction nous permet d'étudier plus tard indépendamment nos propositions de systèmes de types (en particulier avec les kinds) et ceux concernant les mécanismes de

langage (orientation objet, multi-méthodes et leur typage modulaire). Dans le chapitre 2, nous montrons en particulier que deux systèmes de types existants, celui de Hindley et Milner d'une part et $ML_{\leq}$ d'autre part, peuvent être exprimés comme des instances de notre système de types algébriques.

Dans la seconde partie, nous présentons les concepts orientés objets dans ce cadre: les classes dans le chapitre 3, les multi-méthodes dans le chapitre 4, et les appels aux implémentation précédentes de méthodes ("super") dans le chapitre 5. Dans le chapitre 6, nous montrons l'intérêt d'une extension originale des types $ML_{\leq}$ pour typer plus précisément certaines méthodes grâce à l'ajout de *kinds*.

La troisième partie concerne la modularité. Dans le chapitre 8, nous montrons comment les déclarations de méthodes peuvent être faites dans un cadre modulaire, et comment les typer, indépendamment de l'algèbre de types. Nous appliquons ensuite cette approche dans le cas de l'algèbre de types $ML_{\leq}$. Le chapitre 9 traite de l'interaction entre appel aux méthodes antérieures et modules. Le chapitre 10 formalise notre système de kinds dans un cadre modulaire.

Mise en pratique: le langage Nice

En marge de cette présentation théorique, j'ai implémenté un langage généraliste complet fondé sur le principes présents dans cette thèse. Dans la quatrième partie, nous explorons certains aspects révélés par cette mise en pratique. Le chapitre 11 décrit une compilation de notre langage vers un langage de bas niveau, avec typage monomorphe, semblable au bytecode Java. Nous prouvons la correction de cette compilation. Le chapitre 12 décrit comment notre système de type avec kinds peut être implémenté. Le chapitre 13 présente le langage Nice, en détaillant les différences entre la syntaxe utilisée dans cette thèse et celle implémentée concrètement. Dans le chapitre 14, nous étudions le cas classique du "problème des expressions". Nous y proposons une solution utilisant notre système avec multi-méthodes modulaires, et nous la comparons avec d'autres solutions. Enfin, nous comparons différents aspects de notre travail avec dans travaux antérieurs dans le chapitre 15.

Introduction

We may define programming as the human activity consisting in turning a specification into a program. In this context, a program is a formal description of a task that can be executed automatically by a computer. A specification is a higher-level description that specifies a task. Specifications can vary in formality, from a vague description like "a program to read emails" to a complete description of the program's behavior. For the sake of clarity, we insist the programming is performed by humans. One could be tempted to also call specification a formal but high-level description that can be executed by a computer, for instance if a program can be automatically derived from it. However, we can then view such a specification itself as a program. It must have been written from a preceding, higher-level description of the task, which is the one we will call the specification. The fact that programs are written by humans is important because it conditions the criteria that should be used to evaluate the impact of programming languages design.

It is a well-known fact that all general purpose programming languages are equivalent, in the sense that for any program written in one of those languages and performing a certain task, and for any other language, there exists a program in that language that performs the same task. However, this does not mean that the idea of improving existing programming languages — or to create new ones — is pointless. Rather, programming languages should be designed to facilitate programming. We can identify four criteria.

1. Programs need to correctly implement the specification. In particular, the execution of a program should not reach an invalid state where it "crashes" or needs to be stopped before having finished its task. A language that has the property of static safety makes it possible to guarantee that a certain program will never run into this situation. Moreover, a language whose programs can be equipped with statically enforceable properties (types, logical assertions, ...) makes it easier to automatically check that the program conforms with (a possibly weakened form of) the specification. The richer those properties are, the more closely they can express the specification.

2. It should be possible to write programs in a "simple" way. Since simplicity is difficult to formalize, this criteria is difficult to evaluate. Typically, conciseness (by not needing to specify low-level operations) and the ease with which programs can be understood by other programmers are indicators of simplicity. When a formal specification exists, simplicity can be characterized by a small gap from the specification to the program.

3. Since large programs need to be written by teams of programmers, it should be possible to write programs modularly. That is, the main task should be decomposable into smaller ones that can be implemented as program parts called modules. Each module should be implementable with only minimal knowledge about the other modules it depends on.

4. To save efforts, one should be able to share generally useful modules between different programs [26]. This criteria builds on the previous one but goes beyond it for two reasons. First, reuse is often difficult because each program might need a slightly different behavior in a certain case than what the the shared module offers, or the program includes specific cases in addition to those present in the general module. Second, since the reused module was developed independently of the client programs, one cannot assume that it is possible to modify the shared module to accommodate for the specific needs of the program. Therefore, the language should offer mechanisms to customize and extend an imported module.

Our general motivation in this thesis is to identify situations where these criteria are not met by current programming languages, to propose language features to handle those situations, to prove that these features have good properties and to illustrate how they can be used in practice.

In the domain of programming language research, the most widely accepted basis is Robin Milner's Meta-Language (ML). The theory of this language is well-understood, which makes it an ideal basis to build extensions upon. Furthermore, it has appealing properties, like the ability to infer types and to manipulate functions as first class values. Therefore, many current research languages are extensions of ML. However, this research had a real but limited impact on the mainstream programming languages. Traditionally based on the imperative paradigm, those have in the recent years been extended to handle the object-oriented paradigm. This extension was motivated by the need to better support the design of large systems and to provide more expressiveness. These goals could also have been met by using the function paradigm of ML, although in a different way. It probably did not happen that way because object-orientation could be more easily presented as an extension of imperative programming (the first mainstream object-oriented programming language, C++, was backward compatible with C). In any case, it is interesting to compare how object-orientation and functional programming propose to solve similar problems, and how they meet our four criteria. Ultimately, this can help devising new languages or paradigms that better meet those criteria.

Multi-methods

It has already been recognized that the activity of programming has two main facets: defining operations and defining data structures [19]. Therefore, a programming paradigm must provide ways to express these two aspects. The functional paradigm mainly uses sum and product types as its data structures, and functions defined by pattern-matching on data-types as its operations. The object-oriented paradigm provides classes to structure data, and methods to operate on it. However, both paradigms introduce an asymmetry between the two aspects. In a functional program, data-types can be defined independently of functions, while functions need knowledge about the data-type constructors. Conversely, in the object-oriented paradigm, methods are defined locally to a class, while classes include the list of all their methods.

This asymmetry is problematic when it comes to our last two criteria, modularity and extensibility. Following the above dualism, we need both to define new operations on existing data structures and to define new data structures to be handled by existing operations. In the functional paradigm, defining new functions is straightforward. On the other hand, extending existing data-types is not possible since it would break existing functions defined by pattern matching on this type: they would miss the new cases. Conversely, extending data structures amounts to writing new classes, which is precisely the object-oriented paradigm, while defining new methods on existing classes is not allowed.

The fact that each paradigm privileges a different aspect explains why they are viewed as the two major competitors. However, as the modularity issue shows, both choices carry inconveniences. Therefore it is interesting to see how the conflict can be resolved. It involves making both operation and data definitions first-class, toplevel operations. This creates the need for a third concept: the implementation of an operation for a certain data type. This new paradigm initially appeared in CLOS, using object-oriented terms like methods and classes. Such methods are called *multi-methods*. Compared to traditional object-orientation, this new approach adds one aspects of functional programming: the possibility to add a new operation, defined by "pattern-matching" on multiple arguments. However, this approach had not yet been presented in a setting that includes more aspects of functional programming, like generic polymorphism and type inference. Furthermore, an additional challenge is to perform the type-checking of programs with multi-methods in a modular fashion. This is one of the main goals of this thesis.

Kinds

Another aspect of our work is to motivate and formalize a type-system extension to improve on the first criteria of stronger static checking. We present two typing challenges that arise in practice from the interplay

of polymorphism and subtyping, and we propose a single solution to solve them both. The first challenge, which has already been pointed out [32], is the typing of homogeneous methods, that is, methods that accept several (but not all) types for their arguments, while these types cannot be intermixed. A typical example is the comparison operator less, which can be applied to two strings, two integers, two dates, etc, but not to two values of different types, and neither to types that have no canonical ordering like graphical widgets. The second challenge, introduced in this thesis, has arisen from our experience with programming in languages with multi-methods and based on the polymorphic constrained type system $ML_{\leq}$ [6]. We found out that many useful methods are partially polymorphic: their types lie in precision in between a monomorphic and a bounded polymorphic type. We propose to handle those two situations by introducing the notion of *kind*, that is, a property that some types possess. An extra benefit of our solution is to be *modular* in the sense that new classes with a certain kind can be added to an existing class hierarchy without changing the type of the operations that act on types of that kind.

Modular presentation

To formalize our solutions to these challenges, we need to define a complete language that incorporates them in a coherent way and to prove its properties. We take special care in the way we expose this formalization. Since functional programming is better understood theoretically than object orientation, our approach is in essence to start from there, and to add object-oriented features, that is, class definitions and multi-methods. However, we will not base our work on Core-ML. One reason is that Hindley and Milner's type system does not include atomic subtyping, which is necessary for our approach of object-orientation. Furthermore, the combination of generic polymorphism and object-oriented subtyping gives rise to new typing challenges. Those can be solved by enriching the type system further. The need for other extensions is likely to arise. These extensions could take the form of either new operations in the semantics of the language, different types, or both at the same time. Therefore, a direct presentation would have two drawbacks. First because of its size: it would be very hard to comprehend. Second and probably more important, the task of designing the extensions would become increasingly complex, as each one would essentially need a new presentation of the whole system and a new proof of its correctness.

Instead, we chose to experiment with a presentation that is as modular as possible. Therefore, in the first part (Chapter 1), we present an *algebraic type system*. It includes a core language with constants, and its types are fully abstract. This core allows a real language to be built by choosing both concrete types and expression constants, which must verify some properties that guarantee the soundness of the full language. This construction allows us to study independently later on our proposals concerning type systems (in particular kinds) and those concerning language features (object-orientation, multi-methods and their modular typing). In Chapter 2, we show in particular that two existing type systems, Hindley and Milner's and $ML_{\leq}$, can be expressed as instances of our core type system.

In the second part, we present object-oriented features in this framework: classes in Chapter 3, multi-methods in Chapter 4, and calls to previous method implementations (*super* calls) in Chapter 5. In Chapter 6, we motivate an original extension of $ML_{\leq}$ types to type more methods, with the addition of *kinds*.

In the third part, we focus on modularity. In Chapter 8, we show how to declare methods in a modular setting, and how to typecheck them, independently of the type algebra. We also instantiate this feature in the case of the $ML_{\leq}$ type algebra. Chapter 9 discusses the interaction of super calls with modules. Chapter 10 formalizes our system of kinds in a modular setting.

Theory into practice: the Nice language

In parallel, I have implemented a full general-purpose programming language founded on the theory exposed in this dissertation. In the fourth part, we explore some of the aspects involved in this effort of putting this theory into practice. Chapter 11 describes a possible compilation of our language to a monorphically typed, lowlevel bytecode language similar to the Java bytecode, and proves the correctness of this compilation. Chapter 12 describes how the type system with kinds can be implemented. Chapter 13 introduces the

Nice language, detailing the differences between the syntax used in this document and the concrete syntax implemented. In Chapter 14, we study the classical *expression problem*. We propose a solution using our system with multi-methods, and we compare it with other solutions. Finally, we compare the different aspects of our work with related work in Chapter 15.

Part I

Fondations

Chapter 1

Algebraic type system

We present type-checking, type inference and soundness proofs for a core functional language. Instead of exposing the syntax and structure of types, we treat them as an abstract structure with three visible operations, following the structure of expressions: the construction of functional types, the application of one type to another, and the let binding of a type variable in a type. Note that this application is not the application of a type constructor to a type. It is a meta-operator that, given the type of a function f and the type of an argument v, returns the type of the expression $f\ v$. For instance, assume f and v have types $\texttt{int} \to \texttt{bool}$ and $\texttt{int}$ respectively. We shall give the expression $f\ v$ the type $(\texttt{int} \to \texttt{bool})\ \texttt{int}$. That is to say that $(\texttt{int} \to \texttt{bool})\ \texttt{int}$ is the type of the results of a function of type $\texttt{int} \to \texttt{bool}$ applied to a value of type $\texttt{int}$.

Since we want to reason about the construction of types, we shall distinguish the *algebraic types*, which are provided by the instantiation of the framework, and the *syntactic types*, which are constructed by our type system on top of the algebraic types. The instantiation of the system must provide an interpretation of these syntactic types by providing a subtyping relation. Following up on the above example, we may assume for instance that algebraic types are ground types built over the constants `int` and `bool` and the arrow type constructor. That is `int`, `bool`, $\texttt{int} \to \texttt{bool}$ are algebraic types[1]. Conversely, $(\texttt{int} \to \texttt{bool})\ \texttt{int}$ is a syntactic type that is equivalent to `bool`, while $(\texttt{int} \to \texttt{bool})\ \texttt{bool}$ is equivalent to the error syntactic type.

1.1 Type algebra

The algebraic type system is parameterized by a *type algebra*. To define it, we first introduce the notion of syntactic types.

Syntactic type	τ	$::=$	
Algebraic type			a
Type variable		$\mid$	t
Functional type		$\mid$	$\lambda t.\tau$
Application type		$\mid$	$\tau\ \tau$
Let type		$\mid$	$\texttt{let}\ t\ \texttt{be}\ \tau\ \texttt{in}\ \tau$
Error type		$\mid$	$\mathbb{E}$

Figure 1.1: Syntactic types

[1] They happen to be also syntactic types, which contain all algebraic types, see Figure 1.1.

Definition 1 (Syntactic types) *Given an arbitrary set A, whose elements are denoted by a and called algebraic types, and given an infinite set of type variables denoted by t, the set of* syntactic types over A, *written $S(A)$, is defined by the grammar for syntactic types τ of Figure 1.1.*

Informally, $S(A)$ contains the expressions τ of a small calculus (similar to Core-ML) built over an infinite set of type variables t and over constants a of A. Additionally, an error type $\mathbb{E}$ is distinguished. Note that syntactic types are really a piece of syntax. In particular, there is no β-reduction on syntactic types. However, we will see in Section 1.2.1 that a parallel can be drawn between syntactic types and lambda-expressions. In the following, we use "type" as a shorthand for syntactic type.

In the type $\lambda t.\tau$, λ acts as a binder of variable t with scope τ. In the type $\texttt{let}\ t\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2$, $\texttt{let}$ acts as a binder of variable t with scope τ_2. The set of free variables of a type τ (Definition 2), written $FV(\tau)$, and the capture-free substitution of type τ for type variable t in type τ_0 (Definition 3), written $\tau_0\,[t \leftarrow \tau]$, are defined as usual. Syntactic types are equal up to α-conversion. That is, we do not distinguish between $\lambda t_1.\tau$ and $\lambda t_2.\tau\,[t_1 \leftarrow t_2]$ when t_2 does not belong to $FV(\tau)$. Similarly, $\texttt{let}\ t_1\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2$ and $\texttt{let}\ t_2\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2\,[t_1 \leftarrow t_2]$ are the same type when t_2 does not belong to $FV(\tau_2)$.

Definition 2 (Free variables of a syntactic type) *The set of the free variables of a syntactic type is defined inductively by:*

$$\begin{aligned}
FV(a) &= \emptyset \\
FV(t) &= \{t\} \\
FV(\lambda t.\tau) &= FV(\tau) \setminus \{t\} \\
FV(\tau_1\ \tau_2) &= FV(\tau_1) \cup FV(\tau_2) \\
FV(\texttt{let}\ t\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2) &= FV(\tau_1) \cup \big(FV(\tau_2) \setminus \{t\}\big) \\
FV(\mathbb{E}) &= \emptyset
\end{aligned}$$

Definition 3 (Substitution) *The substitution of type τ for type variable t in type τ_0, written $\tau_0\,[t \leftarrow \tau]$, is defined inductively by:*

$$\begin{aligned}
a\,[t \leftarrow \tau] &= a \\
t\,[t \leftarrow \tau] &= \tau \\
t'\,[t \leftarrow \tau] &= t' && (t' \neq t) \\
(\lambda t_1.\tau_1)\,[t \leftarrow \tau] &= \lambda t_1.(\tau_1\,[t \leftarrow \tau]) && (t_1 \neq t, t_1 \notin FV(\tau)) \\
(\tau_1\ \tau_2)\,[t \leftarrow \tau] &= (\tau_1\,[t \leftarrow \tau])\ (\tau_2\,[t \leftarrow \tau]) \\
(\texttt{let}\ t_1\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2)\,[t \leftarrow \tau] &= \texttt{let}\ t_1\ \texttt{be}\ (\tau_1\,[t \leftarrow \tau])\ \texttt{in}\ (\tau_2\,[t \leftarrow \tau]) && (t_1 \neq t, t_1 \notin FV(\tau)) \\
\mathbb{E}\,[t \leftarrow \tau] &= \mathbb{E}
\end{aligned}$$

For functional and let types, the side condition can always be satisfied by renaming the bound variable.

It is easy to check that Definition 3 and Definition 2 are valid, since they do not depend on the name chosen for the bound variable of functional and let types. Furthermore, the substitution can be extended to a total function by renaming of the bound variables whenever the side conditions are not satisfied.

The interest of syntactic types lies in representing the possible ways in which the types of expressions of a programming language can be combined to form the type of a larger expression. Syntactic types need to be interpreted, so as to provide information about the corresponding expressions. This interpretation is provided by a *type algebra*.

Definition 4 (Type algebra) *A* type algebra $\mathcal{A}$ *is a couple $(A, \leq)$, where A is a set of algebraic types, and the relation $\leq$ is a pre-order on $S(A)$ such that the following four axioms are satisfied:*

i. *(Error) The type $\mathbb{E}$ is a maximal element. That is, for all type τ, $\tau \leq \mathbb{E}$ holds. Moreover, for all type τ and type variable t, $\mathbb{E} \leq \mathbb{E}\ \tau$, $\mathbb{E} \leq \tau\ \mathbb{E}$ and $\mathbb{E} \leq \texttt{let}\ t\ \texttt{be}\ \mathbb{E}\ \texttt{in}\ \tau$ hold.*

ii. *(Covariance) Syntactic types are covariant. That is, for all types τ, τ', τ_0, and type variable t, if $\tau' \leq \tau$ holds, then $\tau_0\,[t \leftarrow \tau'] \leq \tau_0\,[t \leftarrow \tau]$ holds.*

iii. *(Reduction) For all types τ, τ' and type variable t, $\tau\,[t \leftarrow \tau'] \leq (\lambda t.\tau)\ \tau'$*

iv. *(Let) For all types τ, τ' and type variable t, $\tau'\,[t \leftarrow \tau] \leq$ `let` t `be` τ `in` τ'*

We discuss the interpretation of these axioms in the next section, where an expression language is introduced. In particular, the maximality of $\mathbb{E}$ is motivated before Definition 8, and a parallel is drawn between these axioms and the reduction rules for the expression language in Section 1.2.1.

When $\tau_1 \leq \tau_2$ holds, we will say that τ_1 is a subtype of τ_2. Two syntactic types τ_1 and τ_2 are equivalent if both $\tau_1 \leq \tau_2$ and $\tau_2 \leq \tau_1$ hold, in which case we write $\tau_1 \equiv \tau_2$. When $\tau_1 \equiv \tau_2$ does not hold, we will write $\tau_1 \not\equiv \tau_2$. The relation $\geq$ is the symmetric relation of $\leq$. The predicate $\tau_1 < \tau_2$ holds when $\tau_1 \leq \tau_2$ holds but $\tau_1 \geq \tau_2$ does not hold. Similarly, $\tau_1 > \tau_2$ holds when $\tau_1 \geq \tau_2$ holds but $\tau_1 \leq \tau_2$ does not hold.

Note that while we don't specifically require a subtyping relation for the algebraic types, they can be compared by the restriction of $\leq$ to the set of algebraic types A. When there is ambiguity about the type algebra in which syntactic types are compared, we write $\mathcal{A} \models \tau_1 \leq \tau_2$ instead of $\tau_1 \leq \tau_2$.

1.1.1 Sub-algebras

We show that a subset of a type algebra is also a type algebra.

Theorem 5 (Sub-algebra) *Let $(A, \leq)$ be a type algebra, and A' be a subset of A. Then $(A', \leq)$ is a type algebra.*

Proof of theorem 5

First, by Definition 1, since A' is a subset of A, $S(A')$ is a subset of $S(A)$. Therefore $\leq$, which is a pre-order on $S(A)$, is also a pre-order on $S(A')$. Finally, all four conditions on $\leq$ in Definition 4 hold for all types in $S(A)$, so they do hold in particular for those in $S(A')$. ■

1.1.2 Example

Let us build a type algebra, based on the following simple monomorphic type system with atomic subtyping and function types. The algebraic types A are defined by the grammar $a ::=$ `int` $\mid$ `float` $\mid a \to a$. These types are ordered as usual, by the smallest relation $\prec$ that verifies:

$$\texttt{int} \prec \texttt{float} \qquad\qquad \frac{a_2 \prec a_2' \qquad a_1' \prec a_2}{a_2' \to a_1' \prec a_2 \to a_1}$$

We must provide a pre-order on $S(A)$ that verifies the axioms of Definition 4. To this end, we introduce an auxiliary definition. We define a translation $\langle\cdot\rangle$ of the elements of $S(A)$ to sets of algebraic types, which maps syntactic types to the set of types they denote.

$$\begin{aligned}
\langle a \rangle &= \{a' \mid a \prec a'\} \\
\langle t \rangle &= A \\
\langle \mathbb{E} \rangle &= \emptyset \\
\langle \tau_1\ \tau_2 \rangle &= \{a_1 \mid \exists a_2 \in \langle \tau_2 \rangle,\ a_2 \to a_1 \in \langle \tau_1 \rangle\} \\
\langle \lambda t.\tau \rangle &= \{\, a_2 \to a_1 \mid a_2 \in A, a_1 \in \langle \tau\,[t \leftarrow a_2] \rangle\} \\
\langle \texttt{let}\ t\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2 \rangle &= \{a_2 \mid \exists a_1 \in \langle \tau_1 \rangle, a_2 \in \langle \tau_2\,[t \leftarrow a_1] \rangle\}
\end{aligned}$$

For instance, $\langle \texttt{int} \rangle = \{\texttt{int}, \texttt{float}\}$ and $\langle \lambda t.t \rangle = \{\, a \to a \mid a \in A\}$. Therefore, the translation of $(\lambda t.t)$ `int` is $\{\texttt{int}, \texttt{float}\}$.

The order on syntactic types is defined in the following way: $\tau_1 \leq \tau_2$ holds if and only if $\langle \tau_1 \rangle \supseteq \langle \tau_2 \rangle$.

We can now show that $(A, \leq)$ is indeed a type algebra. The relation $\leq$ is a pre-order on $S(A)$ since $\supseteq$ is reflexive and transitive. Moreover, it satisfies the axioms of Definition 4:

1. $\langle \mathbb{E} \rangle = \emptyset$. Therefore, for all t and τ, $\langle \tau \rangle \supseteq \langle \mathbb{E} \rangle$, and $\langle \mathbb{E}\ \tau \rangle = \langle \tau\ \mathbb{E} \rangle = \langle \texttt{let}\ t\ \texttt{be}\ \mathbb{E}\ \texttt{in}\ \tau \rangle = \emptyset$. That is, $\tau \leq \mathbb{E}$, $\mathbb{E} \leq \mathbb{E}\ \tau$, $\mathbb{E} \leq \tau\ \mathbb{E}$ and $\mathbb{E} \leq \texttt{let}\ t\ \texttt{be}\ \mathbb{E}\ \texttt{in}\ \tau$ hold.

2. Covariance is proved by induction on the structural size of τ_0. We need to prove that $\tau_0\,[t \leftarrow \tau'] \leq \tau_0\,[t \leftarrow \tau]$, provided that $\tau' \leq \tau$. For the case $\tau_0 = \lambda t_1.\tau_1$, we can assume w.l.o.g that t_1 is different from t and not free in τ nor in τ'. Therefore $(\lambda t_1.\tau_1)\,[t \leftarrow \tau] = \lambda t_1.(\tau_1\,[t \leftarrow \tau])$ and $(\lambda t_1.\tau_1)\,[t \leftarrow \tau'] = \lambda t_1.(\tau_1\,[t \leftarrow \tau'])$. For $a_2 \to a_1$ in $\langle \lambda t_1.\tau_1\,[t \leftarrow \tau] \rangle$, by definition of the translation, $a_1 \in \langle \tau_1\,[t \leftarrow \tau]\,[t_1 \leftarrow a_2] \rangle$. Since $t_1 \neq t$ and t_1 is not free in τ, $\tau_1\,[t \leftarrow \tau]\,[t_1 \leftarrow a_2] = \tau_1\,[t_1 \leftarrow a_2]\,[t \leftarrow \tau]$. Since $\tau_1\,[t_1 \leftarrow a_2]$ has a smaller size than $\lambda t_1.\tau_1$, we can apply the induction hypothesis, and $\langle \tau_1\,[t_1 \leftarrow a_2]\,[t \leftarrow \tau] \rangle \subseteq \langle \tau_1\,[t_1 \leftarrow a_2]\,[t \leftarrow \tau'] \rangle$. Again, $\tau_1\,[t_1 \leftarrow a_2]\,[t \leftarrow \tau'] = \tau_1\,[t \leftarrow \tau']\,[t_1 \leftarrow a_2]$. Therefore $a_1 \in \langle \tau_1\,[t \leftarrow \tau']\,[t_1 \leftarrow a_2] \rangle$. That is, $a_2 \to a_1 \in \langle \lambda t_1.\tau_1\,[t \leftarrow \tau'] \rangle$. Thus, $\lambda t_1.\tau_1\,[t \leftarrow \tau'] \leq \lambda t_1.\tau_1\,[t \leftarrow \tau]$. The case of let types is similar, and the other cases are straightforward.

3. We now prove an auxiliary property: the translation sets are upward-closed. That is, for all algebraic types a and a', and syntactic type τ, if $a \in \langle \tau \rangle$ and $a \prec a'$, then $a' \in \langle \tau \rangle$. The proof is by induction on τ. All cases are immediate except for functional types. For all a in $\langle \lambda t.\tau \rangle$, by definition of the translation, a is of the form $a_2 \to a_1$ where $a_1 \in \langle \tau\,[t \leftarrow a_2] \rangle$. By hypothesis $a \prec a'$ and by definition of $\prec$, a' is of the form $a'_2 \to a'_1$ with $a'_2 \prec a_2$ (**1**) and $a_1 \prec a'_1$ (**2**). By (2) and the induction hypothesis, $a'_1 \in \langle \tau\,[t \leftarrow a_2] \rangle$ (**3**). Furthermore, it follows from (1) that $a'_2 \leq a_2$ holds. Therefore, by the covariance axiom proved above, $\tau\,[t \leftarrow a'_2] \leq \tau\,[t \leftarrow a_2]$. That is, by definition of subtyping, $\langle \tau\,[t \leftarrow a'_2] \rangle \supseteq \langle \tau\,[t \leftarrow a_2] \rangle$. Therefore, (3) implies $a'_1 \in \langle \tau\,[t \leftarrow a'_2] \rangle$. That is, $a' \in \langle \lambda t.\tau \rangle$, which proves the property. This property implies in particular that for all algebraic type a and syntactic type τ, if $a \in \langle \tau \rangle$, then $a \geq \tau$ (**4**).

   We may now prove the reduction axiom: for all types τ, τ' and type variable t, $(\lambda t.\tau)\ \tau' \geq \tau\,[t \leftarrow \tau']$. Let a_1 be in $\langle (\lambda t.\tau)\ \tau' \rangle$. Then there exists a_2 in $\langle \tau' \rangle$ such that $a_2 \to a_1 \in \langle \lambda t.\tau \rangle$. That is, $a_1 \in \langle \tau\,[t \leftarrow a_2] \rangle$. Since $a_2 \in \langle \tau' \rangle$, by (4) we have that $a_2 \geq \tau'$. So by covariance, $\tau\,[t \leftarrow a_2] \geq \tau\,[t \leftarrow \tau']$. That is, $\langle \tau\,[t \leftarrow a_2] \rangle \subseteq \langle \tau\,[t \leftarrow \tau'] \rangle$. Therefore, a_1 is in $\langle \tau\,[t \leftarrow \tau'] \rangle$, which shows the property.

4. For all types τ, τ' and type variable t, $\texttt{let}\ t\ \texttt{be}\ \tau\ \texttt{in}\ \tau' \geq \tau'\,[t \leftarrow \tau]$. Let a_2 be an element of $\langle \texttt{let}\ t\ \texttt{be}\ \tau\ \texttt{in}\ \tau' \rangle$. Then by definition, there exists a_1 in $\langle \tau_1 \rangle$ such that $a_2 \in \langle \tau_2\,[t \leftarrow a_1] \rangle$. By (4), $a_1 \geq \tau_1$. So by covariance, $\tau_2\,[t \leftarrow a_1] \geq \tau_2\,[t \leftarrow \tau_1]$. That is, $\langle \tau_2\,[t \leftarrow a_1] \rangle \subseteq \langle \tau_2\,[t \leftarrow \tau_1] \rangle$. Therefore, a_2 is in $\langle \tau_2\,[t \leftarrow \tau_1] \rangle$, which shows the property.

It is interesting to see in this algebra the meaning of Covariance (Definition 4.**ii**). In particular, type application is covariant on both arguments. This is not in contradiction with the contra-variance of the $\to$ algebraic type constructor on its first argument. For instance, $\texttt{float} \to \texttt{int} \leq \texttt{int} \to \texttt{float}$. Therefore, it is required by Covariance (Definition 4.**ii**) that $(\texttt{float} \to \texttt{int})\ \texttt{int} \leq (\texttt{int} \to \texttt{float})\ \texttt{int}$. This indeed holds, since in the translation, $\langle (\texttt{float} \to \texttt{int})\ \texttt{int} \rangle$ is $\{\texttt{int}, \texttt{float}\}$ and $\langle (\texttt{int} \to \texttt{float})\ \texttt{int} \rangle$ is $\{\texttt{float}\}$.

1.2 Core language

Expression	e	::=	$x \mid \lambda x.e \mid e\ e \mid \texttt{let}\ x\ \texttt{be}\ e\ \texttt{in}\ e \mid c$	
Constant	c	::=	$C \mid f$	
Value	v	::=	$\lambda x.e$	
		$\mid$	$C\ v_1\ ...\ v_n$	$(n \leq \mathsf{arity}(C))$
		$\mid$	$f\ v_1\ ...\ v_n$	$(n < \mathsf{arity}(f))$

Figure 1.2: Language syntax

We consider the set of Core-ML expressions, recalled in Figure 1.2. Lambda abstractions $\lambda x.e$ and let expressions $\texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e$ bind they argument x in their body e, and are considered equal modulo

renaming of x. These expressions are parameterized by a set of *constants* c. Each constant comes with an algebraic type written $\mathsf{constant\text{-}type}(c)$, and with a positive integer arity written $\mathsf{arity}(c)$. Constants are either *data constructors* C or *operators* f. A constant c applied to n arguments is a value when n is less than $\mathsf{arity}(c)$, or c is a data constructor and n is $\mathsf{arity}(c)$. Thus, an operator f applied to exactly $\mathsf{arity}(f)$ arguments is not a value (and therefore it must be reduced).

The semantics of expressions is defined in Figure 1.3. This notion of reduction encompasses both call-by-value and call-by-name. A deterministic restriction is presented in Section 1.2.2. The β and β-LET reduction rules are standard. The (possibly non-deterministic) reductions of an operator f are defined by the set of $(\mathsf{arity}(f)+1)$-tuples $R(f)$ used by Rule OP. Rule CTXT allows reductions to occur inside expressions, except in the body of a let-expression or in the body of a function. Note that it does not change semantics whether contexts have depth one as defined here or an arbitrary depth; it is always possible to apply CTXT several times to obtain reductions at an arbitrary depth inside an expression.

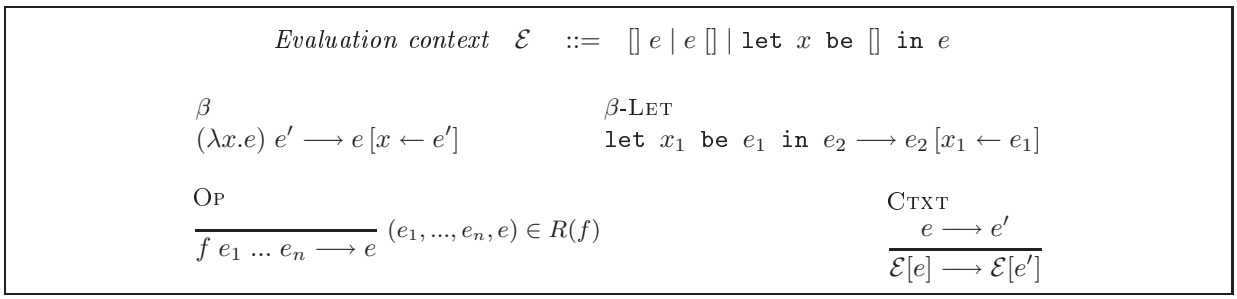


Figure 1.3: General semantics

VAR	$\mathsf{type}(x)$	=	t_x
CST	$\mathsf{type}(c)$	=	$\mathsf{constant\text{-}type}(c)$
LAM	$\mathsf{type}(\lambda x.e)$	=	$\lambda t_x.\mathsf{type}(e)$
APP	$\mathsf{type}(e_1\ e_2)$	=	$\mathsf{type}(e_1)\ \mathsf{type}(e_2)$
LET	$\mathsf{type}(\texttt{let}\ x_1\ \texttt{be}\ e_1\ \texttt{in}\ e_2)$	=	$\texttt{let}\ t_{x_1}\ \texttt{be}\ \mathsf{type}(e_1)\ \texttt{in}\ \mathsf{type}(e_2)$

Figure 1.4: The typing function

We define a function computing the syntactic type of any expression. Instead of using a type environment, we associate to each free variable x of the expression a distinct free type variable t_x. This typing function is defined recursively in Figure 1.4. Expression variables are mapped to their associated type variables (Rule VAR). Rule CST gives Constants their declared type. Note that it is valid to use $\mathsf{constant\text{-}type}(c)$ as a syntactic type, since the grammar of Figure 1.1 implies that algebraic types are also syntactic types. The type of a lambda abstraction $\lambda x.e$ is the syntactic functional type $\lambda t_x.\tau$, where τ is the type of e (rule LAM). Since syntactic types are equal modulo α-conversion, we show in Proposition 7 that the type of a lambda abstraction does not depend on the choice of its parameter name. The type of an application $e_1\ e_2$ is the syntactic application of the types of e_1 and e_2 (rule APP). Finally, in Rule LET, the type of a let-expression $\texttt{let}\ x_1\ \texttt{be}\ e_1\ \texttt{in}\ e_2$ is the type that binds t_{x_1} to $\mathsf{type}(e_1)$ in $\mathsf{type}(e_2)$. Again, this type does not depend on the choice of the name x_1 by alpha-conversion of let types.

We prove a substitution lemma that describes the properties of the typing function. It says that substitution commutes with the typing function. An interpretation of this result is that the type of an expression is a function of the type of all its parts. Moreover, because of the covariance of type algebras (Definition 4.**ii**), that function is covariant in all its arguments.

Lemma 6 (Substitution)
For all variable x and for all expressions e and e',

$$\mathsf{type}(e\,[x \leftarrow e']) = \mathsf{type}(e)\,[t_x \leftarrow \mathsf{type}(e')]$$

Proof of lemma 6 (Substitution)

The proof is by induction on e.

case $e = x$

By definition of the typing function, $\mathsf{type}(x)$ is t_x. Therefore, $\mathsf{type}(x)\,[t_x \leftarrow \mathsf{type}(e')]$ is $t_x\,[t_x \leftarrow \mathsf{type}(e')]$, which by Definition 3 is equal to $\mathsf{type}(e')$.

By definition of substitution, $x\,[x \leftarrow e'] = e'$. Therefore $\mathsf{type}(x\,[x \leftarrow e'])$ is also equal to $\mathsf{type}(e')$, which proves the property.

case $e = x'$ **with** $x' \neq x$

By definition of substitution, $x'\,[x \leftarrow e'] = x'$. Therefore $\mathsf{type}(x'\,[x \leftarrow e']) = \mathsf{type}(x') = t_{x'}$. Since x' is different from x, this implies $t_{x'} \neq t_x$ and therefore $t_{x'}$ is in turn equal by Definition 3 to $t_{x'}\,[t_x \leftarrow \mathsf{type}(e')]$.

case $e = c$

This case is similar to the case $e = x'$.

case $e = \lambda x_1.e_1$

By α-conversion on e, we can assume without loss of generality that $x_1 \neq x$ **(1)** and that $x_1 \notin FV(e')$ **(2)**.

Let T be $\mathsf{type}((\lambda x_1.e_1)\,[x \leftarrow e'])$. By definition of substitution, (1) and (2), we have $(\lambda x_1.e_1)\,[x \leftarrow e'] = \lambda x_1.(e_1\,[x \leftarrow e'])$. Therefore, T is equal to $\mathsf{type}(\lambda x_1.(e_1\,[x \leftarrow e']))$, which by Rule LAM is equal to $\lambda t_{x_1}.\mathsf{type}(e_1\,[x \leftarrow e'])$.

By induction hypothesis, $\mathsf{type}(e_1\,[x \leftarrow e'])$ is equal to $\mathsf{type}(e_1)\,[t_x \leftarrow \mathsf{type}(e')]$. Therefore, $\lambda t_{x_1}.\mathsf{type}(e_1\,[x \leftarrow e'])$ is equal to $\lambda t_{x_1}.(\mathsf{type}(e_1)\,[t_x \leftarrow \mathsf{type}(e')])$, which by Definition 3 with (1) is equal to $(\lambda t_{x_1}.\mathsf{type}(e_1))\,[t_x \leftarrow \mathsf{type}(e')]$. Since by Rule LAM, $\mathsf{type}(\lambda x_1.e_1) = \lambda t_{x_1}.\mathsf{type}(e_1)$, and we finally have $T = \mathsf{type}(\lambda x_1.e_1)\,[t_x \leftarrow \mathsf{type}(e')]$, which finishes the proof of this case.

case $e = e_1\ e_2$

By definition of substitution, $(e_1\ e_2)\,[x \leftarrow e'] = e_1\,[x \leftarrow e']\ e_2\,[x \leftarrow e']$.

This case is then a mere application of the induction hypothesis $\mathsf{type}(e_i\,[x \leftarrow e']) = \mathsf{type}(e_i)\,[t_x \leftarrow \mathsf{type}(e')]$ **(1)** for $i = 1, 2$, combined with the definition of the typing function for application types.

$$\begin{array}{lll}
 & \mathsf{type}((e_1\,[x \leftarrow e'])\ (e_2\,[x \leftarrow e'])) & \\
= & \mathsf{type}(e_1\,[x \leftarrow e'])\ \mathsf{type}(e_1\,[x \leftarrow e']) & (\textsc{App}) \\
= & (\mathsf{type}(e_1)\,[t_x \leftarrow \mathsf{type}(e')])\ (\mathsf{type}(e_2)\,[t_x \leftarrow \mathsf{type}(e')]) & \textbf{(1)} \\
= & (\mathsf{type}(e_1)\ \mathsf{type}(e_2))\,[t_x \leftarrow \mathsf{type}(e')] & (\text{Definition 3}) \\
= & \mathsf{type}(e_1\ e_2)\,[t_x \leftarrow \mathsf{type}(e')] & (\textsc{App})
\end{array}$$

case $e = \texttt{let}\ x_1\ \texttt{be}\ e_1\ \texttt{in}\ e_2$

By α-conversion, we can assume without loss of generality that $x_1 \neq x$ **(1)** and $x_1 \notin FV(e')$ **(2)**. Therefore, by definition of the substitution, $(\texttt{let}\ x_1\ \texttt{be}\ e_1\ \texttt{in}\ e_2)\,[x \leftarrow e'] = \texttt{let}\ x_1\ \texttt{be}\ e_1\,[x \leftarrow e']\ \texttt{in}\ e_2\,[x \leftarrow e']$.

$$\begin{array}{lll}
 & \mathsf{type}(\texttt{let}\ x_1\ \texttt{be}\ e_1\,[x \leftarrow e']\ \texttt{in}\ e_2\,[x \leftarrow e']) & \\
= & \texttt{let}\ t_{x_1}\ \texttt{be}\ \mathsf{type}(e_1\,[x \leftarrow e'])\ \texttt{in}\ \mathsf{type}(e_2\,[x \leftarrow e']) & (\textsc{Let}) \\
= & \texttt{let}\ t_{x_1}\ \texttt{be}\ \mathsf{type}(e_1)\,[t_x \leftarrow \mathsf{type}(e')]\ \texttt{in}\ \mathsf{type}(e_2)\,[t_x \leftarrow \mathsf{type}(e')] & (\text{Ind. Hyp.}) \\
= & (\texttt{let}\ t_{x_1}\ \texttt{be}\ \mathsf{type}(e_1)\ \texttt{in}\ \mathsf{type}(e_2))\,[t_x \leftarrow \mathsf{type}(e')] & (\text{Definition 3 with } \textbf{(1)}) \\
= & \mathsf{type}(\texttt{let}\ x_1\ \texttt{be}\ e_1\ \texttt{in}\ e_2)\,[t_x \leftarrow \mathsf{type}(e')] & (\textsc{Let})
\end{array}$$

■

Proposition 7 *Let x and x' be distinct variables and e an expression such that x' is not in $FV(E)$. Then* $\mathsf{type}(\lambda x.e) = \mathsf{type}(\lambda x'.e\,[x \leftarrow x'])$.

Proof of proposition 7

By the definition in Figure 1.4, $\mathsf{type}(\lambda x.e) = \lambda t_x.\mathsf{type}(e)$ and $\mathsf{type}(\lambda x'.e\,[x \leftarrow x']) = \lambda t_{x'}.\mathsf{type}(e\,[x \leftarrow x'])$. By Lemma 6 (SUBSTITUTION), $\mathsf{type}(e\,[x \leftarrow x'])$ is equal to $\mathsf{type}(e)\,[t_x \leftarrow t_{x'}]$. Therefore, $\mathsf{type}(\lambda x'.e\,[x \leftarrow x'])$ is equal to $\lambda t_{x'}.(\mathsf{type}(e)\,[t_x \leftarrow t_{x'}])$. Since x' is not free in e by hypothesis, $t_{x'}$ is not free in $\mathsf{type}(e)$. Therefore, by alpha-conversion, $\lambda t_{x'}.(\mathsf{type}(e)\,[t_x \leftarrow t_{x'}])$ is equal to $\lambda t_x.\mathsf{type}(e)$, which proves the property. ■

Since typing is done by a function, it associates exactly one type to every expression. Therefore, there is no notion of generalization or instantiation. Instead, in a type algebra with polymorphic types, types are always generalized as much as possible, and the syntactic types bound to by let types correspond to polymorphic types. In particular, this allows a let-bound value to be used polymorphically.

In most type systems, an expression is well-typed when it can be assigned some type according to the typing rules, and ill-typed expressions are those that can be assigned no type. In our framework, the typing function is independent of the concrete type system, that is the type algebra. Therefore, the typing function is more naturally defined as a total function that assigns a syntactic type to every expression. Ill-typed expressions are characterized by the fact that their type is equivalent to the error type. This convention is in agreement with the intuition that smaller types are inhabited by more values than greater types. A natural extension is to introduce an error type that is maximal (as required by Definition 4.**i**) and inhabited by no value. Therefore, ill-typed expressions are those whose type is inhabited by no values, that is, whose type is equivalent to the error type.

Definition 8 (Well-typed expression)
An expression e is well-typed *if and only if* $\mathsf{type}(e) \not\equiv \mathbb{E}$.

To ensure type soundness, we assume that the following requirements on constants are satisfied:

Requirement 9 (Constants) *Let n be* $\mathsf{arity}(f)$.

i. *If $f\ v_1\ ...\ v_n$ is well-typed then there exists an expression e' such that $(v_1, ..., v_n, e')$ belongs to $R(f)$.*

ii. *If $(e_1, ..., e_n, e')$ belongs to $R(f)$, then* $\mathsf{type}(f\ e_1\ ...\ e_n) \geq \mathsf{type}(e')$

iii. *An expression of the form $C\ v_1\ ...\ v_p$ is never well-typed if* $p > \mathsf{arity}(C)$

The first requirement guarantees that an operator has enough reductions to cover all of its legal arguments. The second one requires that all operator reductions lead to expressions with smaller types. Finally, a data constructor must not be applicable to more values than its arity, since no rule would guarantee that such an expression reduces to a value.

The type system verifies subject reduction with respect to the semantics of Figure 1.3. That is, along the paths of reduction, the type of expressions always get smaller.

Theorem 10 (Subject reduction)
For all expressions e and e',

$$\frac{e \longrightarrow e'}{\mathsf{type}(e) \geq \mathsf{type}(e')}$$

Although this theorem is standard, its proof is not. It can be done by using the typing functions and the two requirements on type algebras. This is mere calculation on types, as opposed to the creative reasoning usually required. This simplification is possible because we have a calculus of type expressions and because a part of the proof is abstracted away in Reduction (Definition 4.**iii**). The proof is thus short and easy to check step by step. It should furthermore ease automatic proof checking.

Proof of theorem 10 (Subject reduction)

The proof is by induction on the derivation proof of the reduction.

case OP

This case is exactly covered by Requirement 9 (CONSTANTS).

case β

$$\begin{array}{lll} & \mathsf{type}((\lambda x.e)\ e') & \\ = & \mathsf{type}(\lambda x.e)\ \mathsf{type}(e') & (\textsc{App}) \\ = & \lambda t_x.\mathsf{type}(e)\ \mathsf{type}(e') & (\textsc{Lam}) \\ \geq & \mathsf{type}(e)\,[t_x \leftarrow \mathsf{type}(e')] & (\text{Definition 4.}\mathbf{iii}) \\ = & \mathsf{type}(e\,[x \leftarrow e']) & (\text{Lemma 6}) \end{array}$$

case β-LET

$$\begin{array}{lll} & \mathsf{type}(\texttt{let}\ x_1\ \texttt{be}\ e_1\ \texttt{in}\ e_2) & \\ = & \texttt{let}\ t_{x_1}\ \texttt{be}\ \mathsf{type}(e_1)\ \texttt{in}\ \mathsf{type}(e_2) & (\textsc{Let}) \\ \geq & \mathsf{type}(e_2)\,[t_{x_1} \leftarrow \mathsf{type}(e_1)] & (\text{Definition 4.}\mathbf{iv}) \\ = & \mathsf{type}(e_2\,[x_1 \leftarrow e_1]) & (\text{Lemma 6}) \end{array}$$

case CTXT

In this case, $e = \mathcal{E}[e_1]$ with $e_1 \longrightarrow e_1'$. So, by induction hypothesis, $\mathsf{type}(e_1') \leq \mathsf{type}(e_1)$ **(1)**. We then reason by case on the form of $\mathcal{E}$.

case $\mathcal{E} = []\ e_2$

$$\begin{array}{lll} & \mathsf{type}(e_1'\ e_2) & \\ = & \mathsf{type}(e_1')\ \mathsf{type}(e_2) & (\textsc{App}) \\ \leq & \mathsf{type}(e_1)\ \mathsf{type}(e_2) & (\text{Definition 4.}\mathbf{ii}\text{ with }\mathbf{(1)}) \\ = & \mathsf{type}(e_1\ e_2) & (\textsc{App}) \end{array}$$

case $\mathcal{E} = e_2\ []$

This case is similar to $\mathcal{E} = []\ e_2$.

case $\mathcal{E} = \texttt{let}\ x_1\ \texttt{be}\ []\ \texttt{in}\ e_2$

$$\begin{array}{lll} & \mathsf{type}(\texttt{let}\ x_1\ \texttt{be}\ e_1'\ \texttt{in}\ e_2) & \\ = & \texttt{let}\ t_{x_1}\ \texttt{be}\ \mathsf{type}(e_1')\ \texttt{in}\ \mathsf{type}(e_2) & (\textsc{Let}) \\ \leq & \texttt{let}\ t_{x_1}\ \texttt{be}\ \mathsf{type}(e_1)\ \texttt{in}\ \mathsf{type}(e_2) & (\text{Definition 4.}\mathbf{ii}\text{ with }\mathbf{(1)}) \\ = & \mathsf{type}(\texttt{let}\ x_1\ \texttt{be}\ e_1\ \texttt{in}\ e_2) & (\textsc{Let}) \end{array}$$

■

1.2.1 Interpretation

In most type systems, expressions are directly assigned a type. In our system, they are instead assigned a syntactic type, which is then interpreted in a type algebra. In this section, we present an intuition about the significance of these syntactic types.

One can note that there is a strong correspondence in structure between the syntactic types τ and the expressions e of the core language:

$$\begin{array}{l} \tau ::= t \mid a \mid \lambda t.\tau \mid \tau\ \tau \mid \texttt{let}\ t\ \texttt{be}\ \tau\ \texttt{in}\ \tau \\ e ::= x \mid c \mid \lambda x.e \mid e\ e \mid \texttt{let}\ x\ \texttt{be}\ e\ \texttt{in}\ e \end{array}$$

Intuitively, the typing function returns a syntactic type that reflects the structure of the program. It only approximates constants to their defined types. This parallel is natural if one considers types as an

abstraction of expressions, where the type of an expression provides partial knowledge about this expression. Thus, the typing function that maps an expression to its (principal) type is an abstraction function.

Because of this parallel, the subject reduction theorem can be proved after imposing properties on types that follow the reduction of expressions. Thus, the property **iii** of Definition 4 (Reduction) at the level of types corresponds to the β-reduction rule for expressions:

$$\begin{array}{lcl} (\lambda t.\tau)\ \tau' & \geq & \tau\,[t \leftarrow \tau'] \\ (\lambda x.e)\ e' & \longrightarrow & e\,[x \leftarrow e'] \end{array}$$

Similarly, property **ii** of Definition 4 (Covariance) correspond to the reductions CTXT, and property **iv** of Definition 4 (Let) corresponds to the β-let reduction rule:

$$\begin{array}{lcl} \texttt{let}\ t\ \texttt{be}\ \tau\ \texttt{in}\ \tau' & \geq & \tau\,[t \leftarrow \tau'] \\ \texttt{let}\ x\ \texttt{be}\ e\ \texttt{in}\ e' & \longrightarrow & e\,[t \leftarrow e'] \end{array}$$

However, one should not think that this presentation is merely postponing all the proof requirements to the type algebra. Important parts of the proof are done in the framework, as shown by Lemma 6 (SUBSTITUTION) and Theorem 10 (SUBJECT REDUCTION). Furthermore, the remaining proofs in the type algebra are done at a higher level of abstraction, since they do not need to refer to expressions anymore.

1.2.2 Progress

We chose so far to present semantics with a notion of reduction that encompasses both call-by-value and call-by-name. Our framework can therefore be easily adapted to use these two semantics while preserving Theorem 10 (SUBJECT REDUCTION), since it is easy to see that they have fewer reductions than the reduction defined in Figure 1.3. Furthermore, it is straightforward to impose a deterministic evaluation order by restricting the set of evaluation contexts $\mathcal{E}$. In this section, we present a call-by-value semantics with left-to-right evaluation order. It is defined in Figure 1.5.

Evaluation context $\mathcal{E}_{\mathsf{CBV}}\ [] \quad ::= \quad []\ e \mid v\ [] \mid \texttt{let}\ x\ \texttt{be}\ []\ \texttt{in}\ e$

β
$(\lambda x.e)\ v \longrightarrow_{\mathsf{CBV}} e\,[x \leftarrow v]$

β-LET
$\texttt{let}\ x_1\ \texttt{be}\ v_1\ \texttt{in}\ e_2 \longrightarrow_{\mathsf{CBV}} e_2\,[x_1 \leftarrow v_1]$

OP
$$\frac{}{f\ v_1\ ...\ v_n \longrightarrow_{\mathsf{CBV}} e}\ (v_1, ..., v_n, e) \in R(f)$$

CTXT
$$\frac{e \longrightarrow_{\mathsf{CBV}} e'}{\mathcal{E}_{\mathsf{CBV}}[e] \longrightarrow \mathcal{E}_{\mathsf{CBV}}[e']}$$

Figure 1.5: Call-by-value semantics

Theorem 11 (Progress) *If an expression e is closed and well-typed then either e is a value, or there exists an expression e' such that $e \longrightarrow_{\mathsf{CBV}} e'$.*

Proof

The proof is by induction on e.

case $e = x$
This case is impossible because x is not closed.

case $e = c$
Then by definition e is a value.

case $e = \lambda x.e'$
Then e is a value.

case $e = \texttt{let}\ x_1\ \texttt{be}\ e_1\ \texttt{in}\ e_2$
By property **i** of Definition 4 (Error), the expression e_1 must be well-typed, otherwise e would not be well-typed. Therefore, by induction hypothesis, either e_1 is a value, in which case e reduces by CTXT, or e_1 reduces, in which case e reduces by β-LET.

case $e = e_1\ e_2$
By APP, e_1 and e_2 are well-typed. By induction hypothesis on e_1, there are two cases. If $e_1 \longrightarrow_{\mathsf{CBV}} e_1'$, then by CTXT, $e \longrightarrow_{\mathsf{CBV}} e_1'\ e_2$, which proves the desired property. Otherwise, e_1 is a value v_1. In this case we apply the induction hypothesis to e_2. If $e_2 \longrightarrow_{\mathsf{CBV}} e_2'$, then by CTXT $e \longrightarrow_{\mathsf{CBV}} v_1\ e_2'$, which proves the property. Otherwise, e_2 is a value v_2. We now proceed by case on v_1. By Figure 1.2, there are three cases:

case $v_1 = \lambda x_1.e_1'$
Then e reduces by β.

case $v_1 = C\ v_1'\ ...\ v_n'$ **with** $n \leq \mathsf{arity}(C)$
By hypothesis, $e = C\ v_1'\ ...\ v_n'\ v_2$ is well-typed. So by Requirement 9 (CONSTANTS), $n+1 \leq \mathsf{arity}(C)$. Hence e is also a value.

case $v_1 = f\ v_1'\ ...\ v_n'$ **with** $n < \mathsf{arity}(f)$
If $n+1 = \mathsf{arity}(f)$, then since $e = f\ v_1'\ ...\ v_{n-1}'\ v_2$, Requirement 9 (CONSTANTS) imposes that there exists an e' such that $e \longrightarrow_{\mathsf{CBV}} e'$. Otherwise, $n+1 < \mathsf{arity}(f)$, so e is a value.

■ We can therefore present a type soundness result, which is a trivial consequence of Theorem 10 (SUBJECT REDUCTION) and Theorem 11 (PROGRESS):

Theorem 12 (Soundness)
Let e be a well-typed expression. Either all reductions of e are infinite, or there exists a value v such that $e \longrightarrow_{\mathsf{CBV}} ... \longrightarrow_{\mathsf{CBV}} v$ and v is well-typed. Furthermore, $\mathsf{type}(e) \geq \mathsf{type}(v)$.

Proof of theorem 12 (Soundness)
Let's assume that there exists a finite reduction of e, that is $e \longrightarrow^*_{\mathsf{CBV}} e'$ and $e' \not\longrightarrow_{\mathsf{CBV}}$. Theorem 10 (SUBJECT REDUCTION) shows that $\mathsf{type}(e) \geq \mathsf{type}(e')$. Furthermore, since $e' \not\longrightarrow_{\mathsf{CBV}}$, Theorem 11 (PROGRESS) shows that e' is a value. ■

1.3 Instantiating the algebraic type system

The algebraic type system contains two degrees of freedom, that can be used to build type systems for various programming languages.

First, a domain A for (algebraic) types must be provided together with a pre-order $\leq$ on $S(A)$ such that $(A, \leq)$ is a type algebra. Given an expression e, our framework computes the syntactic type $\mathsf{type}(e)$. Typability and type-checking are decidable if $\leq$ is decidable, by applying the following definitions, respectively: the expression e is well typed if $\mathsf{type}(e) < \mathbb{E}$ and it has algebraic type a if $\mathsf{type}(e) \leq a$. Regarding type inference, the syntactic type $\mathsf{type}(e)$ can not be considered as a satisfactory type for an expression e, since it is roughly as large as e. On the other hand, it is often possible to provide a partial function that simplifies syntactic types into another set (for instance algebraic types). In that case, the composition of the typing function and that translation function performs type inference. We present in the next section two detailed instantiations: the Hindley-Milner type system and the $\mathrm{ML}_{\leq}$ type system. Both instantiations support type inference. This shows that our system is general enough to express complex and various type systems, while factoring out a substantial part of the soundness proofs.

The second parameter of our framework is the set of operators, which can be used to model a complete language by providing data constructors and operators that satisfy Requirement 9 (CONSTANTS). In Part II, we use this facility to model a realistic object-oriented language with classes and multi-methods.

Chapter 2

Type algebras

2.1 The Hindley-Milner type system

As an example, we show here that Hindley-Milner type schemes form a type algebra. Our framework can thus be used to present the type system of ML with a novel and concise proof of its correctness.

The syntax for Hindley-Milner types is recalled in Figure 2.1. Type variables are denoted by α as usual in ML, while types are denoted by θ (rather than τ so as to avoid confusion with syntactic types). Type schemes are considered equal modulo renaming of bound type variables and removal of quantified type variables that do not occur in the type, and reordering of quantified type variables.

Base type	ι	::=	$\texttt{int} \mid \texttt{bool} \mid \ldots$
Type	θ	::=	$\iota \mid \theta \to \theta \mid \alpha$
Type scheme	σ	::=	$\forall\overline{\alpha}.\theta$

Figure 2.1: Type syntax

2.1.1 Instantiation of the framework

To instantiate the framework, we first define the set of algebraic types HM as the set of type schemes. We then provide a pre-order on $S(HM)$ — the set of syntactic types built on HM — that verifies the requirements of Definition 4. As in the example of Section 1.1.2, we do so in two steps. First, we translate syntactic types into a suitable form in which they can be compared and second we provide the ordering on the translated form. However, it would not be sufficient to translate syntactic types to type schemes, since that would incur a loss of information. For instance, consider the syntactic type $(\texttt{int} \to \texttt{bool})\ t$. Its natural translation would be the type $\texttt{bool}$. However, we also need to keep track of the information that the syntactic type variable t is constrained to be equal to $\texttt{int}$. Therefore, we define *constrained* type schemes in Definition 13 as triples $\forall\overline{\alpha}|C.\theta$ where C is a set of equalities of the form $\theta_1 = \theta_2$. We call HMC the set of constrained type schemes. The translation of type $(\texttt{int} \to \texttt{bool})\ t$ can then be defined as $\forall|t = \texttt{int.bool}$, which retains the desired information on t.

Therefore, we will form a type algebra based on HMC. Since HM can be seen as a subset of HMC by using empty constraints, this will induce by Theorem 5 (Sub-algebra) a type algebra on HM.

Definition 13 (Constrained type schemes) *A constrained type scheme is a triple written $\forall\overline{\alpha}|C.\theta$ where $\forall\overline{\alpha}.\theta$ is a type scheme, and C is a set of equalities of the form $\theta_1 = \theta_2$.*

Definition 14 (Translation of Hindley-Milner types) *Given a syntactic types τ in $S(HMC)$, its translation $\langle\tau\rangle$ into a constrained type scheme is defined by induction in Figure 2.2.*

$$\langle \forall\overline{\alpha}|C.\theta\rangle = \forall\overline{\alpha}|C.\theta \qquad \langle t\rangle = \forall\emptyset|\emptyset.\alpha_t \qquad \langle \mathbb{E}\rangle = \forall\alpha|\texttt{int} = \texttt{bool}.\alpha \qquad \frac{\langle\tau\rangle = \forall\overline{\alpha}|C.\theta}{\langle\lambda t.\tau\rangle = \forall\overline{\alpha}\alpha_t|C.\,\alpha_t \to \theta}$$

$$\frac{\langle\tau_1\rangle = \forall\overline{\alpha}_1|C_1.\theta_1 \qquad \langle\tau_2\rangle = \forall\overline{\alpha}_2|C_2.\theta_2}{\langle\tau_1\ \tau_2\rangle = \forall\overline{\alpha}_1\overline{\alpha}_2\alpha|C_1 \cup C_2 \cup \{\theta_1 = \theta_2 \to \alpha\}.\alpha}\ \alpha \notin \overline{\alpha}_1, \overline{\alpha}_2, C_1, C_2, \theta_1, \theta_2$$

$$\frac{\langle\tau_1\rangle = \forall\overline{\alpha}_1|C_1.\theta_1 \qquad \langle\tau_2\rangle = \forall\overline{\alpha}_2|C_2.\theta_2}{\langle\texttt{let}\ t\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2\rangle = \forall\overline{\alpha}_1\overline{\alpha}_2|C_1 \cup C_2.\theta_2}\ t \notin FV(\tau_2)$$

$$\frac{}{\langle\texttt{let}\ t\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2\rangle = \langle\tau_2\,[t \leftarrow \langle\tau_1\rangle]\rangle}\ t \in FV(\tau_2)$$

Figure 2.2: Translation for Hindley-Milner

The choice of the names of bound type variables does not matter, since constrained type schemes, like syntactic types, are equal up to α-conversion. The translation of the error type is a constrained type scheme with an unsatisfiable constraint, which we arbitrarily chose to be $\texttt{int} = \texttt{bool}$. The translation of $\lambda t.\tau$ is basically $\langle t\rangle \to \langle\tau\rangle$, except that generalization on t is done immediately.

It should not be surprising that unification occurs during type application. In particular this fact makes explicit that in the usual ML rule

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}$$

the two occurrences of τ in the premises amount to an equality constraint between two *a priori* different types.

The translation function distinguishes let types depending on whether the bound type variable occurs in the body. In $\texttt{let}\ t\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2$, if t does not appear free in τ_2, we need to make sure that the constraints generated by the translation of τ_1 is copied in the resulting translation. Otherwise, it could miss the fact that τ_1 and τ_2 impose incompatible constraints on a free type variable. This is for instance the case in $\texttt{let}\ t\ \texttt{be}\ (\texttt{int} \to \texttt{int})\ t_0\ \texttt{in}\ (\texttt{boolean} \to \texttt{boolean})\ t_0$ (which correspond to the expression $\texttt{let}\ x\ \texttt{be}\ 1 + x_0\ \texttt{in not}\ x_0$).

We are now able to define the order on $S(HMC)$ in the following way. We shall denote ground substitutions by ρ, and write $\rho(C)$ if for each $\theta_1 = \theta_2$ in C, $\rho(\theta_1)$ is syntactically equal to $\rho(\theta_2)$. For two ground substitutions ρ_1 and ρ_2, and a set of type variables $\overline{\alpha}$, we will say that ρ_1 and ρ_2 are equal on $\overline{\alpha}$, which we write $\rho_1 \overset{\overline{\alpha}}{=} \rho_2$, if for all type variable α in $\overline{\alpha}$, $\rho_1(\alpha) = \rho_2(\alpha)$.

Definition 15 (Ordering of constrained type schemes)
Let $\forall\overline{\alpha}_1|C_1.\theta_1$ and $\forall\overline{\alpha}_2|C_2.\theta_2$ be two constrained type schemes. Let $\overline{\alpha}$ be $FV(\forall\overline{\alpha}_1|C_1.\theta_1) \cup FV(\forall\overline{\alpha}_2|C_2.\theta_2)$. By α-conversion, we assume that $\overline{\alpha}_1$ and $\overline{\alpha}_2$ are disjoint from $\overline{\alpha}$.

The relation $\forall\overline{\alpha}_1|C_1.\theta_1 \leq \forall\overline{\alpha}_2|C_2.\theta_2$ holds if and only if for all ground substitution ρ_2 with domain $\overline{\alpha} \cup \overline{\alpha}_2$ such that $\rho_2(C_2)$ holds, there exists a substitution ρ_1 with domain $\overline{\alpha} \cup \overline{\alpha}_1$ such that $\rho_1 \overset{\overline{\alpha}}{=} \rho_2$, $\rho_1(C_1)$ holds, and $\rho_1(\theta_1) = \rho_2(\theta_2)$.

We will say that two constrained type schemes are equivalent if each one is smaller than the other.

Definition 16 (Order on $S(HMC)$) *Let τ_1 and τ_2 be two syntactic types of $S(HMC)$. The relation $\tau_1 \leq_{HMC} \tau_2$ holds if and only if $\langle\tau_1\rangle \leq \langle\tau_2\rangle$ holds.*

In this section, we shall use $\leq$ to denote $\leq_{HMC}$ when that does not lead to ambiguities.

It is worth noting that the relation $\tau_1 \leq_{HMC} \tau_2$ always holds if the constraint in the translation of τ_2 is not satisfiable. Since the error type is maximal, this means that syntactic types that generate unsatisfiable

constraints are equivalent to the error type. Therefore, values of these types are not well-typed. For instance, the translation of the syntactic type $(\texttt{int} \to \texttt{int})\ \texttt{bool}$ is $\forall\emptyset|\texttt{bool} \leq \texttt{int}.\texttt{int}$. Since the constraint $\texttt{bool} \leq \texttt{int}$ is not satisfiable, we have $(\texttt{int} \to \texttt{int})\ \texttt{bool} \equiv \mathbb{E}$.

We can now assert that we have formed a type algebra in the sense of Section 1.1.

Theorem 17 (HMC) *The couple $(HMC, \leq_{HMC})$ is a type algebra.*

By Theorem 5 (Sub-algebra), this implies that Hindley-Milner type schemes also form a type algebra.

Corollary 18 (HM) *The couple $(HM, \leq_{HM})$ is a type algebra, where, for any type $\forall\overline{\alpha}_1.\theta_1$ and $\forall\overline{\alpha}_2.\theta_2$ in $S(HM)$, $\forall\overline{\alpha}_1.\theta_1 \leq \forall\overline{\alpha}_2.\theta_2$ iff $\forall\overline{\alpha}_1|\theta_1.\emptyset \leq_{HMC} \forall\overline{\alpha}_2|\theta_2.\emptyset$.*

The proof is given below. It only needs to deal with technical properties of the pre-order we introduced. This fact shows that we indeed achieved to factorize substantial parts of the soundness proof.

We define a size function on syntactic types, designed so that the size of a type is smaller than the size of its components and so that $\text{size}(\texttt{let}\ t_1\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2) > \text{size}(\tau_2\,[t_1 \leftarrow \tau_1])$. Note that for all type τ, $\text{size}(\tau) \geq 1$.

Definition 19 (Size of a syntactic type) *Given a syntactic type τ, and a function f from type variables to integer numbers, we define $\text{size}_f(\tau)$ by structural induction on τ, with:*

$$\begin{aligned}
\text{size}_f(a) &= 1\\
\text{size}_f(t) &= 1\\
\text{size}_f(\lambda t.\tau) &= 1 + \text{size}_{f+t\mapsto 1}(\tau)\\
\text{size}_f(\tau_1\ \tau_2) &= \text{size}_f(\tau_1) + \text{size}_f(\tau_2)\\
\text{size}_f(\texttt{let}\ t_1\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2) &= \text{size}_f(\tau_1) + \text{size}_{f+t_1\mapsto \text{size}_f(\tau_1)}(\tau_2)\\
\text{size}_f(\mathbb{E}) &= 1
\end{aligned}$$

Given a syntactic type τ, we define $\text{size}(\tau)$ as $\text{size}_{\overline{1}}(\tau)$, where $\overline{1}$ is the constant function mapping all type variables to 1.

Proof of theorem 17 (HMC)

Let us first prove property **i** of Definition 4 (Error). Since the constraint of $\langle\mathbb{E}\rangle$ is unsatisfiable, $\tau \leq \mathbb{E}$ always holds by Definition 15. Furthermore, the constraint in the translation of $\tau\ \mathbb{E}$ and $\mathbb{E}\ \tau$ includes the constraint of $\langle\mathbb{E}\rangle$, so it is also unsatisfiable, and these two types are indeed greater than $\mathbb{E}$. It is also the case of $\texttt{let}\ t\ \texttt{be}\ \mathbb{E}\ \texttt{in}\ \tau$: when t does not appear in τ, the unsatisfiable constraint in $\langle\mathbb{E}\rangle$ is directly copied into the constraint set of $\langle\texttt{let}\ t\ \texttt{be}\ \mathbb{E}\ \texttt{in}\ \tau\rangle$; when t appears in τ, a straightforward induction show that the unsatisfiable constraint appears in $\langle\tau\,[t \leftarrow \mathbb{E}]\rangle$.

- We now prove property **ii** of Definition 4 (Covariance). We therefore assume $\tau_1 \leq \tau_2$. We prove that $\tau\,[t \leftarrow \tau_1] \leq \tau\,[t \leftarrow \tau_2]$ by induction on τ.

  case $\tau = t$
  We need to prove $\tau_1 \leq \tau_2$, which is the hypothesis.

  case $\tau = \mathbb{E}$ **or** $\tau = a$ **or** $\tau = t'$ **with** $t' \neq t$
  The result is immediate since by Definition 3, both $\tau\,[t \leftarrow \tau_1]$ and $\tau\,[t \leftarrow \tau_2]$ are equal to τ.

  case $\tau = \tau_0\ \tau_0'$
  By Definition 3, for $i = 1, 2$, $\tau\,[t \leftarrow \tau_i] = (\tau_0\,[t \leftarrow \tau_i])\ (\tau_0'\,[t \leftarrow \tau_i])$. For $i = 1, 2$, let $\forall\overline{\alpha}_i|C_i.\theta_i$ be $\langle\tau_0\,[t \leftarrow \tau_i]\rangle$, $\overline{\beta}_i$ be $FV(\forall\overline{\alpha}_i|C_i.\theta_i)$ and $\overline{\beta}$ be $\overline{\beta}_1 \cup \overline{\beta}_2$. Let also $\forall\overline{\alpha}_i'|C_i'.\theta_i'$ be $\langle\tau_0'\,[t \leftarrow \tau_i]\rangle$, $\overline{\beta}_i'$ be $FV(\forall\overline{\alpha}_i'|C_i'.\theta_i')$ and, $\overline{\beta}'$ be $\overline{\beta}_1' \cup \overline{\beta}_2'$. We can assume w.l.o.g. that the $\overline{\alpha}_i$ and $\overline{\alpha}_i'$ are all pair-wise disjoint (**3**) and disjoint with $\overline{\beta} \cup \overline{\beta}'$. Then, by Definition 14, $\langle(\tau_0\,[t \leftarrow \tau_i])\ (\tau_0'\,[t \leftarrow \tau_i])\rangle = \forall\overline{\alpha}_i\overline{\alpha}_i'\alpha|C_i \cup C_i' \cup \{\theta_i = \theta_i' \to \alpha\}.\alpha$, where α is fresh, that is $\alpha \notin \overline{\alpha}_i\overline{\alpha}_i'$ (**4**). Note that

we are able to choose to share the same α in both translations by α-conversion on constrained type schemes. Let ρ_2 be a substitution such that $\rho_2(C_2 \cup C_2' \cup \{\theta_2 = \theta_2' \to \alpha\})$ holds (**5**). Since $\rho_2(C_2)$ holds by construction, by the induction hypothesis $\tau_0\,[t \leftarrow \tau_1] \leq \tau_0\,[t \leftarrow \tau_2]$ (that is $\forall\overline{\alpha}_1|C_1.\theta_1 \leq \forall\overline{\alpha}_2|C_2.\theta_2$) and Definition 15, there exists ρ_1 such that $\rho_1 \overset{\overline{\beta}}{=} \rho_2$ (**6**), $\rho_1(C_1)$ holds (**7**) and $\rho_1(\theta_1) = \rho_2(\theta_2)$ (**8**). Similarly, $\rho_2(C_2')$ holds, so there exists ρ_1' such that $\rho_1' \overset{\overline{\beta}'}{=} \rho_2$ (**9**), $\rho_1'(C_1')$ holds (**10**) and $\rho_1'(\theta_1') = \rho_2(\theta_2')$ (**11**). Since $\overline{\alpha}_1$ and $\overline{\alpha}_1'$ are disjoint by (3), we define the substitution ρ with:

$$\rho : \begin{cases} \gamma \in \overline{\alpha}_1 & \mapsto \rho_1(\gamma) \quad (\mathbf{12}) \\ \gamma \in \overline{\alpha}_1' & \mapsto \rho_1'(\gamma) \quad (\mathbf{13}) \\ \gamma \in \overline{\beta} \cup \overline{\beta}' & \mapsto \rho_2(\gamma) \quad (\mathbf{14}) \\ \alpha & \mapsto \rho_2(\alpha) \quad (\mathbf{15}) \end{cases}$$

This substitution verifies $\rho \overset{\overline{\beta}\overline{\beta}'}{=} \rho_2$ by (14). In particular, $\rho(\alpha) = \rho_2(\alpha)$. By (14), (6) and (9), ρ agrees with all of ρ_1, ρ_1' and ρ_2 on variables in $\overline{\beta} \cup \overline{\beta}'$ (**16**). Furthermore, $\rho(C_1)$ holds since $\rho_1(C_1)$ holds by (7) and $\rho \overset{\alpha_1\beta_1}{=} \rho_1$ by (12) and (16). Similarly, $\rho(C_2)$ holds by (10), (13) and (16). Finally, $\rho(\theta_1) = \rho_1(\theta_1)$ by (12) and (16), and $\rho_1(\theta_1) = \rho_2(\theta_2)$ by (8). By (5), $\rho_2(\theta_2) = \rho_2(\theta_2' \to \alpha)$, which is by definition $\rho_2(\theta_2') \to \rho_2(\alpha)$. By (11) $\rho_2(\theta_2') = \rho_1'(\theta_1')$ and by (13) and (16) $\rho_1'(\theta_1') = \rho(\theta_1')$. By (15) $\rho_2(\alpha) = \rho(\alpha)$. Therefore, $\rho_2(\theta_2') \to \rho_2(\alpha)$ is equal to $\rho(\theta_1') \to \rho(\alpha)$. This shows that ρ satisfies $\theta_1 = \theta_1' \to \alpha$. Therefore, by Definition 15, $(\tau_0\ \tau_0')\,[t \leftarrow \tau_1] \leq (\tau_0\ \tau_0')\,[t \leftarrow \tau_2]$.

case $\tau = \lambda t_0.\tau_0$

We can assume w.l.o.g. that t_0 is different from t, and not free in τ_1 nor τ_2. For $i = 1, 2$, let $\forall\overline{\alpha}_i|C_i.\theta_i$ be $\langle\tau_0\,[t \leftarrow \tau_i]\rangle$, $\overline{\beta}_i$ be $FV(\forall\overline{\alpha}_i|C_i.\theta_i)$ and $\overline{\beta}$ be $\overline{\beta}_1 \cup \overline{\beta}_2$. Then by Definition 14 $\langle\lambda t_0.\tau_0\,[t \leftarrow \tau_i]\rangle = \forall\overline{\alpha}_i\alpha_{t_0}|C_i.\,\alpha_{t_0} \to \theta_i$. Let ρ_2 be such that $\rho_2(C_2)$ holds. By the induction hypothesis $\tau_0\,[t \leftarrow \tau_1] \leq \tau_0\,[t \leftarrow \tau_2]$, there exists ρ_1 such that $\rho_1 \overset{\overline{\beta}}{=} \rho_2$ (**1**), $\rho_1(C_1)$ holds (**2**) and $\rho_1(\theta_1) = \rho_2(\theta_2)$ (**3**). First, (1) implies in particular that ρ_1 is equal to ρ_2 on $\overline{\beta} \setminus \{\alpha_{t_0}\}$. Second, $\rho_1(C_1)$ holds by (2). Third,

$$\begin{array}{lll} & \rho_1(\,\alpha_{t_0} \to \theta_1) & \\ = & \rho_1(\alpha_{t_0}) \to \rho_1(\theta_1) & \\ = & \rho_2(\alpha_{t_0}) \to \rho_1(\theta_1) & (1) \\ = & \rho_2(\alpha_{t_0}) \to \rho_2(\theta_2) & (3) \\ = & \rho_2(\,\alpha_{t_0} \to \theta_2) & \end{array}$$

Therefore $\lambda t_0.\tau_0\,[t \leftarrow \tau_1] \leq \lambda t_0.\tau_0\,[t \leftarrow \tau_2]$.

case $\tau = \texttt{let}\ t_0\ \texttt{be}\ \tau_0\ \texttt{in}\ \tau_0'$ **with** $t_0 \in FV(\tau_0')$

We can assume w.l.o.g. that t_0 is different from t (**1**) and not free in τ_1 (**2**) nor in τ_2 (**3**). Therefore,

$$\begin{array}{lll} & (\texttt{let}\ t_0\ \texttt{be}\ \tau_0\ \texttt{in}\ \tau_0')\,[t \leftarrow \tau_1] & \\ = & \texttt{let}\ t_0\ \texttt{be}\ \tau_0\,[t \leftarrow \tau_1]\ \texttt{in}\ \tau_0'\,[t \leftarrow \tau_1] & \text{(Definition 3)} \\ \equiv & \tau_0'\,[t \leftarrow \tau_1]\,[t_0 \leftarrow \langle\tau_0\,[t \leftarrow \tau_1]\rangle] & \text{(Definition 14 and 16)} \\ = & \tau_0'\,[t_0 \leftarrow \langle\tau_0\,[t \leftarrow \tau_1]\rangle]\,[t \leftarrow \tau_1] & \text{((1) and (2) and } t \notin \langle\tau_0\,[t \leftarrow \tau_1]\rangle) \end{array}$$

By induction hypothesis on τ_0, $\tau_0\,[t \leftarrow \tau_1]$ is smaller than $\tau_0\,[t \leftarrow \tau_2]$. That is, by Definition 16, $\langle\tau_0\,[t \leftarrow \tau_1]\rangle$ is smaller than $\langle\tau_0\,[t \leftarrow \tau_2]\rangle$.

Therefore, by induction hypothesis on τ_0', $\tau_0'\,[t_0 \leftarrow \langle\tau_0\,[t \leftarrow \tau_1]\rangle]$ is smaller than $\tau_0'\,[t_0 \leftarrow \langle\tau_0\,[t \leftarrow \tau_2]\rangle]$.

The type $\tau_0'\,[t_0 \leftarrow \langle\tau_0\,[t \leftarrow \tau_2]\rangle]$ has a smaller size than τ since $\langle\tau_0\,[t \leftarrow \tau_2]\rangle$, being an algebraic type, has size 1 by Definition 19. Therefore, we can apply the induction hypothesis, which implies that $\tau_0'\,[t_0 \leftarrow \langle\tau_0\,[t \leftarrow \tau_2]\rangle]\,[t \leftarrow \tau_1]$ is smaller than $\tau_0'\,[t_0 \leftarrow \langle\tau_0\,[t \leftarrow \tau_2]\rangle]\,[t \leftarrow \tau_2]$. Furthermore,

$$
\begin{array}{lll}
 & \tau_0'\,[t_0 \leftarrow \langle \tau_0\,[t \leftarrow \tau_2]\rangle]\,[t \leftarrow \tau_2] & \\
= & \tau_0'\,[t \leftarrow \tau_2]\,[t_0 \leftarrow \langle \tau_0\,[t \leftarrow \tau_2]\rangle] & ((1) \text{ and } (3) \text{ and } t \notin \langle \tau_0\,[t \leftarrow \tau_2]\rangle) \\
\equiv & \texttt{let } t_0 \texttt{ be } \tau_0\,[t \leftarrow \tau_2] \texttt{ in } \tau_0'\,[t \leftarrow \tau_2] & \text{(Definition 14)} \\
= & (\texttt{let } t_0 \texttt{ be } \tau_0 \texttt{ in } \tau_0')\,[t \leftarrow \tau_2] & \text{(Definition 3)}
\end{array}
$$

case $\tau = \texttt{let } t_0 \texttt{ be } \tau_0 \texttt{ in } \tau_0'$ **with** $t_0 \notin FV(\tau_0')$

We can assume w.l.o.g. that t_0 is different from t and not free in τ_1 nor in τ_2. By Definition 3, $\tau\,[t \leftarrow \tau_1] = \texttt{let } t_0 \texttt{ be } \tau_0\,[t \leftarrow \tau_1] \texttt{ in } \tau_0'\,[t \leftarrow \tau_1]$.

For $i = 1, 2$, let $\forall \overline{\alpha}_i | C_i.\theta_i$ be $\langle \tau_0\,[t \leftarrow \tau_i]\rangle$, $\overline{\beta}_i$ be $FV(\forall \overline{\alpha}_i | C_i.\theta_i)$ and $\overline{\beta}$ be $\overline{\beta}_1 \cup \overline{\beta}_2$. Let also $\forall \overline{\alpha}_i' | C_i'.\theta_i'$ be $\langle \tau_0'\,[t \leftarrow \tau_i]\rangle$, $\overline{\beta}_i'$ be $FV(\forall \overline{\alpha}_i' | C_i'.\theta_i')$ and, $\overline{\beta}'$ be $\overline{\beta}_1' \cup \overline{\beta}_2'$. We can assume w.l.o.g. that the $\overline{\alpha}_i$ and $\overline{\alpha}_i'$ are all pair-wise disjoint (**1**) and disjoint with $\overline{\beta} \cup \overline{\beta}'$.

Then, by Definition 14, $\langle \texttt{let } t_0 \texttt{ be } \tau_0\,[t \leftarrow \tau_i] \texttt{ in } \tau_0'\,[t \leftarrow \tau_i]\rangle = \forall \overline{\alpha}_i \overline{\alpha}_i' | C_i \cup C_i'.\theta_i'$. For any substitution ρ_2 such that $\rho_2(C_2 \cup C_2')$ holds, by construction, $\rho_2(C_2)$ holds. By induction hypothesis, $\tau_0\,[t \leftarrow \tau_1]$ is smaller than $\tau_0\,[t \leftarrow \tau_2]$ That is, by Definition 16, $\langle \tau_0\,[t \leftarrow \tau_1]\rangle$ is smaller than $\langle \tau_0\,[t \leftarrow \tau_2]\rangle$. Therefore, by Definition 15, there exists ρ_1 such that $\rho_1 \overset{\overline{\beta}}{=} \rho_2$ (**2**), $\rho_1(C_1)$ holds (**3**) and $\rho_1(\theta_1) = \rho_2(\theta_2)$. Similarly, $\rho_2(C_2')$ holds, so there exists ρ_1' such that $\rho_1' \overset{\overline{\beta}'}{=} \rho_2$ (**4**), $\rho_1'(C_1')$ holds (**5**) and $\rho_1'(\theta_1') = \rho_2(\theta_2')$ (**6**). By (1), we can define the substitution ρ that agrees with ρ_1 on $\overline{\alpha}_1$ (**7**), with ρ_1' on $\overline{\alpha}_1'$ (**8**), and with all of ρ_1, ρ_1' and ρ_2 on other variables (**9**), which is possible by (2) and (4). Therefore, $\rho \overset{\overline{\beta}_1 \overline{\beta}'}{=} \rho_2$. Furthermore, $\rho(C_1)$ holds by (3), (7) and (9), and $\rho(C_1')$ holds by (5), (8) and (9). Finally, $\rho(\theta_1') = \rho_1'(\theta_1')$ by (8) and (9), and $\rho_1'(\theta_1') = \rho_2(\theta_2')$ by (6). Therefore, by Definition 15, $(\texttt{let } t_0 \texttt{ be } \tau_0 \texttt{ in } \tau_0')\,[t \leftarrow \tau_1] \leq (\texttt{let } t_0 \texttt{ be } \tau_0 \texttt{ in } \tau_0')\,[t \leftarrow \tau_2]$.

- We now prove property **iii** of Definition 4 (Reduction): for all types τ, τ' and type variable t, $(\lambda t.\tau)\ \tau' \geq \tau\,[t \leftarrow \tau']$. The proof is by induction on the size of τ as defined in Definition 19.

  case $\tau = \lambda t_0.\tau_0$

  We can assume w.l.o.g. that t_0 is different from t and not free in τ'. Therefore, by Definition 3, $(\lambda t_0.\tau_0)\,[t \leftarrow \tau'] = \lambda t_0.(\tau_0\,[t \leftarrow \tau'])$. Since $\text{size}(\tau_0) < \text{size}(\lambda t_0.\tau_0)$, we can apply the induction hypothesis, which shows that $\tau_0\,[t \leftarrow \tau'] \leq (\lambda t.\tau_0)\ \tau'$ (**1**). Since property **ii** of Definition 4 (Covariance) is already proved, we can apply it to (1) and to the type $\lambda t_0.t_1$ where t_1 is a fresh type variable. This shows that $\lambda t_0.(\tau_0\,[t \leftarrow \tau']) \leq \lambda t_0.((\lambda t.\tau_0)\ \tau')$ holds. Thus, it only remains to show that $\lambda t_0.((\lambda t.\tau_0)\ \tau') \leq (\lambda t.\lambda t_0.\tau_0)\ \tau'$ holds. Let $\forall \overline{\alpha}_0 | C_0.\theta_0$ be $\langle \tau_0 \rangle$ and $\forall \overline{\alpha}' | C'.\theta'$ be $\langle \tau' \rangle$. Then by Definition 14, $\langle (\lambda t.\lambda t_0.\tau_0)\ \tau' \rangle = \forall \alpha \overline{\alpha}' \alpha_t \alpha_{t_0} \overline{\alpha}_0 | C' \cup C_0 \cup \{\, \alpha_t \to (\, \alpha_{t_0} \to \theta_0) = \theta' \to \alpha \}.\alpha$ and $\langle \lambda t_0.((\lambda t.\tau_0)\ \tau') \rangle = \forall \alpha' \overline{\alpha}' \alpha_t \alpha_{t_0} \overline{\alpha}_0 | C' \cup C_0 \cup \{\, \alpha_t \to \theta_0 = \theta' \to \alpha' \}.\, \alpha_{t_0} \to \alpha'$. We need to show that the latter is smaller than the former. For any substitution ρ_2 that satisfies $C' \cup C_0 \cup \{\, \alpha_t \to (\, \alpha_{t_0} \to \theta_0) = \theta' \to \alpha \}$ (**2**), let ρ_1 be $\rho_2 + \{\alpha' \mapsto \rho_2(\theta_0)\}$ (**3**). Then by (3), $\rho_1(\, \alpha_{t_0} \to \alpha') = \rho_2(\, \alpha_{t_0} \to \theta_0)$ and $\rho_2(\, \alpha_{t_0} \to \theta_0) = \rho_2(\alpha)$ by (2). Furthermore, ρ_1 satisfies $C' \cup C_0$ by (2) and (3). Finally, ρ_1 satisfies $\alpha_t \to \theta_0 = \theta' \to \alpha'$ since $\rho_2(\alpha_t) = \rho_2(\theta')$ by (2) and since $\rho_1(\theta_0) = \rho_2(\theta_0) = \rho_1(\alpha')$ by (3).

  case $\tau = \texttt{let } t_1 \texttt{ be } \tau_1 \texttt{ in } \tau_2$ **where** $t_1 \in FV(\tau_2)$ (**1**)

  We can assume w.l.o.g that t_1 is different from t (**2**), and does not belong to $FV(\tau')$ (**3**).

  By Definition 14, $\langle \tau \rangle = \langle \tau_2\,[t_1 \leftarrow \langle \tau_1 \rangle] \rangle$. That is, $\tau \equiv \tau_2\,[t_1 \leftarrow \langle \tau_1 \rangle]$. By Definition 14, $\langle \langle \tau_1 \rangle \rangle = \langle \tau_1 \rangle$, so by Definition 16, $\langle \tau_1 \rangle \equiv \tau_1$. So, by property **ii** of Definition 4 (Covariance) which we have already proved, $\tau_2\,[t_1 \leftarrow \langle \tau_1 \rangle] \equiv \tau_2\,[t_1 \leftarrow \tau_1]$. Therefore, again by property **ii** of Definition 4 (Covariance), $(\lambda t.\tau)\ \tau' \equiv (\lambda t.\tau_2\,[t_1 \leftarrow \tau_1])\ \tau'$. By Definition 19, $\tau_2\,[t_1 \leftarrow \tau_1]$ has a smaller size than $\texttt{let } t_1 \texttt{ be } \tau_1 \texttt{ in } \tau_2$. Therefore, we can apply the induction hypothesis, and $(\lambda t.\tau_2\,[t_1 \leftarrow \tau_1])\ \tau'$ is greater than $\tau_2\,[t_1 \leftarrow \tau_1]\,[t \leftarrow \tau']$, which by (2) and (3) is equal to $\tau_2\,[t \leftarrow \tau']\,[t_1 \leftarrow \tau_1\,[t \leftarrow \tau']]$. By property **ii** of Definition 4 (Covariance), this type is equivalent to $\tau_2\,[t \leftarrow \tau']\,[t_1 \leftarrow \langle \tau_1\,[t \leftarrow \tau'] \rangle]$. Since t_1 belongs to $FV(\tau_2)$ by (1), this type is by Definition 14 and Definition 16 equivalent to

$\texttt{let}\ t_1\ \texttt{be}\ \tau_1\,[t \leftarrow \tau']\ \texttt{in}\ \tau_2\,[t \leftarrow \tau']$. This shows the property, since by Definition 3, this type is equal to $(\texttt{let}\ t_1\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2)\,[t \leftarrow \tau']$.

case $\tau = \texttt{let}\ t_1\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2$ **where** $t_1 \notin FV(\tau_2)$

We can assume w.l.o.g that t_1 is different from t (**1**), and does not belong to $FV(\tau')$ (**2**).

By Definition 3 with (1) and (2), $\tau\,[t \leftarrow \tau'] = \texttt{let}\ t_1\ \texttt{be}\ \tau_1\,[t \leftarrow \tau']\ \texttt{in}\ \tau_2\,[t \leftarrow \tau']$. By applying the induction hypothesis to τ_1 and τ_2, and by property **ii** of Definition 4 (Covariance), $\texttt{let}\ t_1\ \texttt{be}\ \tau_1\,[t \leftarrow \tau']\ \texttt{in}\ \tau_2\,[t \leftarrow \tau']$ is smaller than $\texttt{let}\ t_1\ \texttt{be}\ (\lambda t.\tau_1)\ \tau'\ \texttt{in}\ (\lambda t.\tau_2)\ \tau'$. It remains to show that $\texttt{let}\ t_1\ \texttt{be}\ (\lambda t.\tau_1)\ \tau'\ \texttt{in}\ (\lambda t.\tau_2)\ \tau'$ is smaller than $(\lambda t.\texttt{let}\ t_1\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2)\ \tau'$. To this end, we compute their translations. Let $\forall\overline{\alpha}_1|C_1.\theta_1$ be $\langle\tau_1\rangle$, $\forall\overline{\alpha}_2|C_2.\theta_2$ be $\langle\tau_2\rangle$ and $\forall\overline{\alpha}'|C'.\theta'$ be $\langle\tau'\rangle$. Since τ' appears several times, we also define $\forall\overline{\alpha}''|C''.\theta''$ as an α-converted copy of $\langle\tau'\rangle$. That is, there exists a substitution ρ' renaming the variables $\overline{\alpha}''$ such that $\rho'(\overline{\alpha}'') = \overline{\alpha}'$, $\rho'(\theta'') = \theta'$, and $\rho'(C'') = C'$. Then by Definition 14,

$$\begin{array}{rcl}
\langle\texttt{let}\ t_1\ \texttt{be}\ (\lambda t.\tau_1)\ \tau'\ \texttt{in}\ (\lambda t.\tau_2)\ \tau'\rangle & = & \forall\overline{\alpha}_1\overline{\alpha}_2\alpha_t\alpha_{t'}\overline{\alpha}'\overline{\alpha}''\alpha_0\alpha_0'|C_1 \cup C_2 \cup C' \cup C''\cup \\
 & & \{\,\alpha_t \to \theta_1 = \theta' \to \alpha_0,\, \alpha_{t'} \to \theta_2 = \theta'' \to \alpha_0'\}.\alpha_0' \\
\langle(\lambda t.\texttt{let}\ t_1\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2)\ \tau'\rangle & = & \forall\overline{\alpha}_1\overline{\alpha}_2\alpha_t\overline{\alpha}'\alpha_0|C_1 \cup C_2 \cup C' \cup \{\,\alpha_t \to \theta_2 = \theta' \to \alpha_0\}.\alpha_0
\end{array}$$

For every substitution ρ_2 that satisfies $C_1 \cup C_2 \cup C' \cup \{\,\alpha_t \to \theta_2 = \theta' \to \alpha_0\}$ (**3**), let ρ_1 be $(\rho_2 \circ \rho') + \{\alpha_0 \mapsto \rho_2(\theta_1), \alpha_0' \mapsto \rho_2(\theta_2), \alpha_{t'} \mapsto (\rho 2 \circ \rho')(\theta')\}$. Then by definition of ρ_1 and (3), $\rho_1(C_1 \cup C_2 \cup C' \cup C'' \cup \{\,\alpha_t \to \theta_1 = \theta' \to \alpha_0,\, \alpha_{t'} \to \theta_2 = \theta'' \to \alpha_0'\})$ holds, and $\rho_1(\alpha_0')$ is equal to $\rho_2(\theta_2)$, which is equal to $\rho_2(\alpha_0)$ since ρ_2 satisfies $\alpha_t \to \theta_2 = \theta' \to \alpha_0$. So by Definition 16, $\texttt{let}\ t_1\ \texttt{be}\ (\lambda t.\tau_1)\ \tau'\ \texttt{in}\ (\lambda t.\tau_2)\ \tau'$ is smaller than $(\lambda t.\texttt{let}\ t_1\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2)\ \tau'$.

case $\tau = \tau_1\ \tau_2$

Similarly, it follows from induction hypothesis applied to τ_1 and τ_2 and from Definition 4.**ii** that $((\lambda t.\tau_1)\ \tau')\ ((\lambda t.\tau_2)\ \tau') \geq (\tau_1\ \tau_2)\,[t \leftarrow \tau']$. It remains to show that $((\lambda t.\tau_1)\ \tau')\ ((\lambda t.\tau_2)\ \tau')$ is smaller than $(\lambda t.(\tau_1\ \tau_2))\ \tau'$. Let $\forall\overline{\alpha}_1|C_1.\theta_1$ be $\langle\tau_1\rangle$, $\forall\overline{\alpha}_2|C_2.\theta_2$ be $\langle\tau_2\rangle$ and $\forall\overline{\alpha}'|C'.\theta'$ be $\langle\tau'\rangle$. Since τ' appears several times, we also define $\forall\overline{\alpha}''|C''.\theta''$ as an α-converted copy of $\langle\tau'\rangle$. That is, there exists a substitution ρ' renaming the variables $\overline{\alpha}''$ such that $\rho'(\overline{\alpha}'') = \overline{\alpha}'$, $\rho'(\theta'') = \theta'$, and $\rho'(C'') = C'$. Then by Definition 14,

$$\begin{array}{rcl}
\langle((\lambda t.\tau_1)\ \tau')\ ((\lambda t.\tau_2)\ \tau')\rangle & = & \forall\overline{\alpha}_1\overline{\alpha}_2\alpha_t\alpha_{t'}\overline{\alpha}'\overline{\alpha}''\beta'\beta''\beta|C_1 \cup C' \cup C_2 \cup C''\cup \\
 & & \{\,\alpha_t \to \theta_1 = \theta' \to \beta_1,\, \alpha_{t'} \to \theta_2 = \theta'' \to \beta_2, \beta_1 = \beta_2 \to \beta\}.\beta \\
\langle(\lambda t.(\tau_1\ \tau_2))\ \tau'\rangle & = & \forall\overline{\alpha}_1\overline{\alpha}_2\alpha_t\overline{\alpha}'\alpha_0\beta|C_1 \cup C_2 \cup C'\cup \\
 & & \{\,\alpha_t \to \beta = \theta' \to \alpha_0, \theta_1 = \theta_2 \to \beta\}.\alpha_0
\end{array}$$

For every substitution ρ_2 that satisfies $C_1 \cup C_2 \cup C' \cup \{\,\alpha_t \to \beta = \theta' \to \alpha_0, \theta_1 = \theta_2 \to \beta\}$ (**1**), let ρ_1 be $(\rho_2 \circ \rho') + \{\beta_1 \mapsto \rho_2(\theta_1), \beta_2 \mapsto \rho_2(\theta_2), \alpha_{t'} \mapsto \rho_2(\theta')\}$. Then by definition of ρ_1 and (1), $\rho_1(C_1 \cup C' \cup C_2 \cup C'' \cup \{\,\alpha_t \to \theta_1 = \theta' \to \beta_1,\, \alpha_{t'} \to \theta_2 = \theta'' \to \beta_2, \beta_1 = \beta_2 \to \beta\})$ holds and $\rho_1(\beta)$ is equal to $\rho_2(\beta)$, which is equal to $\rho_2(\alpha_0)$ since by (1) ρ_2 satisfies $\alpha_t \to \beta = \theta' \to \alpha_0$. So by Definition 16, $((\lambda t.\tau_1)\ \tau')\ ((\lambda t.\tau_2)\ \tau')$ is smaller than $(\lambda t.(\tau_1\ \tau_2))\ \tau'$.

case $\tau = t$

Let $\forall\overline{\alpha}'|C'.\theta'$ be $\langle\tau'\rangle$. Since $\tau = t$, $\tau\,[t \leftarrow \tau']$ is equal to τ' by Definition 3. Therefore,

$$\begin{array}{rcl}
 & & \langle t\,[t \leftarrow \tau']\rangle \\
 & = & \langle\tau'\rangle \\
 & = & \forall\overline{\alpha}'|C'.\theta' \\
 & = & \forall\overline{\alpha}'\alpha_t|C' \cup \{\alpha_t = \theta'\}.\alpha_t \\
 & = & \langle(\lambda t.t)\ \tau'\rangle
\end{array}$$

case $\tau = \mathbb{E}$ **or** $\tau = \sigma$ **or** $\tau = t' \neq t$

Let $\forall\overline{\alpha}|C.\theta$ be $\langle\tau\rangle$ and $\forall\overline{\alpha}'|C'.\theta'$ be $\langle\tau'\rangle$. Since $FV(\tau)$ is empty, $\tau\,[t \leftarrow \tau']$ is by Definition 3 equal

to τ. Therefore, $\langle \tau\,[t \leftarrow \tau'] \rangle = \langle \tau \rangle = \forall \overline{\alpha} | C.\theta$. Furthermore, $\langle (\lambda t.\tau)\ \tau' \rangle = \forall \overline{\alpha\alpha'} \alpha_t | C \cup C' \cup \{\alpha_t = \theta'\}.\theta$.

The inequality $\forall \overline{\alpha} | C.\theta \leq \forall \overline{\alpha\alpha'} \alpha_t | C \cup C' \cup \{\alpha_t = \theta'\}.\theta$ is trivial: for all ρ_2 such that $\rho_2(C \cup C' \cup \{\alpha_t = \theta'\})$, take $\rho_1 = \rho_2$ since then $\rho_1(C)$ and $\rho_1(\theta) = \rho_2(\theta)$. Note that the reverse inequality does not hold as soon as C' is not trivial.

- Finally, for property **iv** of Definition 4 (Let), we need to show that for all types τ, τ' and type variable t, $\tau'\,[t \leftarrow \tau] \leq \texttt{let}\ t\ \texttt{be}\ \tau\ \texttt{in}\ \tau'$.

  If t belongs to $FV(\tau_2)$, then by Definition 14, $\texttt{let}\ t\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2 \equiv \tau_2\,[t \leftarrow \langle \tau_1 \rangle]$. Furthermore, $\langle \tau_1 \rangle \equiv \tau_1$ by Definition 16 since $trad[\langle \tau \rangle] = \langle \tau \rangle$. Therefore, by property **ii** of Definition 4 (Covariance), $\tau_2\,[t \leftarrow \langle \tau_1 \rangle]$ is equal to $\tau_2\,[t \leftarrow \tau_1]$, which proves the property. If t does not belong to $FV(\tau_2)$, let $\forall \overline{\alpha}_1 | C_1.\theta_1$ be $\langle \tau_1 \rangle$ and $\forall \overline{\alpha}_2 | C_2.\theta_2$ be $\langle \tau_2 \rangle$. Then $\langle \tau_2\,[t \leftarrow \tau_1] \rangle = \langle \tau_2 \rangle = \forall \overline{\alpha}_2 | C_2.\theta_2$. Furthermore, $\langle \texttt{let}\ t\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2 \rangle = \forall \overline{\alpha}_1 \overline{\alpha}_2 | C_1 \cup C_2.\theta_2$. For any substitution ρ_2 such that $\rho_2(C_1 \cup C_2)$ holds, take $\rho_1 = \rho_2$. Then $\rho_1(C_1)$ holds, and $\rho_1(\theta_2) = \rho_2(\theta_2)$, so $\texttt{let}\ t\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2 \geq \tau_2\,[t \leftarrow t_1]$ holds.

■

2.1.2 Simplification

By definition, $\tau_1 \equiv \tau_2$ if $\tau_1 \leq \tau_2$ and $\tau_2 \leq \tau_1$. This provides an opportunity for type simplification. Let us write $\mathsf{mgu}(C)$ a most general unifier of a constraint C. That is, $\mathsf{mgu}(C)$ is a substitution that satisfies C, and such that for all substitution ρ that satisfies C, there exists a substitution ρ' such that $\rho = \rho' \circ \mathsf{mgu}(C)$.

Lemma 20 (Simplification) *Let $\forall \overline{\alpha} | C.\theta$ be a constrained type scheme. Let θ' be $\mathsf{mgu}(C)(\theta)$, $\overline{\alpha}' = \overline{\alpha} \cap FV(\theta')$ and C' the set of constraints $\{\mathsf{mgu}(C)(\theta_1) = \mathsf{mgu}(C)(\theta_2)\}$ for all $\{\theta_1 = \theta_2\}$ in C such that $\mathsf{mgu}(C)(\theta_1)$ is different from $\mathsf{mgu}(C)(\theta_2)$. Then $\forall \overline{\alpha} | C.\theta$ is equivalent to $\forall \overline{\alpha}' | C'.\theta'$*

Proof

First we prove that $\forall \overline{\alpha} | C.\theta$ is smaller than $\forall \overline{\alpha}' | C'.\theta'$. For all ρ_2 that satisfies C', take $\rho_1 = \rho_2 \circ \mathsf{mgu}(C)$. For all $\theta_1 = \theta_2$ in C, there are two cases. If $\mathsf{mgu}(C)(\theta_1)$ is equal to $\mathsf{mgu}(C)(\theta_2)$, then $\rho_2 \circ \mathsf{mgu}(C)(\theta_1)$ is equal to $\rho_2 \circ \mathsf{mgu}(C)(\theta_2)$. Otherwise, $\mathsf{mgu}(C)(\theta_1) = \mathsf{mgu}(C)(\theta_2)$ belongs to C' by definition of simplification. Since ρ_2 satisfies C' by hypothesis, $\rho_2 \circ \mathsf{mgu}(C)(\theta_1)$ is equal to $\rho_2 \circ \mathsf{mgu}(C)(\theta_2)$. That is, ρ_1 satisfies $\theta_1 = \theta_2$.

Second, we prove that $\forall \overline{\alpha} | C.\theta$ is greater than $\forall \overline{\alpha}' | C'.\theta'$. For all ρ_2 that satisfies C, since $\mathsf{mgu}(C)$ is a most general unifier of C, there exists ρ_1 such that $\rho_2 = \rho_1 \circ \mathsf{mgu}(C)$. For each constraint of C', it is by definition of simplification of the form $\mathsf{mgu}(C)(\theta_1) = \mathsf{mgu}(C)(\theta_2)$ where $\theta_1 = \theta_2$ belongs to C. Therefore it is satisfied by ρ_1, since by hypothesis on ρ_2 that it satisfies $\theta_1 = \theta_2$, $\rho_1 \circ \mathsf{mgu}(C)(\theta_1)$ is equal to $\rho_1 \circ \mathsf{mgu}(C)(\theta_2)$. ■

Using this simplification, we can infer the type of expressions.

Corollary 21 (Principal type for Hindley-Milner) *Let e be a closed expression. Then the principal type of e is $\forall \overline{\alpha}'.\theta'$, where $\langle \mathsf{type}(e) \rangle = \forall \overline{\alpha} | C.\theta$, $\theta' = \mathsf{mgu}(C)(\theta)$ and $\overline{\alpha}' = \overline{\alpha} \cap FV(\theta')$.*

2.1.3 Example

As a first example, let us consider the typing of $(\lambda x.x)\ 1$. By the rules of the algebraic type system, this expression has syntactic type:

$$\begin{array}{rl} & \mathsf{type}((\lambda x.x)\ 1) \\ = & \mathsf{type}(\lambda x.x)\ \mathsf{type}(1) \\ = & (\lambda t_x.\mathsf{type}(x))\ \texttt{int} \\ = & (\lambda t_x.t_x)\ \texttt{int} \end{array}$$

This type can now be translated to a constrained type scheme. We first translate each side of the application: $\langle \lambda t_x.t_x \rangle = \forall \alpha_{t_x} | \emptyset.\, \alpha_{t_x} \rightarrow \alpha_{t_x}$ and $\langle \texttt{int} \rangle = \texttt{int}, \emptyset$. The system of constraints generated by the

application is therefore $\alpha_{t_x} \to \alpha_{t_x} = \texttt{int} \to \alpha$, and the considered expression has the constrained type scheme $\forall \alpha_{t_x} \alpha | \{ \alpha_{t_x} \to \alpha_{t_x} = \texttt{int} \to \alpha \}.\alpha$. By Lemma 20 (SIMPLIFICATION) this type simplifies to $\texttt{int}, \emptyset$.

As a second example, let us now consider the expression $\texttt{let } id \texttt{ be } \lambda x.x \texttt{ in } \lambda yz.z\ (id\ 1)\ (id\ \texttt{false})$.

$$\begin{array}{rl} & \texttt{type}(\texttt{let } id \texttt{ be } \lambda x.x \texttt{ in } \lambda yz.z\ (id\ 1)\ (id\ \texttt{false})) \\ = & \texttt{let } t_{id} \texttt{ be } \lambda t_x.t_x \texttt{ in } \lambda t_y t_z.t_z\ (t_{id}\ \texttt{int})\ (t_{id}\ \texttt{bool}) \end{array}$$

By Definition 14, its translation is:

$$\begin{array}{ll} & \langle \texttt{let } t_{id} \texttt{ be } \lambda t_x.t_x \texttt{ in } \lambda t_y t_z.t_z\ (t_{id}\ \texttt{int})\ (t_{id}\ \texttt{bool}) \rangle \\ = & \langle \lambda t_y t_z.t_z\ (t_{id}\ \texttt{int})\ (t_{id}\ \texttt{bool})\ [t_{id} \leftarrow \forall \alpha_{t_x} | \emptyset.\alpha_{t_x}] \rangle \\ = & \langle \lambda t_y t_z.t_z\ ((\forall \alpha_{t_x} | \emptyset.\alpha_{t_x})\ \texttt{int})\ ((\forall \alpha_{t_{x'}} | \emptyset.\alpha_{t_{x'}})\ \texttt{bool}) \rangle \\ = & \forall \alpha_1 \alpha_2 \alpha_3 \alpha_4 \alpha_{t_y} \alpha_{t_z} \alpha_{t_x} \alpha_{t_{x'}} | \\ & \{ \alpha_{t_x} \to \alpha_{t_x} = \texttt{int} \to \alpha_1,\ \alpha_{t_{x'}} \to \alpha_{t_{x'}} = \texttt{bool} \to \alpha_2,\ \alpha_{t_y} \to (\alpha_{t_z} \to \alpha_{t_z}) = \alpha_1 \to \alpha_3, \alpha_3 = \alpha_2 \to \alpha_4 \}.\alpha_4 \end{array}$$

The most general unifier of that constraint maps α_1, α_{t_y} and α_{t_x} to $\texttt{int}$, α_2, α_4, α_{t_z} and $\alpha_{t_{x'}}$ to $\texttt{bool}$ and α_3 to $\texttt{bool} \to \texttt{bool}$. Therefore, by Lemma 20 (SIMPLIFICATION), this translated type is equivalent to $\texttt{bool}$, which is consequently the type of the expression, as expected.

2.1.4 Beyond Core-ML

So far, we have only considered features of Core-ML. A real language usually has more features. Data-types and pattern matching can be added easily as in ML. Each data-type declaration can be considered as syntactic sugar for the introduction of a new base type name, constructors for the different cases and a matching operator. Pattern-matching can then be seen as syntactic sugar for the application of the matching operator, including matching arbitrarily deep structures, default cases and textual ordering of the branches.

Exceptions and references could also be added to the core language. However, references are not a simple instance of the framework, but require an extension to the framework. As for ML, one could augment the semantics with a global store, and provide references via primitives; this would also require restriction of polymorphism to values. This restriction can be handled in the type algebra, through the translation function defined in this section to instantiate the framework. That is, given the primitive operator $\texttt{ref}$ of type $\forall \alpha.\, \alpha \to \texttt{ref}\ \alpha$, we can modify our translation function so that type variables that appear inside a $\texttt{ref}$ type constructor are not present in the quantifiers. Therefore the typing part of the framework can be left untouched. It would actually be interesting to study how could the semantics itself be made a parameter of our system, so that features requiring specific semantics could be added without modifying the framework at all.

2.2 $\mathrm{ML}_{\leq}$

$\mathrm{ML}_{\leq}$ [5, 6] is a rank-1 polymorphic constrained type system. It has been developed to type an extension of ML with multi-methods and object-orientation. The $\mathrm{ML}_{\leq}$ type system is especially adapted to type an object-oriented language because:

- constraints allow to model atomic subtyping, which can express the "sub-class" relationships;
- (parametric) polymorphism allows to define generic classes and operations;
- its open-world properties fit well with separate compilation of program modules that can define new types.

However, the whole $\mathrm{ML}_{\leq}$ system requires type annotations on lambda-expressions and thus lacks type inference. Furthermore, the presentation in [5, 6] is *ad-hoc* and rather unusual, making it difficult to study.

Therefore, in this section, we focus on the type system itself. We observe that $\text{ML}_\leq$ types form a type algebra that can be used with the algebraic type system of Section 1. We shall use this instance in Section 8.6 to model multi-methods as a concrete instantiation of the generic functions of Chapter 8.

We first recall the definition of the $\text{ML}_\leq$ type system. The, we show in Section 2.2.4 how it can be considered as an instance of the algebraic type system.

2.2.1 Type structure

Type-checking in $\text{ML}_\leq$ is done with respect to a type structure $\mathcal{T}$. The syntax for type structures is given in Figure 2.3. A type structure is a partially ordered set $\mathcal{C}$ of type constructor constants c_V. Type constructors can be in sub-typing relation, written $\mathcal{T} \vdash c_V \leq c'_V$. Type constructors are annotated by their variance V. A variance is a tuple over $\{\oplus, \ominus, \otimes\}$, which stand for co-, contra-, and non-variant type parameters respectively. Only type constructors of the same variance can be in sub-typing relation. That is, $c_{V_1} \leq c'_{V_2}$ implies $V_1 = V_2$.

Type structure	$\mathcal{T}$	$::=$	$(\mathcal{C}, \leq)$
Single variance	v	$::=$	$\oplus \mid \ominus \mid \otimes$
Variance	V	$::=$	$\overline{v}$
Type constructor constant	c_V	$\in$	$\mathcal{C}$
Ground monotype	θ^g	$::=$	$c_V[\overline{\theta^g}]$

Figure 2.3: Type structure syntax

A ground monotype θ^g is built by the application of a type constructor c_V to a list of ground monotypes $\overline{\theta^g}$, and is written $c_V[\overline{\theta^g}]$. In particular, if c_V is a nullary type constructor, then $c_V[]$ is a ground monotype. In that case, we will omit the brackets and denote this monotype by c_V. We require that monotypes built on type constructors respect the arity of their variance: in $c_V[\overline{\theta^g}]$, the number of elements in $\overline{\theta^g}$ must match the arity of variance V.

We will omit the annotation on type constructors when it is obvious from the context. We assume the existence an arrow type constructor $\rightarrow_{(\ominus,\oplus)}$ used to represent functional types. As usual the arrow is contra-variant on its domain and co-variant on its codomain.

Definition 22 (Variant subtyping) *The notation* $(\theta_1, ..., \theta_n) \leq_{(v_1,...,v_n)} (\theta'_1, ..., \theta'_n)$ *stands for the set* $(\theta_1 \leq_{v_1} \theta'_1) \cup ... \cup (\theta_n \leq_{v_n} \theta'_n)$*, where:*

$$\begin{aligned} \theta_1 \leq_\oplus \theta_2 &= \{\theta_1 \leq \theta_2\} \\ \theta_1 \leq_\ominus \theta_2 &= \{\theta_2 \leq \theta_1\} \\ \theta_1 \leq_\otimes \theta_2 &= \{\theta_1 \leq \theta_2, \theta_2 \leq \theta_1\} \end{aligned}$$

Definition 23 (Ground subtyping) *We write* $\mathcal{T} \vdash \theta^g \leq \theta'^g$ *the subtyping on ground monotypes. The relation* $\mathcal{T} \vdash c_V[\overline{\theta^g}] \leq c'_V[\overline{\theta'^g}]$ *holds if and only if* $\mathcal{T} \vdash c_V \leq c'_V$ *and for all* $\theta \leq \theta'$ *in* $\overline{\theta^g} \leq_V \overline{\theta'^g}$, $\mathcal{T} \vdash \theta \leq \theta'$.

For instance, the $\text{ML}_\leq$ type structure corresponding to the example of Section 1.1.2 is $\mathcal{T} = (\{\texttt{int}_{()}, \texttt{float}_{()}, \rightarrow_{(\ominus,\oplus)}\}, \{\texttt{int} \leq \texttt{float}\})$. It then follows from Definition 23 that, for instance, $\mathcal{T} \vdash \texttt{float} \rightarrow \texttt{int} \leq \texttt{int} \rightarrow \texttt{float}$.

2.2.2 Constraints

The syntax for constraints and monotypes is given in Figure 2.4. Monotypes are similar to the ground monotypes defined in Section 2.2.1, with the addition of type constructor variables and monotype variables.

Type constructor variables can stand for type constructor constants while monotype variables can stand for arbitrary monotypes. It is possible to quantify over these two flavors of variables. For instance, assuming a covariant `list` type constructor, a syntactically valid constraint implication is $\forall t_\oplus, u.\ t_\oplus \leq \texttt{list} \wedge u \leq \texttt{int} \models t_\oplus[u] \leq \texttt{list}[\texttt{int}]$.

Type constructor	ϕ_V	::=	
Type constructor constant			c_V
Type constructor variable		|	t_V
Monotype	θ	::=	
Monotype variable			t
Constructed monotype		|	$\phi_V[\overline{\theta}]$
Constraint	κ	::=	$\overline{\theta \leq \theta} \mid \phi_V \leq \phi_V$
Variable list	ϑ	::=	$\overline{t \mid t_V}$
Constraint implication			$\forall \vartheta.\ \overline{\kappa_1} \models \overline{\kappa_2}$

Figure 2.4: Constraint syntax

For convenience, we will freely consider constraint sets as conjuncts of constraints, by writing *true* for the empty set of constraints, $\kappa_1 \wedge \kappa_2$ instead of $\kappa_1 \cup \kappa_2$, and allow κ to denote a set of constraints. Therefore, we have in particular that $\kappa_1 \wedge \kappa_1$ is identical to κ_1, that $\kappa_1 \wedge \kappa_2$ is identical to $\kappa_2 \wedge \kappa_1$, and that $\kappa_1 \wedge (\kappa_2 \wedge \kappa_3)$ is identical to $(\kappa_1 \wedge \kappa_2) \wedge \kappa_3$.

We now define the notion of constraint implication with the predicate $\mathcal{T} \vdash \forall \vartheta.\ \kappa_1 \models \kappa_2$, which reads "in type structure $\mathcal{T}$, for all ϑ, constraint κ_1 implies constraint κ_2". For the intuition, it is important to note that the universal quantification over ϑ applies to both κ_1 and κ_2. This predicate is defined as the least predicate verifying the axioms of Figure 2.5. This definition is equivalent to the original presentation of $\mathrm{ML}_\leq$ [5].

Often, the type structure can be left implicit and we will simply write $\forall \vartheta.\ \kappa_1 \models \kappa_2$.

Intuitively, the relation $\forall \vartheta.\ \kappa_1 \models \kappa_2$ holds if, for every valuation of the variables in ϑ such that κ_1 is satisfied, there exists a valuation of the other variables such that κ_2 is satisfied. However, it would not be desirable to have that property in a closed-world setting. For instance, in a type structure with a single type constructor A, the relation $\forall t.\ t \leq A \models A \leq t$ should not hold, although the only known valuation for t is A. Otherwise, it would become impossible to define a subclass of A in a different module. In Chapter 7, we will formally consider modular type-checking, which involves extending type structures while preserving the soundness of some previously type-checked code. In particular, Definition 55 is a semantic interpretation of this constraint implication. We will also characterize how the constraint language can be extended to make the type system more expressive, and list the properties that must hold for such extensions to be valid.

Trans states that constraint implication is transitive. Triv states that a constraint implies any subset of itself. VarIntro states that a given constraint κ is implied by any constraint obtained by instantiation of variables of κ not in the quantified set ϑ. Intuitively, this is correct since for every valuation of the variables in ϑ such that $\sigma(\kappa)$ is satisfied, κ can indeed be satisfied by instantiating its variables using σ. The next four rules deal with monotypes, as emphasized by the prefix M in their names. MRef and MTrans state the reflexivity and transitivity of monotype subtyping. MIntro and MElim express the relation between subtyping of constructed monotypes and subtyping of their components, as in Definition 23. The next three rules state the properties of type constructors: the ordering of type constructors is reflexive and transitive, and ordering of ground type constructors can be used when it is present in the context $\mathcal{T}$. Finally, VElim states that the constraints are structural: if a type variable is comparable to a constructed monotype, then it must have the same shape. That is, it is built on a type constructor of the same variance.

As an illustration, let us prove that the constraint implication $\forall t.\ \texttt{int} \to \texttt{int} \leq t \models u \to u \leq t \wedge u \leq \texttt{float}$ holds in the example type structure defined above.

By CStruct we have $\forall t.\ \texttt{int} \to \texttt{int} \leq t \models \texttt{int} \to \texttt{int} \leq t \wedge \texttt{int} \leq \texttt{float}$. Furthermore, applying VarIntro with σ being the substitution that maps u to `int` and leaves all other variables unchanged, we

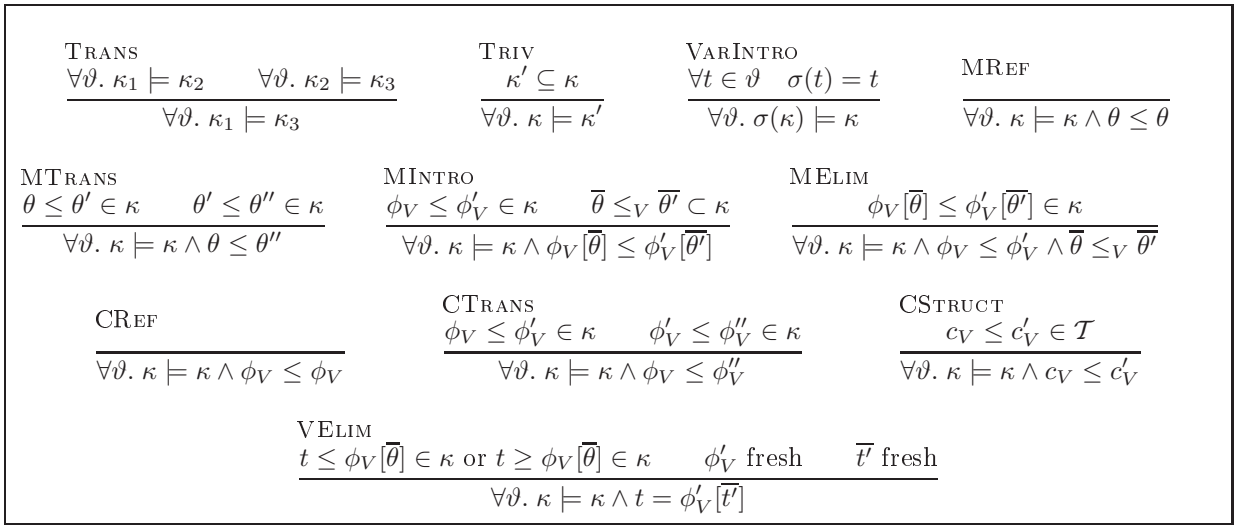


Figure 2.5: Axioms of constraint implication

have $\forall t.\ \texttt{int} \to \texttt{int} \leq t \wedge \texttt{int} \leq \texttt{float} \models u \to u \leq t \wedge u \leq \texttt{float}$. Finally, we can apply Trans to get the desired implication.

Since many proofs will include chains of implications linked by the transitivity rule, we will often write them in a more condensed form by leaving the use of transitivity implicit. For instance, the above proof can also be written as:

$\forall t.$
$\quad \texttt{int} \to \texttt{int} \leq t$
$\models \quad \texttt{int} \to \texttt{int} \leq t \wedge \texttt{int} \leq \texttt{float}$ (CStruct)
$\models \quad u \to u \leq t \wedge u \leq \texttt{float}$ (VarIntro with $\sigma = \texttt{id} + \{u \mapsto \texttt{int}\}$)

We will use the following three properties, which are proved in [5].

The first one shows that it is always possible to make the set of quantified variables of a constraint implication smaller:

Lemma 24 ($\forall$E)

$$\frac{\forall\vartheta.\ \kappa \models \kappa' \qquad \vartheta' \subset \vartheta}{\forall\vartheta'.\ \kappa \models \kappa'}$$

Proof of lemma 24

The only rules in which quantified variables sets play a role are VarIntro and VElim. For VarIntro, the condition $\forall t \in \vartheta \quad \sigma(t) = t$ trivially implies $\forall t \in \vartheta' \quad \sigma(t) = t$ when ϑ' is a subset of ϑ. Therefore $\forall\vartheta.\ \sigma(\kappa) \models \kappa$ holds when $\forall\vartheta'.\ \sigma(\kappa) \models \kappa$ holds. Similarly, for VElim, the freshness condition on ϕ'_V and $\overline{t'}$ is only weakened by using a smaller qualified set of variables. A structural induction for the case Trans finishes the proof. ■

Conversely, one can add quantified variables that do not appear on the right hand side of the implication:

Lemma 25 ($\forall$I)

$$\frac{\forall\vartheta.\ \kappa \models \kappa' \qquad FV(\kappa') \cap \vartheta' = \emptyset}{\forall\vartheta, \vartheta'.\ \kappa \models \kappa'}$$

Lemma 25 can be proved by noting that since κ' has no free variable in ϑ', it is possible to modify the proof of $\forall\vartheta.\ \kappa \models \kappa'$ so that it does not introduce any variable in ϑ', after a renaming of κ. Therefore, there is also a proof of $\forall\vartheta, \vartheta'.\ \kappa \models \kappa'$.

It is possible to combine two implications by conjunction, provided that all the common variables of the right hand sides are quantified over:

Lemma 26 (Conjunction)

$$\frac{\forall\vartheta.\ \kappa_1 \models \kappa_1' \qquad \forall\vartheta.\ \kappa_2 \models \kappa_2'}{\forall\vartheta.\ \kappa_1 \wedge \kappa_2 \models \kappa_1' \wedge \kappa_2'}\ FV(\kappa_1') \cap FV(\kappa_2') \subset \vartheta$$

The sketch of the proof is the following. First, if κ_1 and κ_2 share variables not in ϑ, these can be renamed in κ_2. This ensures together with the hypothesis $FV(\kappa_1') \cap FV(\kappa_2') \subset \vartheta$ that we can now assume the derivations of $\forall\vartheta.\ \kappa_1 \models \kappa_1'$ and $\forall\vartheta.\ \kappa_2 \models \kappa_2'$ only share variables in ϑ **(1)**. Then we check that for every step of the proof of $\forall\vartheta.\ \kappa_1 \models \kappa_1'$, we can add κ_2 to both sides of the implication. This is only non-trivial in the case of an application of VarIntro. In that case, we want to prove $\forall\vartheta.\ \sigma(\kappa) \wedge \kappa_2 \models \kappa \wedge \kappa_2$. But since we could suppose **(1)**, σ must leave κ_2 invariant. So this is equivalent to $\forall\vartheta.\ \sigma(\kappa \wedge \kappa_2) \models \kappa \wedge \kappa_2$, which is another instance of VarIntro. We have thus proved $\forall\vartheta.\ \kappa_1 \wedge \kappa_2 \models \kappa_1' \wedge \kappa_2$. Similarly, we can prove $\forall\vartheta.\ \kappa_1' \wedge \kappa_2 \models \kappa_1' \wedge \kappa_2'$, which finishes the proof.

2.2.3 Constrained types

A $\mathrm{ML}_{\leq}$ type is a constrained monotype, as defined in Figure 2.6. The $\mathrm{ML}_{\leq}$ type $\forall\vartheta.\kappa \Rightarrow \theta$ is *well-formed* if and only if $\forall\emptyset.\ true \models \kappa$ holds, that is, if the constraint κ is satisfiable.

$$\textit{Type} \quad \tau \quad ::= \quad \forall\vartheta.\kappa \Rightarrow \theta$$

Figure 2.6: Constrained types

We can now define a partial order on $\mathrm{ML}_{\leq}$ types.

Definition 27 (Subtyping in $\mathrm{ML}_{\leq}$)
Let τ_1 be $\forall\vartheta_1.\kappa_1 \Rightarrow \theta_1$ and τ_2 be $\forall\vartheta_2.\kappa_2 \Rightarrow \theta_2$. , then $\mathcal{T} \vdash \tau_1 \leq_{\mathrm{ML}_{\leq}} \tau_2$ holds. $\mathcal{T} \vdash \tau_1 \leq_{\mathrm{ML}_{\leq}} \tau_2$ holds if and only if the constraint implication

$$\mathcal{T} \vdash \forall FV(\tau_1), FV(\tau_2), t.\ \kappa_2 \wedge \theta_2 \leq t \models \kappa_1 \wedge \theta_1 \leq t$$

holds for a fresh variable t.

Informally, this definition can be interpreted as follows: given an arbitrary value for the type variables in the context (corresponding to $\forall FV(\tau_1), FV(\tau_2)$), for any monotype t, if there is a ground instance of type τ_2 that is a subtype of t (that is $\kappa_2 \wedge \theta_2 \leq t$), then there is a ground instance of type τ_1 that is a subtype of t. Therefore, τ_1 is a more precise type than τ_2, that is to say that $\tau_1 \leq \tau_2$.

For instance, in the example type structure defined page 41, it is true that $\forall u.\, u \leq \texttt{float} \Rightarrow u \to u \leq \texttt{int} \to \texttt{int}$. This amounts to the constraint implication $\forall t.\ \texttt{int} \to \texttt{int} \leq t \models u \leq \texttt{float} \wedge u \to u \leq t$, which was proved to hold in Section 2.2.2.

Subtyping and well-formedness have been proved decidable in [5].

2.2.4 Instantiation of the framework

We now show that $\mathrm{ML}_{\leq}$ types form a type algebra. The set of algebraic of types MLS is the set of $\mathrm{ML}_{\leq}$ types denoted by τ. We have to define the pre-order on $S(MLS)$. Since $\mathrm{ML}_{\leq}$ types already include constraints, they are powerful enough to represent all syntactic types. Formally, we define a translation function from $S(MLS)$ to MLS and use it to lift the subtyping relation to $S(MLS)$.

$$\langle \forall \vartheta.\, \kappa \Rightarrow \theta \rangle = \forall \vartheta.\, \kappa \Rightarrow \theta \qquad \langle t \rangle = \forall \emptyset.\, true \Rightarrow t \qquad \langle \mathbb{E} \rangle = \forall t.\, t \leq t \to t \Rightarrow t \qquad \frac{\langle \tau \rangle = \forall \vartheta.\, \kappa \Rightarrow \theta}{\langle \lambda t.\tau \rangle = \forall \vartheta, t.\, \kappa \Rightarrow t \to \theta}$$

$$\frac{\langle \tau_1 \rangle = \forall \vartheta_1.\, \kappa_1 \Rightarrow \theta_1 \qquad \langle \tau_2 \rangle = \forall \vartheta_2.\, \kappa_2 \Rightarrow \theta_2}{\langle \tau_1\ \tau_2 \rangle = \forall \vartheta_1, \vartheta_2, t.\, \kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq (\theta_2 \to t) \Rightarrow t}\ t \notin \vartheta_1, \vartheta_2, FV(\kappa_1), FV(\kappa_2), FV(\theta_1), FV(\theta_2)$$

$$\frac{}{\langle \texttt{let}\ t_1\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2 \rangle = \langle \tau_2 [t_1 \leftarrow \tau_1] \rangle}\ t_1 \in FV(\tau_2)$$

$$\frac{\langle \tau_1 \rangle = \forall \vartheta_1.\, \kappa_1 \Rightarrow \theta_1 \qquad \langle \tau_2 \rangle = \forall \vartheta_2.\, \kappa_2 \Rightarrow \theta_2}{\langle \texttt{let}\ t_1\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2 \rangle = \forall \vartheta_1, \vartheta_2.\, \kappa_1 \wedge \kappa_2 \Rightarrow \theta_2}\ t_1 \notin FV(\tau_2)$$

Figure 2.7: Translation for $\mathrm{ML}_{\leq}$

Definition 28 (Translation of $\mathrm{ML}_{\leq}$ types) *The translation function $\langle \cdot \rangle$ from $S(MLS)$ to MLS is defined by cases in Figure 2.7.*

Algebraic types translate to themselves. The translation of a type variable is an unconstrained type, whose monotype component is the type variable. The translation of $\mathbb{E}$ is an arbitrary ill-formed $\mathrm{ML}_{\leq}$ type. By Definition 27, any type whose translation is ill-formed is equivalent to $\mathbb{E}$. The translation of a lambda type is generalized over the type of the argument. Since $\mathrm{ML}_{\leq}$ types are equal up to α-conversion, the definition is independent of the choice of a name for the bound type variable, which must not appear in ϑ. The translation of an application type is done by constraining the monotype of the function to be a subtype of an arrow type whose domain is the monotype of the argument. By α-conversion, we assume that ϑ_1 and ϑ_2 are disjoint and we choose a t that does not appear in any of them. We translate let types differently whether the bound type variable appears free in the body of the type or not, for the same reason as in the translation for Hindley-Milner in Section 2.1. In the case where it does appear free, we define the translation of $\texttt{let}\ t_1\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2$ as the translation of $\tau_2 [t_1 \leftarrow \tau_1]$. This definition is well founded thanks to Definition 19. Moreover, we show in Lemma 32 that we can equivalently define it as the translation of $\tau_2 [t_1 \leftarrow \langle \tau_1 \rangle]$.

Definition 29 (Order on $S(MLS)$) *Given a type structure $\mathcal{T}$, for all syntactic types τ_1 and τ_2 in $S(MLS)$, $\mathcal{T} \vdash \tau_1 \leq \tau_2$ holds if and only if $\mathcal{T} \vdash \langle \tau_1 \rangle \leq_{\mathrm{ML}_{\leq}} \langle \tau_2 \rangle$ holds.*

This definition allows to identify syntactic types τ to their algebraic translation $\langle \tau \rangle$ and to use the meta-variable τ for both.

Given a type structure $\mathcal{T}$, we can now built a canonical type algebra based on $\mathcal{T}$. We will write $\mathcal{A}(\mathcal{T})$ for the couple $(S(MLS), \mathcal{T} \vdash \cdot \leq \cdot)$. Our main result in this section is that $\mathcal{A}(\mathcal{T})$ is indeed a type algebra.

Theorem 30 (ML-Sub) *For all type structure $\mathcal{T}$, $\mathcal{A}(\mathcal{T})$ is a type algebra.*

Before proving this theorem, we first characterize the $\mathrm{ML}_{\leq}$ types that are the result of the translation of a syntactic type.

Lemma 31 (Translation) *Let τ be a syntactic type, in which t possibly appears free, and $\langle \tau \rangle$ be $\forall \vartheta.\, \kappa \Rightarrow \theta$. Let ϑ' be a variable list, κ' be a $\mathrm{ML}_{\leq}$ constraint and θ' be a $\mathrm{ML}_{\leq}$ monotype. Then $\forall \vartheta, t, \vartheta'.\, \kappa \wedge \kappa' \wedge \theta' \leq t \Rightarrow \theta$ is greater or equal to $\forall \vartheta, \vartheta'.\, \kappa [t \leftarrow \theta'] \wedge \kappa' \Rightarrow \theta [t \leftarrow \theta']$.*

Informally, this lemma states that the translation of syntactic type variables always occurs covariantly in the $\mathrm{ML}_{\leq}$ translated type.

Proof of lemma 31 (Translation)

Let ϑ_0 be $FV(\forall\vartheta, t, \vartheta'.\, \kappa \wedge \kappa' \wedge \theta' \leq t \Rightarrow \theta) \cup FV(\forall\vartheta, \vartheta'.\, \kappa\,[t \leftarrow \theta'] \wedge \kappa' \Rightarrow \theta\,[t \leftarrow \theta'])$. Then the proposition is by Definition 29 and Definition 27 equivalent to: $\forall u\vartheta_0.\, \kappa \wedge \kappa' \wedge \theta' \leq t \wedge \theta \leq u \models \kappa\,[t \leftarrow \theta'] \wedge \kappa' \wedge \theta\,[t \leftarrow \theta'] \leq u$ for a fresh u. We prove a generalization of this implication quantified by $u\vartheta'_0$ for an arbitrary superset ϑ'_0 of ϑ_0, by induction on the size of τ:

case $\tau = a$ **or** $\tau = t'$ **or** $\tau = \mathbb{E}$
The type variable t is not free in τ, so it does not appear in κ nor in θ. Therefore this is a simple application of the TRIV axiom.

case $\tau = t$
That is to say that $\theta = t$ and $\kappa = \mathit{true}$. We have to prove $\forall u\vartheta'_0.\, \kappa' \wedge \theta' \leq t \wedge t \leq u \models \kappa' \wedge \theta' \leq u$, which is an instance of MTRANS.

case $\tau = \lambda t_1.\tau_1$
Let $\langle\tau_1\rangle$ be $\forall\vartheta_1.\, \kappa_1 \Rightarrow \theta_1$. Then by Definition 28, $\langle\lambda t_1.\tau_1\rangle = \forall\vartheta_1, t_1.\, \kappa_1 \Rightarrow t_1 \to \theta_1$. By induction hypothesis, $\forall u'\vartheta'_0.\, \kappa_1 \wedge \kappa' \wedge \theta' \leq t \wedge \theta_1 \leq u' \models \kappa_1\,[t \leftarrow \theta'] \wedge \kappa' \wedge \theta_1\,[t \leftarrow \theta'] \leq u'$, so by Lemma 25 this also holds $\forall u'\vartheta'_0 u''u$. By Lemma 26 with $u = u'' \to u'$ we get $\forall u'\vartheta'_0 u''u.\, \kappa_1 \wedge \kappa' \wedge \theta' \leq t \wedge u = u'' \to u' \wedge \theta_1 \leq u' \models \kappa_1\,[t \leftarrow \theta'] \wedge \kappa' \wedge u = u'' \to u' \wedge \theta_1\,[t \leftarrow \theta'] \leq u'$ (**1**). Therefore,

$$
\begin{array}{lll}
\forall u\vartheta'_0 u''u. & & \\
\quad \kappa_1 \wedge \kappa' \wedge \theta' \leq t \wedge t_1 \to \theta_1 \leq u & & \\
\models \quad \kappa_1 \wedge \kappa' \wedge \theta' \leq t \wedge u = u'' \to u' \wedge u'' \leq t_1 \wedge \theta_1 \leq u' & & (\text{VELIM}, \text{MELIM}) \\
\models \quad \kappa_1\,[t \leftarrow \theta'] \wedge \kappa' \wedge u = u'' \to u' \wedge u'' \leq t_1 \wedge \theta'\,[t \leftarrow \theta'] \leq u' & & (1) \\
\models \quad \kappa_1\,[t \leftarrow \theta'] \wedge \kappa' \wedge t_1 \to \theta_1\,[t \leftarrow \theta'] \leq u'' \to u' = u & & (\text{MINTRO})
\end{array}
$$

We conclude by applying Lemma 24.

case $\tau = \tau_1\ \tau_2$
Let $\langle\tau_1\rangle$ be $\forall\vartheta_1.\, \kappa_1 \Rightarrow \theta_1$ and $\langle\tau_2\rangle$ be $\forall\vartheta_2.\, \kappa_2 \Rightarrow \theta_2$. Then by Definition 28, $\langle\tau\rangle = \forall\vartheta_1, \vartheta_2, v.\, \kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq (\theta_2 \to v) \Rightarrow v$. By induction hypothesis on τ_1 and τ_2, for $i = 1, 2$, $\forall u_i\vartheta'_i.\, \kappa_i \wedge \kappa'_i \wedge \theta' \leq t \wedge \theta_i \leq u_i \models \kappa_i\,[t \leftarrow \theta'] \wedge \kappa' \wedge \theta_i\,[t \leftarrow \theta'] \leq u_i$ for arbitrary ϑ'_i and κ'_i. In particular, we apply this hypothesis with $i = 1$, $\vartheta'_1 = u\vartheta'_0$ and $\kappa'_1 = \kappa_2\,[t \leftarrow \theta'] \wedge \kappa' \wedge u_1 = \theta_2\,[t \leftarrow \theta'] \to t_1 \wedge t_1 \leq u$ and remove the quantification on u_1 by Lemma 24 (RQ) (**1**) and with $i = 2$, $\vartheta'_2 = u\vartheta'_0$ and $\kappa'_2 = \kappa' \wedge \theta' \leq t \wedge \theta_1 = u_2 \to u_1 \wedge \kappa_1 \wedge u_1 \leq t_1 \wedge t_1 \leq u$ and remove the quantification on u_2 by Lemma 24 (RQ) (**2**). Therefore,

$$
\begin{array}{lll}
\forall u\vartheta'_0. & & \\
\quad \kappa_1 \wedge \kappa_2 \wedge \kappa' \wedge \theta' \leq t \wedge \theta_1 \leq\ \theta_2 \to t_1 \wedge t_1 \leq u & & \\
\models \quad \left\{\begin{array}{l} \kappa_2 \wedge \kappa' \wedge \theta' \leq t \wedge \theta' \leq t \wedge \theta_2 \leq u_2 \\ \wedge\, \theta_1 = u_2 \to u_1 \wedge \kappa_1 \wedge u_1 \leq t_1 \wedge t_1 \leq u \end{array}\right. & & (\text{VELIM},\ \text{MELIM}) \\
\models \quad \left\{\begin{array}{l} \kappa_2\,[t \leftarrow \theta'] \wedge \kappa' \wedge \theta' \leq t \wedge \theta_2\,[t \leftarrow \theta'] \leq u_2 \\ \wedge\, \theta_1 = u_2 \to u_1 \wedge \kappa_1 \wedge u_1 \leq t_1 \wedge t_1 \leq u \end{array}\right. & & (2) \\
\models \quad \left\{\begin{array}{l} \kappa_2\,[t \leftarrow \theta'] \wedge \kappa' \wedge \theta' \leq t \wedge \theta_1 \leq (\,\theta_2\,[t \leftarrow \theta'] \to t_1) \\ \wedge\, \kappa_1 \wedge t_1 \leq u \end{array}\right. & & (\text{MINTRO}) \\
\models \quad \left\{\begin{array}{l} \kappa_2\,[t \leftarrow \theta'] \wedge \kappa' \wedge \theta' \leq t \wedge \theta_1 \leq v \\ \wedge\, v = (\,\theta_2\,[t \leftarrow \theta'] \to t_1) \wedge \kappa_1 \wedge t_1 \leq u \end{array}\right. & & (\text{VARINTRO with } v \mapsto \theta_2\,[t \leftarrow \theta'] \to t_1) \\
\models \quad \left\{\begin{array}{l} \kappa_2\,[t \leftarrow \theta'] \wedge \kappa' \wedge \kappa_1\,[t \leftarrow \theta'] \\ \wedge\, \theta_1\,[t \leftarrow \theta'] \leq v = (\,\theta_2\,[t \leftarrow \theta'] \to t_1) \wedge t_1 \leq u \end{array}\right. & & (1) \\
\models \quad (\kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq\ \theta_2 \to t_1)\,[t \leftarrow \theta'] \wedge \kappa' \wedge t_1 \leq u & & (\text{MTRANS})
\end{array}
$$

case $\tau = \texttt{let}\ t_1\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2$ **where** $t_1 \notin FV(\tau_2)$
Let $\langle\tau_1\rangle$ be $\forall\vartheta_1.\, \kappa_1 \Rightarrow \theta_1$ and $\langle\tau_2\rangle$ be $\forall\vartheta_2.\, \kappa_2 \Rightarrow \theta_2$. Then by Definition 28, $\langle\tau\rangle = \forall\vartheta_1, \vartheta_2.\, \kappa_1 \wedge \kappa_2 \Rightarrow \theta_2$. Let u_1 and u_2 be fresh variables. By induction hypothesis on τ_1 and τ_2, for $i = 1, 2$, $\forall u_i\vartheta'_0.\, \kappa_i \wedge \kappa'_i \wedge \theta' \leq$

$t \wedge \theta_i \leq u_i \models \kappa_i\,[t \leftarrow \theta'] \wedge \kappa' \wedge \theta_i\,[t \leftarrow \theta'] \leq u_i$ for an arbitrary κ'_i. By MRef on θ_1 and VarIntro on u_1, $\forall u_2 \vartheta'_0.\ \kappa_1 \wedge \kappa_2 \wedge \kappa' \wedge \theta' \leq t \wedge \theta_2 \leq u \models \kappa_1 \wedge \kappa_2 \wedge \kappa' \wedge \theta' \leq t \wedge \theta_2 \leq u \wedge \theta_1 \leq u_1$

By induction hypothesis on τ_1, removing the quantification on u_1 by Lemma 24,

$$
\begin{array}{lll}
\forall u_2 \vartheta'_0. & & \\
\quad \kappa_1 \wedge \kappa_2 \wedge \kappa' \wedge \theta' \leq t \wedge \theta_2 \leq u_2 \wedge \theta_1 \leq u_1 & & \\
\models \quad \kappa_1\,[t \leftarrow \theta'] \wedge \kappa_2 \wedge \kappa' \wedge \theta' \leq t \wedge \theta_2 \leq u_2 \wedge \theta_1\,[t \leftarrow \theta'] \leq u_1 & & \\
\models \quad \kappa_1\,[t \leftarrow \theta'] \wedge \kappa_2 \wedge \kappa' \wedge \theta' \leq t \wedge \theta_2 \leq u_2 & & (\textsc{Triv}) \\
\models \quad \kappa_1\,[t \leftarrow \theta'] \wedge \kappa_2\,[t \leftarrow \theta'] \wedge \kappa' \wedge \theta_2\,[t \leftarrow \theta'] \leq u_2 & & (\text{Ind. Hyp. } 2) \\
\models \quad (\kappa_1 \wedge \kappa_2)\,[t \leftarrow \theta'] \wedge \kappa' \wedge \theta_2\,[t \leftarrow \theta'] \leq u_2 & &
\end{array}
$$

case $\tau = \texttt{let}\ t_1\ \texttt{be}\ \tau_1\ \texttt{in}\ \tau_2$ **where** $t_1 \in FV(\tau_2)$

Then by Definition 28, $\langle\tau\rangle = \langle\tau_2\,[t \leftarrow \tau_1]\rangle$, so the property is true by induction hypothesis.

■

Proof of theorem 30 (ML-Sub)

- We first prove property **ii** of Definition 4 (Covariance). We therefore assume $\tau_1 \leq \tau_2$ and we prove that $\tau\,[t \leftarrow \tau_1] \leq \tau\,[t \leftarrow \tau_2]$. The proof is by induction on the size of τ, using the axioms of constraint implication. The cases of algebraic types, type variables and the error type are trivial.

  case $\tau = \lambda t_0.\tau_0$

  We can assume w.l.o.g. that t_0 is different from t **(1)** and not free in τ **(2)** nor in τ' **(3)**. Therefore, by Definition 3, $(\lambda t_0.\tau_0)\,[t \leftarrow \tau] = \lambda t_0.(\tau_0\,[t \leftarrow \tau])$ and $(\lambda t_0.\tau_0)\,[t \leftarrow \tau'] = \lambda t_0.(\tau_0\,[t \leftarrow \tau'])$. By induction hypothesis, we have $\tau_0\,[t \leftarrow \tau'] \leq \tau_0\,[t \leftarrow \tau]$ **(1)**. Let $\forall \vartheta_1.\ \kappa_1 \Rightarrow \theta_1$ be $\langle\tau_0\,[t \leftarrow \tau]\rangle$ and $\forall \vartheta'_1.\ \kappa'_1 \Rightarrow \theta'_1$ be $\langle\tau_0\,[t \leftarrow \tau']\rangle$. (1) is therefore by Definition 29 and Definition 27 $\forall F, u_1.\ \kappa_1 \wedge \theta_1 \leq u_1 \models \kappa'_1 \wedge \theta'_1 \leq u_1$, where F is $FV(\tau_0\,[t \leftarrow \tau]) \cup FV(\tau_0\,[t \leftarrow \tau'])$. By Lemma 25 applied to fresh variables u and u_0, we have $\forall F, u, u_0, u_1.\ \kappa_1 \wedge \theta_1 \leq u_1 \models \kappa'_1 \wedge \theta'_1 \leq u_1$. Then by Lemma 26 with the trivial implication $\forall F, u, u_0, u_1.\ u = u_0 \rightarrow u_1 \models u = u_0 \rightarrow u_1$, we have $\forall F, u, u_0, u_1.\ \kappa_1 \wedge \theta_1 \leq u_1 \wedge u = u_0 \rightarrow u_1 \models \kappa'_1 \wedge \theta'_1 \leq u_1 \wedge u = u_0 \rightarrow u_1$. Finally, we apply Lemma 24 to get $\forall F \setminus \{t_0\}, u.\ \kappa_1 \wedge \theta_1 \leq u_1 \wedge u = u_0 \rightarrow u_1 \models \kappa'_1 \wedge \theta'_1 \leq u_1 \wedge u = u_0 \rightarrow u_1$ **(4)**. We can now prove the desired property: $\lambda t_0.\tau_0\,[t \leftarrow \tau'] \leq \lambda t_0.\tau_0\,[t \leftarrow \tau]$. By Definition 29 and Definition 27, this amounts to:

  $$
  \begin{array}{lll}
  \forall F \setminus \{t_0\}, u. & & \\
  \quad \kappa_1 \wedge t_0 \rightarrow \theta_1 \leq u & & \\
  \models \quad \kappa_1 \wedge t_0 \geq u_0 \wedge \theta_1 \leq u_1 \wedge u = u_0 \rightarrow u_1 & & (\textsc{VElim}, \textsc{MElim}) \\
  \models \quad \kappa'_1 \wedge t_0 \geq u_0 \wedge \theta'_1 \leq u_1 \wedge u = u_0 \rightarrow u_1 & & \textbf{(4)} \\
  \models \quad \kappa'_1 \wedge t_0 \rightarrow \theta'_1 \leq u & & (\textsc{MIntro})
  \end{array}
  $$

  case $\tau = \tau_1\ \tau_2$

  By Definition 3, $(\tau_1\ \tau_2)\,[t \leftarrow \tau] = (\tau_1\,[t \leftarrow \tau])\ (\tau_2\,[t \leftarrow \tau])$ and $(\tau_1\ \tau_2)\,[t \leftarrow \tau'] = (\tau_1\,[t \leftarrow \tau])\ (\tau_2\,[t \leftarrow \tau'])$. Let $\forall \vartheta_1.\ \kappa_1 \Rightarrow \theta_1$ be $\langle\tau_1\,[t \leftarrow \tau]\rangle$, $\forall \vartheta'_1.\ \kappa'_1 \Rightarrow \theta'_1$ be $\langle\tau'_1\,[t \leftarrow \tau]\rangle$, $\forall \vartheta_2.\ \kappa_2 \Rightarrow \theta_2$ be $\langle\tau_2\,[t \leftarrow \tau]\rangle$, and $\forall \vartheta'_2.\ \kappa'_2 \Rightarrow \theta'_2$ be $\langle\tau'_2\,[t \leftarrow \tau]\rangle$, and let F_1, F'_1, F_2 and F'_2 be the free variables of $\tau_1\,[t \leftarrow \tau]$, $\tau_1\,[t \leftarrow \tau']$, $\tau_2\,[t \leftarrow \tau]$, and $\tau_2\,[t \leftarrow \tau']$ respectively. We can assume w.l.o.g that the bound variables ϑ_1, ϑ'_1, ϑ_2 and ϑ'_2 are all disjoint from each other and from $F_1 \cup F'_1 \cup F_2 \cup F'_2$ **(1)**. We need to prove $(\tau_1\,[t \leftarrow \tau'])\ (\tau_2\,[t \leftarrow \tau']) \leq (\tau_1\,[t \leftarrow \tau])\ (\tau_2\,[t \leftarrow \tau])$. By Definition 29 and Definition 27, this amounts to $\forall F_1, F_2, F'_1, F'_2, u.\ \kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq \theta_2 \rightarrow t \wedge t \leq v \models \kappa'_1 \wedge \kappa'_2 \wedge \theta'_1 \leq \theta'_2 \rightarrow t \wedge t \leq v$.

  $$
  \begin{array}{lll}
  \forall F_1, F_2, F'_1, F'_2, v. & & \\
  \quad \kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq \theta_2 \rightarrow t \wedge t \leq v & & \\
  \models \quad \kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq \theta_1 \wedge \theta_1 \leq \theta_2 \rightarrow t \wedge t \leq v & & (\textsc{MRef}) \\
  \models \quad \kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq u_1 \wedge u_1 \leq \theta_2 \rightarrow t \wedge t \leq v & & (\textsc{VarIntro} \text{ with } u_1 \mapsto \theta_1) \\
  \models \quad \kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq u_1 \wedge u_1 = u_2 \rightarrow u'_2 \wedge \theta_2 \leq u_2 \wedge u'_2 \leq t \leq v & & (\textsc{VElim}, \textsc{MElim})
  \end{array}
  $$

We reference this implication as (**2**). We now use the induction hypothesis, which is $\tau_1[t \leftarrow \tau] \leq \tau_1[t \leftarrow \tau']$ and $\tau_2[t \leftarrow \tau] \leq \tau_2[t \leftarrow \tau']$. By Definition 29 and Definition 27, this amounts to the constraint implications $\forall F_1, F_1', u_1.\ \kappa_1 \wedge \theta_1 \leq u_1 \models \kappa_1' \wedge \theta_1' \leq u_1$, and $\forall F_2, F_2', u_2.\ \kappa_2 \wedge \theta_2 \leq u_2 \models \kappa_2' \wedge \theta_2' \leq u_2$. In the first implication, the variables appearing in the right hand side are either the free variables F_1' of $\forall \vartheta_1'.\ \kappa_1' \Rightarrow \theta_1'$, which are already quantified over, or the bound variables ϑ_1', which are disjoint from ϑ_2', F_2 and F_2' by (1). The same argument applies the second implication. Therefore, we can extend both implications to $\forall F_1, F_2, F_1', F_2', u_1, u_2, u$ by Lemma 25. Furthermore, we can apply Lemma 26, which shows that $\forall F_1, F_2, F_1', F_2', u_1, u_2, u.\ \kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq u_1 \wedge \theta_2 \leq u_2 \models \kappa_1' \wedge \kappa_2' \wedge \theta_1' \leq u_1' \wedge \theta_2' \leq u_2'$ holds. We apply once again Lemma 26 with the implication $\forall F_1, F_2, F_1', F_2', u_1, u_2, u.\ u_1 = u_2 \rightarrow u_2' \wedge u_2' \leq t \wedge t \leq u \models u_1 = u_2 \rightarrow u_2' \wedge u_2' \leq t \wedge t \leq u$, which is an instance of Triv. Using Lemma 24, this gives us the next step in our main proof, namely:

$$
\begin{array}{ll}
& \forall F_1, F_2, F_1', F_2', u. \\
& \quad \kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq u_1 \wedge u_1 = u_2 \rightarrow u_2' \wedge \theta_2 \leq u_2 \wedge u_2' \leq t \wedge t \leq v \\
\models & \quad \kappa_1' \wedge \kappa_2' \wedge \theta_1' \leq u_1 \wedge u_1 = u_2 \rightarrow u_2' \wedge \theta_2' \leq u_2 \wedge u_2' \leq t \wedge t \leq v \\
\models & \quad \kappa_1' \wedge \kappa_2' \wedge \theta_1' \leq u_1 \wedge u_1 = \theta_2' \rightarrow t \wedge t \leq v & \text{(MIntro)} \\
\models & \quad \kappa_1' \wedge \kappa_2' \wedge \theta_1' \leq \theta_2' \rightarrow t \wedge t \leq v & \text{(MTrans)}
\end{array}
$$

By transitivity with (2), this shows the desired property.

case $\tau = \texttt{let}\ t_0\ \texttt{be}\ \tau_0\ \texttt{in}\ \tau_0'$ **with** $t_0 \in FV(\tau_0')$
We can assume w.l.o.g. that t_0 is different from t (**1**) and not free in τ' (**2**) nor in τ (**3**). Therefore,

$$
\begin{array}{lll}
& (\texttt{let}\ t_0\ \texttt{be}\ \tau_0\ \texttt{in}\ \tau_0')[t \leftarrow \tau'] & \\
= & \texttt{let}\ t_0\ \texttt{be}\ \tau_0[t \leftarrow \tau']\ \texttt{in}\ \tau_0'[t \leftarrow \tau'] & \text{(Definition 3)} \\
\equiv & \tau_0'[t \leftarrow \tau'][t_0 \leftarrow \tau_0[t \leftarrow \tau']] & \text{(Definition 28)} \\
= & \tau_0'[t_0 \leftarrow \tau_0][t \leftarrow \tau'] & \text{((1) and (2))} \\
\leq & \tau_0'[t_0 \leftarrow \tau_0][t \leftarrow \tau] & \text{(Induction hypothesis)} \\
= & \tau_0'[t \leftarrow \tau'][t_0 \leftarrow \tau_0[t \leftarrow \tau]] & \text{((1) and (3))} \\
\equiv & \texttt{let}\ t_0\ \texttt{be}\ \tau_0[t \leftarrow \tau]\ \texttt{in}\ \tau_0'[t \leftarrow \tau] & \text{(Definition 28)} \\
= & (\texttt{let}\ t_0\ \texttt{be}\ \tau_0\ \texttt{in}\ \tau_0')[t \leftarrow \tau] & \text{(Definition 3)}
\end{array}
$$

Note that we can apply the induction hypothesis on $\tau_0'[t_0 \leftarrow \tau_0]$ since by Definition 19 it has a smaller size than τ.

case $\tau = \texttt{let}\ t_0\ \texttt{be}\ \tau_0\ \texttt{in}\ \tau_0'$ **with** $t_0 \notin FV(\tau_0')$
We can assume w.l.o.g. that t_0 is different from t (**1**) and not free in τ_1 (**2**) nor in τ_2 (**3**). We suppose that $\tau_1 \leq \tau_2$, and we must prove that $(\texttt{let}\ t_0\ \texttt{be}\ \tau_0\ \texttt{in}\ \tau_0')[t \leftarrow \tau_1] \leq (\texttt{let}\ t_0\ \texttt{be}\ \tau_0\ \texttt{in}\ \tau_0')[t \leftarrow \tau_2]$. By Definition 3 with (1), (2) and (3), for $i = 1, 2$, $\tau[t \leftarrow \tau_i] = \texttt{let}\ t_0\ \texttt{be}\ \tau_0[t \leftarrow \tau_i]\ \texttt{in}\ \tau_0'[t \leftarrow \tau_i]$. Let, for $i = 1, 2$, $\forall \vartheta_i.\ \kappa_i \Rightarrow \theta_i$ be $\langle \tau_0[t \leftarrow \tau_i] \rangle$ and $\forall \vartheta_i'.\ \kappa_i' \Rightarrow \theta_i'$ be $\langle \tau_0'[t \leftarrow \tau_i] \rangle$. We can assume w.l.o.g. that the ϑ_i and ϑ_i' are all pair-wise disjoint (**4**). Let ϑ be $FV(\tau_0[t \leftarrow \tau_1]) \cup FV(\tau_0[t \leftarrow \tau_2])$ and ϑ' be $FV(\tau_0'[t \leftarrow \tau_1]) \cup FV(\tau_0'[t \leftarrow \tau_2])$. Then by Definition 28, $\langle \texttt{let}\ t_0\ \texttt{be}\ \tau_0[t \leftarrow \tau_i]\ \texttt{in}\ \tau_0'[t \leftarrow \tau_i] \rangle = \forall \vartheta_i \vartheta_i'.\ \kappa_i \wedge \kappa_i' \Rightarrow \theta_i'$. By the induction hypothesis $\tau_0[t \leftarrow \tau_1] \leq \tau_0[t \leftarrow \tau_2]$, $\forall u \vartheta.\ \kappa_2 \wedge \theta_2 \leq u \models \kappa_1 \wedge \theta_1 \leq u$. By Lemma 24, $\forall \vartheta.\ \kappa_2 \wedge \theta_2 \leq u \models \kappa_1 \wedge \theta_1 \leq u$. So by VarIntro and Triv, $\forall \vartheta.\ \kappa_2 \models \kappa_1$. Furthermore, by the induction hypothesis $\tau_0'[t \leftarrow \tau_1] \leq \tau_0'[t \leftarrow \tau_2]$, $\forall u' \vartheta'.\ \kappa_2' \wedge \theta_2' \leq u' \models \kappa_1' \wedge \theta_1' \leq u'$. For every variable in ϑ' that does not belong to ϑ, it does not appear in κ_1 since ϑ includes the free variables present in κ_1, and the bound variables are different by (4). Furthermore, the same reasoning is valid for variables in ϑ that do not belong to ϑ', and u' is fresh. So, by Lemma 25, $\forall u'(\vartheta \cup \vartheta').\ \kappa_2 \models \kappa_1$ and $\forall u'(\vartheta \cup \vartheta').\ \kappa_2' \wedge \theta_2' \leq u' \models \kappa_1' \wedge \theta_1' \leq u'$. Therefore, by Lemma 26 (Conjunction), $\forall u'(\vartheta \cup \vartheta').\ \kappa_2 \wedge \kappa_2' \wedge \theta_2' \leq u' \models \kappa_1 \wedge \kappa_1' \wedge \theta_1' \leq u'$. That is, $(\texttt{let}\ t_0\ \texttt{be}\ \tau_0\ \texttt{in}\ \tau_0')[t \leftarrow \tau_1] \leq (\texttt{let}\ t_0\ \texttt{be}\ \tau_0\ \texttt{in}\ \tau_0')[t \leftarrow \tau_2]$.

- Let us prove property **iii** of Definition 4 (Reduction): $(\lambda t.\tau)\ \tau' \geq \tau[t \leftarrow \tau']$. Let $\forall \vartheta.\ \kappa \Rightarrow \theta$ be $\langle \tau \rangle$ and

$\forall \vartheta'.\, \kappa' \Rightarrow \theta'$ be $\langle \tau' \rangle$. If t appears in τ, then $\langle \tau\,[t \leftarrow \tau'] \rangle$ is $\forall \vartheta, \vartheta'.\, \kappa\,[t \leftarrow \theta'] \wedge \kappa' \Rightarrow \theta\,[t \leftarrow \theta']$, otherwise $\langle \tau\,[t \leftarrow \tau'] \rangle$ is $\forall \vartheta.\, \kappa \Rightarrow \theta$.

$$
\begin{array}{lll}
 & \langle (\lambda t.\tau)\ \tau' \rangle & \\
= & \forall \vartheta, t, \vartheta', v.\, \kappa \wedge \kappa' \wedge\ t \rightarrow \theta \leq \theta' \rightarrow v \Rightarrow v & \text{(Definition)} \\
\equiv & \forall \vartheta, t, \vartheta', v.\, \kappa \wedge \kappa' \wedge \theta' \leq t \wedge \theta \leq v \Rightarrow v & (\text{MELIM for } \rightarrow) \\
\equiv & \forall \vartheta, t, \vartheta'.\, \kappa \wedge \kappa' \wedge \theta' \leq t \Rightarrow \theta &
\end{array}
$$

By Lemma 31 (TRANSLATION), $\forall \vartheta, t, \vartheta'.\, \kappa \wedge \kappa' \wedge \theta' \leq t \Rightarrow \theta$ is greater or equal to $\forall \vartheta, \vartheta'.\, \kappa\,[t \leftarrow \theta'] \wedge \kappa' \Rightarrow \theta\,[t \leftarrow \theta']$.

If t appears in τ, this type is equal to $\langle \tau\,[t \leftarrow \tau'] \rangle$. Otherwise, t does not appear in κ nor θ, so this type is equal to $\forall \vartheta, \vartheta'.\, \kappa \wedge \kappa' \Rightarrow \theta$, which is by TRIV greater or equal to $\forall \vartheta.\, \kappa \Rightarrow \theta$, which is $\langle \tau\,[t \leftarrow \tau'] \rangle$. So the inequality holds in both cases.

- Finally, property **iv** of Definition 4 (Let) is straightforward. If t belongs to $FV(\tau_2)$, then by Definition 28 and Definition 29, `let` t `be` τ_1 `in` $\tau_2 \equiv \tau_2\,[t \leftarrow \tau_1]$. Otherwise, let $\forall \vartheta_1.\, \kappa_1 \Rightarrow \theta_1$ be $\langle \tau_1 \rangle$ and $\forall \vartheta_2.\, \kappa_2 \Rightarrow \theta_2$ be $\langle \tau_2 \rangle$. Then $\langle \tau_2\,[t \leftarrow t_1] \rangle = \langle \tau_2 \rangle = \forall \vartheta_2.\, \kappa_2 \Rightarrow \theta_2$. Furthermore, $\forall t.\ \kappa_1 \wedge \kappa_2 \wedge \theta_2 \leq t \models \kappa_2 \wedge \theta_2 \leq t$ is a direct instance of TRIV, so `let` t `be` τ_1 `in` $\tau_2 \geq \tau_2\,[t \leftarrow \tau_1]$ holds by Definition 29.

■

In practice, the definition in Definition 28 of the translation of `let` t_1 `be` τ_1 `in` τ_2 where t_1 is free in τ_2 as the translation of $\tau_2\,[t_1 \leftarrow \tau_1]$ is problematic, because it implies that τ_1 must be translated multiple times, for each occurence of t_1 in τ_2. We now show that this can be avoided.

Lemma 32 (Efficient translation of let types) *Let t_1 be a type variable, and τ_1 and τ_2 two syntactic types of $S(MLS)$. Then the* $\mathrm{ML}_{\leq}$ *types $\langle \tau_2\,[t_1 \leftarrow \tau_1] \rangle$ and $\langle \tau_2\,[t_1 \leftarrow \langle \tau_1 \rangle] \rangle$ are equivalent.*

Consequently, using $\langle \tau_2\,[t_1 \leftarrow \langle \tau_1 \rangle] \rangle$ in the translation leads to an identical type algebra.

Proof of lemma 32 (Efficient translation of let types)

By Definition 28, $\langle \langle \tau_1 \rangle \rangle = \langle \tau_1 \rangle$. So by Definition 29, $\langle \tau_1 \rangle \equiv \tau_1$. Therefore, by Theorem 30 (ML-SUB) and property **ii** of Definition 4 (Covariance), $\langle \tau_2\,[t_1 \leftarrow \tau_1] \rangle \equiv \langle \tau_2\,[t_1 \leftarrow \langle \tau_1 \rangle] \rangle$ holds. ■

Part II

Object-orientation

Chapter 3

Classes

In this chapter we present an extension language for the declaration of classes. Our classes are tagged extensible records of fields. They do not contain methods, since methods can be declared independently, as formalized in Chapter 4. This removes the need to distinguish subtyping from subclassing, even in presence of covariant specialization of methods [11]. Therefore, we only describe how class declarations declare object creation and field access operators. Most of the original aspect of the work lies in the typing of multi-methods described in Section 8.6. However, it should be possible to fit other approaches to object-orientation in our framework.

We tackle the general case of multiple inheritance, which includes single inheritance as a subcase. Classes can be parameterized by types. Therefore, each class does not define a type, but a type constructor with the same name. Classes come with a signature that describes its number of type parameters and the variance of each of them: respectively $\oplus$, $\ominus$ and $\otimes$ for co-, contra- and non-variant type parameters. A class can only inherit from classes with the same variance. We shall thus often leave the variance annotations implicit. For instance, a class $\texttt{list}_{\oplus}[t]$ defines a type constructor with one covariant type parameter.

Class declarations list the set of fields of the declared class. We use integers to label fields for the ease of the presentation, but using names instead would raise no problem. The concrete syntax for the declaration of a class C of variance V is:

$$\textbf{class } C_V[\overline{t}] \textbf{ extends } C_1, ..., C_m \ \{1 : F_1(\overline{t}), ..., p : F_p(\overline{t})\}$$

where each F_i is a function mapping the type parameters $\overline{t}$ to monotypes. If a value has type $C_V[\Theta]$ (and therefore is an instance of class C_V by Requirement 38), then $F_i(\Theta)$ is the type of its i^{th} field.

For instance, lists can be defined by an abstract class `list` and two subclasses, the empty list `nil` and the non-empty list `cons`:

class $\texttt{list}_{\oplus}[t]$ { }
class $\texttt{nil}_{\oplus}[t]$ **extends** `list` { }
class $\texttt{cons}_{\oplus}[t]$ **extends** `list` { $1 : t, 2 : \texttt{list}[t]$ }

Field types are monomorphic: they can only depend on the type parameters of the class. This restriction arises from the fact that we want field access operators to have a rank-1 polymorphic type, that is, with all quantifiers at the front of the type. Otherwise, if we could define **class** $C[t]$ { $1{:}\forall u.\ u \to t$ } , then the type of the field access operator would be $\forall t.\ C[t] \to (\forall u.\ u \to t)$, of rank 2. Note that, unlike in the encoding of classes as extensible records of fields and methods, this restriction does not prevent methods from being polymorphic, since methods are not class members (see Section 4).

A class declaration can be interpreted as a partial definition for an $\text{ML}_{\leq}$ type structure, as introduced in Section 2.2. It introduces a new type constructor for the class, as well as subtyping between this new constructor and the type constructors corresponding to its super-classes.

Definition 33 (Partial type structure) *The partial type structure induced by a class declaration, noted* $TS(\textbf{\textit{class}}\ C_V[\overline{t}]\ \textbf{\textit{extends}}\ C_1, ..., C_m\ ...)$*, is the couple* $(\{C_V\}, \{C_V \leq C_1, ..., C_V \leq C_m\})$.

In a program, which typically includes several class declarations, the type structure is the union of all partial class structures. We formalize the notion of program in Chapter 4.

Definition 34 (Type structure) *Given a set of class declarations $\mathcal{C}_1, ..., \mathcal{C}_n$, let (c_i, o_i) be $TS(\mathcal{C}_i)$. The induced type structure $TS(\mathcal{C}_1, ..., \mathcal{C}_n)$ is $(\bigcup_{i=1}^{n} c_i, (\bigcup_{i=1}^{n} o_i)^*)$, where * is the reflective and transitive closure of relations.*

For instance, the type structure defined by the list hierarchy is

$$(\{\texttt{list}, \texttt{nil}, \texttt{cons}\}, \{\texttt{nil} \leq \texttt{list}, \texttt{cons} \leq \texttt{list}\})$$

For a class declaration to be valid, its field types must respect the variance of the class. For instance, since $\texttt{cons}_{\oplus}[t]$ has variance $\oplus$, it can have a field with type t, but not with a type of the form $t \to \theta$ for some monotype θ, since t appears contra-variantly in that type. Without this restriction, we could define a class **class** $\texttt{bad}_{\oplus}[t]$ $\{\, 1 : t \to \texttt{bool} \,\}$. By covariance, a value of type $\texttt{bad}[\texttt{int}]$ could be used in a context a value of type $\texttt{bad}[\texttt{float}]$ is expected. However, this context could fetch the field, and expect it to be of type $\texttt{float} \to \texttt{bool}$, while the value might only be of type $\texttt{int} \to \texttt{bool}$, which is not a subtype of $\texttt{float} \to \texttt{bool}$ by the contra-variance of $\to$. Similarly, if we formalized mutability, a class with a covariant type parameter could not have a mutable field of that type. The general condition is formalized in the following requirement:

Requirement 35 (Fields) *Let C be a class declared by* ***class*** $C_V[\overline{t}]$ ***extends*** $C_1, ..., C_m$ $\{1 : F_1(\overline{t}), ..., p : F_p(\overline{t})\}$. *Then, the following* $\mathrm{ML}_{\leq}$ *implication must hold for all i in $1..p$:*

$$\forall \overline{t}\,\overline{t}'.\ \overline{t} \leq_V \overline{t}' \models F_i(\overline{t}) \leq F_i(\overline{t}')$$

3.1 Object instantiation

We shall now see how class declarations implicitly add constants to the core language. Each class declaration introduces a **new** data constructor whose arity is equal to the number of fields of instances of C, including the fields defined in the super-classes of C. We can therefore represent an object as the application of a **new** data constructor to a value for each field in a given, arbitrary order.

We use ordered lists to represent the super-classes and the fields of classes. We shall write $[x_1, x_2, ..., x_n]$ for the list of n elements, in order x_1, then x_2, until x_n. We shall write $+$ for list concatenation. That is, $[x_1, ..., x_n] + [y_1, ..., y_m]$ is $[x_1, ..., x_n, y_1, ..., y_m]$. Given two lists L_1 and L_2, we shall write $L_1 \setminus L_2$ for the list that has all elements of L_1 that do not appear in L_2. Finally, we write $\bullet$ for the duplicate-free concatenation of lists. That is, $L_1 \bullet L_2$ is defined as $L_1 + (L_2 \setminus L_1)$. It follows that $\bullet$ is associative: $(L_1 \bullet L_2) \bullet L_3$ is the same list as $L_1 \bullet (L_2 \bullet L_3)$, and can therefore be simply written as $L_1 \bullet L_2 \bullet L_3$.

Definition 36 (Super-class and field lists) *Let C be a class declared by* ***class*** $C_V[\overline{t}]$ ***extends*** $C_1, ..., C_m$ $\{1 : F_1(\overline{t}), ..., p : F_p(\overline{t})\}$.

$$sc(C) = \mathrm{sc}(C_1) \bullet \mathrm{sc}(C_2) \bullet ... \bullet \mathrm{sc}(C_m) \bullet [C]$$

$$\mathrm{Fields}(C) = [F_1, ..., F_p]$$

$$\mathrm{AllFields}(C) = \mathrm{Fields}(D_1) + ... + \mathrm{Fields}(D_n) \ \textit{where}\ [D_1, ..., D_n] = sc(C)$$

The use of $\bullet$ avoids listing a class twice if it is inherited through two different paths.

We will need to compute the first index of the fields declared by a class C in the ordered list of all fields of a subclass C' of C. We will compute this index as $\mathrm{shift}(sc(C'), C)$, where $\mathrm{shift}(\overline{C}, C)$ is defined inductively by case on the first element of the first parameter:

$$\begin{array}{lcll} \mathrm{shift}([C_1, C_2, ..., C_n], C) & = & 0 & (C_1 = C) \\ \mathrm{shift}([C_1, C_2, ..., C_n], C) & = & \left|\mathrm{Fields}\,(C_1)\right| + \mathrm{shift}([C_2, ..., C_n], C) & (C_1 \neq C) \end{array}$$

The shift meta-operator verifies the following lemma:

Lemma 37 (Shift) *Let C be an element of $sc(C')$, $[F_1, ..., F_p]$ be* Fields (C)*, and i be an integer in $1..p$. Then*

$$\text{AllFields}(C')_{\text{shift}(sc(C'),C)+i} = F_i$$

The data constructor **new** C is used to create object values by application to field expressions of the expected type. Let $[F'_1, ..., F'_n]$ be AllFields(C), then

$$\textsf{constant-type}(\textbf{new}\ C) = \forall \overline{t}.\ F'_1(\overline{t}) \to ... \to F'_n(\overline{t}) \to C[\overline{t}]$$

For instance, the instantiation operators for the list hierarchy have the following types:

$$\textsf{constant-type}(\textbf{new}\ \texttt{nil}) = \forall t.\, \texttt{nil}[t]$$
$$\textsf{constant-type}(\textbf{new}\ \texttt{cons}) = \forall t.\ t \to \texttt{list}[t] \to \texttt{cons}[t]$$

A class C can also be declared *abstract* in order to assert that no object is built on this class. In this case the operator **new** C is not introduced.

Below, we require that **new** C is the only data constructor that may create an object with a type constructed on C, as formalized by the following requirement.

Requirement 38 (Class type) *Let C be a class of variance V. Let $\overline{t}$ be a list of fresh type variables of length $arity(V)$. Let v be a value of type $\forall \vartheta_v.\, \kappa_v \Rightarrow \theta_v$. Then if $true \models \kappa_v \wedge \theta_v \leq C[\overline{t}]$ holds, v is of the form **new** $C'\ v_1\ ...\ v_n$ with $n = |AllFields(C')|$ and $C \in sc(C')$.*

The constraint implication $true \models \kappa_v \wedge \theta_v \leq C[\overline{t}]$ holds when there exists monotypes $\overline{\theta}$ and an instance of the type of v which is a subtype of $C[\overline{\theta}]$. In particular, it follows that the field access operators of class C, which we define in the next section, can be applied to v. Requirement 38 will allow us to state in Lemma 39 that this is only possible for object instances, created by **new**.

This requirement does restrict the expressiveness of objects but simply rules out faulty data constructors that would have an object type without being an object of class C. In turn, this would prevent field access operators from being total on their domain. Without Requirement 38, one could for instance define a data constructor **null** with type $\forall t.\, t$. The value **null** would therefore be usable anywhere any object is expected, but there would be no possible meaningful reduction for the application of field access operator to **null**.

3.2 Field access in classes

Object values are built by instantiation of classes. Symmetrically, it is necessary to be able to deconstruct these objects by accessing the values of their fields. To this end, each class declaration **class** $C_V[\overline{t}]$ **extends** $C_1, ..., C_m$ $(1 : F_1(\overline{t}), ..., p : F_p(\overline{t}))$ defines p field access operators for each field declared in C. The operator that accesses the i^{th} field declared in class C is written $C.i$. It has the following type:

$$\textsf{constant-type}(C.i) = \forall \overline{t}.\ C[\overline{t}] \to F_i(\overline{t})$$

For instance, the field access operators for the example list hierarchy have the following types:

$$\textsf{constant-type}(\texttt{cons}.1) = \forall t.\ \texttt{list}[t] \to t$$
$$\textsf{constant-type}(\texttt{cons}.2) = \forall t.\ \texttt{list}[t] \to \texttt{list}[t]$$

Lemma 39 (Field access) *Let C be a class, $C.i$ be a field access operator of class C, and v be a value such that $C.i\ v$ is well-typed. Then v is of the form **new** $C'\ v_1\ ...\ v_n$ with $n = |AllFields(C')|$ and $C \in sc(C')$.*

Proof of lemma 39

Let $\forall\vartheta.\,\kappa \Rightarrow \theta$ be $\mathsf{type}(v)$, let V be the signature of class C and $\overline{t}$ be a list of type variable of length $\mathsf{arity}(V)$. By Definition 27 and the hypothesis that $C.i\ v$ is well-typed, the type $\forall\overline{t},\vartheta.\,\kappa \wedge \theta \leq C[\overline{t}] \Rightarrow F_i(\overline{t})$ is well formed. That is, the constraint implication $\mathit{true} \models \kappa_v \wedge \theta_v \leq C'[\overline{t}]$ holds. Therefore, by Requirement 38 (CLASS TYPE), v is of the form $\mathbf{new}\ C'\ v_1\ ...\ v_n$, where $n = |AllFields(C')|$ and $C \in sc(C')$. ■

In order to define the reduction rule for the field access operator $C.i$, we use the shift meta-operator defined in the previous section.

$$C.i\ (\mathbf{new}\ C'\ e_1\ ...\ e_n) \longrightarrow e_{\mathrm{shift}(sc(C'),C)+i}$$

Note that, by Lemma 39, $C.i\ (\mathbf{new}\ C'\ e_1\ ...\ e_n)$ can only be well-typed if C' is a subclass of C. This ensures that $\mathrm{shift}(sc(C'), C)$ is well-defined.

Next, we show that the field access operator is soundly defined.

Theorem 40 (Field access soundness) *Given a type structure and a set of operators such that Requirement 35 and Requirement 38 are fulfilled, let C.i be a field access operator. Then the two following properties hold:*

- *if $C.i\ v$ is well-typed, then there exists a value v' such that $C.i\ v \longrightarrow v'$. Furthermore, v is of the form $\boldsymbol{new}\ C'\ v_1\ ...\ v_n$ and $v' = v_{\mathrm{shift}(sc(C'),C)+i}$;*
- *if*

  $$\mathsf{type}(C.i\ (\boldsymbol{new}\ C'\ v_1\ ...\ v_n)) = \tau$$

  then there exists τ' such that

  $$\mathsf{type}(v_{shift(sc(C'),C)+i}) = \tau' \text{ and } \tau' \leq \tau$$

Proof of theorem 40 (Field access soundness)

- By hypothesis, $C.i\ v$ is well-typed. Therefore, by Lemma 39 (FIELD ACCESS), v is of the form $\mathbf{new}\ C'\ v_1\ ...\ v_n$, where $n = |AllFields(C')|$ and $C \in sc(C')$. So by the reduction rule for field access, the expression $C.i\ v$ reduces to $v_{shift(sc(C'),C)+i}$.
- We omit here the annotation by variance V. Let $\forall\vartheta_{v_j}.\,\kappa_{v_j} \Rightarrow \theta_{v_j}$ be $\mathsf{type}(v_j)$ for j in $1..n$ and $\forall\overline{t}.\,F_1(\overline{t}) \rightarrow ... \rightarrow F_n(\overline{t}) \rightarrow C'[\overline{t}]$ be $\mathsf{type}(\mathbf{new}\ C')$. Therefore, by APP and Definition 28:

  $$\mathsf{type}(\mathbf{new}\ C'\ v_1\ ...\ v_n) = \forall\overline{t}\vartheta_{v_j}.\,\kappa_{v_j} \wedge \theta_{v_j} \leq F_j(\overline{t}) \Rightarrow C'[\overline{t}]$$

  with j ranging from 1 to n. Let $\overline{t}$ and $\overline{t}'$ be distinct fresh monotype variable lists of length $\mathsf{arity}(V)$. Then by Figure 1.4 and Definition 28, $\mathsf{type}(C.i\ (\mathbf{new}\ C'\ v_1\ ...\ v_n)) = \forall\overline{t}\,\overline{t}'\vartheta_{v_j}.\,\kappa_{v_j} \wedge \theta_{v_j} \leq F_j(\overline{t}) \wedge C'[\overline{t}] \leq C[\overline{t}'] \Rightarrow F'_i(\overline{t}')$ (**5**).

  Let s be the integer $\mathrm{shift}(sc(C'), C) + i$. We claim that the type $\forall\vartheta_{e_s}.\,\kappa_s \Rightarrow \theta_s$ of v_s is a subtype of (5). By Definition 27 this amounts to:

  $\forall t\,\overline{t}\,\overline{t}'.$

	$\kappa_{v_j} \wedge \theta_{v_j} \leq F_j(\overline{t})$	(j ranging from 1 to n) $\wedge\ C'[\overline{t}] \leq C[\overline{t}'] \wedge F'_i(\overline{t}') \leq t$
$\models$	$\kappa_{v_s} \wedge \theta_{e_s} \leq F_s(\overline{t}) \wedge C'[\overline{t}] \leq C[\overline{t}'] \wedge F'_i(\overline{t}') \leq t$	(TRIV, keeping only $j = s$)
$\models$	$\kappa_{v_s} \wedge \theta_{e_s} \leq F'_i(\overline{t}) \wedge C'[\overline{t}] \leq C[\overline{t}'] \wedge F'_i(\overline{t}') \leq t$	(Lemma 37: $F_s = F'_i$)
$\models$	$\kappa_{v_s} \wedge \theta_{e_s} \leq F'_i(\overline{t}) \wedge \overline{t} \leq \overline{t}' \wedge F'_i(\overline{t}') \leq t$	(MELIM)
$\models$	$\kappa_{v_s} \wedge \theta_{e_s} \leq F'_i(\overline{t}) \leq F'_i(\overline{t}') \leq t$	(Requirement 35, Lemma 26)
$\models$	$\kappa_{v_s} \wedge \theta_{e_s} \leq t$	(MTRANS)

  Restriction to $\forall t$ by Lemma 24 concludes.

■

Chapter 4

Generic functions

Multi-methods are less popular than mono-methods used in single-dispatch languages (C++ [41], Java [25], OCaml [40], ...). Nevertheless, they have been studied and used in several programming languages (CLOS [4, 23], Dylan [21], Cecil [13, 14], ...). However, their type-checking in the presence of polymorphic types, in a decidable and modular way that preserves type inference on ML expressions is still an issue.

In this section, we present a formalization of generic functions in our framework. Generic functions are operations that can select different behavior depending on the type of their arguments. They capture the essential properties of multi-methods, while being more general, by abstracting over the type algebra used to typecheck them. Thus they are a good framework to formalize type-checking and modularity. In this section we only present the type-checking aspect. We handle modularity in Chapter 8.

4.1 Example

For this example we assume a type algebra that has bounded polymorphic types. As this example is only meant to support intuition, we deliberately remain informal. Base types include `int`, `float`, and `num` such that $\texttt{int} \leq \texttt{float} \leq \texttt{num}$. Type `num` is abstract in the sense that it has no direct instance. Conversely, `int` and `float` are concrete.

In order to motivate the introduction of generic functions, assume given two monomorphic functions `opp_float` and `opp_int` that compute the unary minus function on floats and integers. We would like to define a function that computes the opposite of any number. The function `opp_float` could be used, because an integer can be considered as a float by subsumption. However, in that case the result has type `float`, while we know statically that it is `int`. Bounded polymorphism allows to give it the more precise type $\forall t \leq \texttt{num}.\, t \to t$. However, `opp_float` cannot be given this type since it always returns a float. Furthermore, the only function with this type that can be written in Core-ML is the identity on `num`.

What we need is a *generic function* construct that pattern matches on the runtime-type of its arguments. In our example:

$$\textbf{generic}\ \texttt{opp}\ :\ \forall t \leq \texttt{num}.\, t \to t\ =\ \begin{array}{l} \texttt{float} \Rightarrow \texttt{opp\_float} \\ \texttt{int} \Rightarrow \texttt{opp\_int} \end{array}$$

The intent is that one of the implementations of the generic function will be selected and applied, at runtime, depending on the runtime type of the argument. Each implementation branch consists of a list of *patterns* to declare the types it handles and of an expression that implements the generic function in that case. There must be one pattern for each argument of the generic function. In the example above, the lists of patterns of each implementation have a single element, respectively `float` and `int`, since the generic function has only one parameter. No implementation is needed for `num` since it is an abstract type.

4.2 Syntax

The syntax for programs with generic functions is:

$$\begin{array}{lcll} \textit{Generic function} & \mathcal{G} & ::= & \textbf{generic g} \ : \ \tau \ = \mathcal{I}_1, ... \mathcal{I}_p \\ \textit{Implementation} & \mathcal{I} & ::= & \overline{\pi} \Rightarrow e \\ \textit{Class} & \mathcal{C} & ::= & \textbf{class } C \textbf{ extends } \overline{C} \ \{ \ ... \ \} \\ \textit{Declaration} & \mathcal{D} & ::= & \mathcal{G} \mid \mathcal{C} \\ \textit{Program} & \mathcal{P} & ::= & \textbf{let rec } \overline{\mathcal{D}} \textbf{ in } e \end{array}$$

where π belongs to a language of *patterns*. A pattern represents the set of types that belong to that pattern, as defined by the binary predicate $\tau \in \pi$. Both the set of patterns and this predicate are left abstract at this stage. A semantics for programs with generic functions is first defined in Section 4.3; then in Section 4.4 we state type-checking rules and express the property that these abstract components must verify so as to ensure type soundness. We provide a concrete instance of generic functions in Section 8.6.

4.3 Semantics

We need to define the reduction rules for generic function operators. To this end, we formalize what it means for a pattern to match a tuple of types in the following definition:

Definition 41 (Pattern matching) *A pattern* $\overline{\pi}$ matches $(\tau_1, ..., \tau_n)$ *if for all* i *in* $1..n$, $\tau_i \in \pi_i$ *holds.*

We also define an ordering on patterns. A pattern is smaller than another one if it matches more types.

Definition 42 (Pattern ordering) *A pattern* $\overline{\pi}_1$ *is smaller or equal to a pattern* $\overline{\pi}_2$*, written* $\overline{\pi}_1 \leq \overline{\pi}_2$*, if for all types* $(\tau_1, ..., \tau_n)$ *such that* $\overline{\pi}_2$ *matches* $(\tau_1, ..., \tau_n)$*,* $\overline{\pi}_1$ *matches* $(\tau_1, ..., \tau_n)$*.*

A pattern $\overline{\pi}_1$ *is smaller than a pattern* $\overline{\pi}_2$*, written* $\overline{\pi}_1 < \overline{\pi}_2$*, if* $\overline{\pi}_1 \leq \overline{\pi}_2$ *holds and* $\overline{\pi}_2 \leq \overline{\pi}_1$ *does not hold.*

We order implementations based on the order of their patterns.

Definition 43 (Implementation ordering) *An implementation* $\overline{\pi}_1 \Rightarrow e_1$ *is smaller than (respectively smaller or equal to) a pattern* $\overline{\pi}_2 \Rightarrow e_2$ *if* $\overline{\pi}_1$ *is smaller than (respectively smaller or equal to)* $\overline{\pi}_2$*.*

Intuitively, smaller implementations are more precise, because they are applicable to less types.

The declaration **generic g** $: \ \tau \ = \overline{\pi}_1 \Rightarrow e_1, ..., \overline{\pi}_p \Rightarrow e_p$ introduces an operator g in the language, with $\mathsf{constant\text{-}type}(g) = \tau$. The reduction rules for this operator are defined by:

$$\frac{\overline{\pi}_i \text{ matches } (\mathsf{type}(v_1), ..., \mathsf{type}(v_n))}{(v_1, ..., v_n, e_i \ v_1 \ ... \ v_n) \in R(g)}$$

Hence, we have defined an operator that fits the general framework of Section 1. In particular, the derived rule obtained by combination with the reduction rule for operators Op is then:

$$\frac{\overline{\pi}_i \text{ matches } (\mathsf{type}(v_1), ..., \mathsf{type}(v_n))}{g \ v_1 \ ... \ v_n \longrightarrow e_i \ v_1 \ ... \ v_n}$$

Note that the semantics is possibly non-deterministic since several implementations might match a given tuple of arguments. For instance, consider the generic function

$$\textbf{generic } \texttt{add} \ : \ \forall t \leq \texttt{num}. \, (t, t) \rightarrow t \ = \begin{array}{l} \texttt{float}, \texttt{float} \Rightarrow ... \\ \texttt{float}, \texttt{int} \Rightarrow ... \\ \texttt{int}, \texttt{float} \Rightarrow ... \end{array}$$

The addition of two integers could be handled by the second and the third implementation, and, by symmetry, there is no reason to choose one rather than the other. In practice, it is often desirable to enforce a

deterministic behavior. This can be achieved by requiring that generic functions are *not ambiguous*. This requires that for each possible call there exists a *most specific* matching implementation. Reduction then happens unambiguously by selection of the most specific implementation. This aspect is well-known from the study of multi-methods [28, 2, 13, 6], and is orthogonal to type-soundness.

The semantics is typed in the sense that the types of the values are used in the definition of the reduction rule. However, this does not mean that expressions have to carry a type at run-time. In opposition to dynamic values (values carrying runtime types on which type matching can be performed, as in [30, 1]), types involved here are not static types carried at runtime, but runtime types. In the general case, reduction can therefore require the computation of the runtime types, which can be costly in term of performance. However, depending on the actual language of patterns that is used, reduction might also be performed without actually computing the types of the arguments. For instance, in an object-oriented language, object values may carry the class of which they are an instance. Therefore, if one can decide whether a type matches a pattern solely considering the class on which the type is built, reduction can be implemented without any runtime type computation. This is indeed the case for multi-methods, as can be seen in Section 8.6.

Note that since generic functions are operators, they are made available globally to the whole program by rule CST. They can therefore appear in generic function bodies, so that generic functions are globally mutually recursive. In particular, this allows for polymorphic recursion. This does not lead to undecidability of typechecking since types of generic functions are declared and not inferred.

4.4 Type-checking

Given a program $\mathcal{P}$ containing the class declarations $\mathcal{C}_1, ..., \mathcal{C}_n$, we place ourselves in the type structure $TS(\mathcal{C}_1, ..., \mathcal{C}_n)$ as defined in Definition 34.

We now present a static type system to detect errors in programs with generic functions. Thanks to our general soundness result, Theorem 12, this reduces to proving that generic functions define operators that satisfy Requirement 9 (CONSTANTS). Informally, the type of a generic function introduces two requirements on its implementations, corresponding to the "subject-reduction" and "progress" parts of the requirement for operators.

Firstly, it can be seen as a specification that every implementation must meet. However, since the patterns restrict the domain of the arguments, this type can be specialized, taking into account the information in the patterns. Thus, we require a notion of *restriction of a type to a list of patterns*, and we demand that each implementation has a subtype of the restriction of the generic type to the implementation's patterns. For instance, the restriction of the type $\forall t \leq \texttt{num}.\, t \to t$ of `opp` to pattern `float` is $\texttt{float} \to \texttt{float}$. A particular case arises when the restricted type is the error type $\mathbb{E}$. Since $\mathbb{E}$ is maximal, the above requirement is trivially satisfied. However, this situation corresponds to an implementation that would never be used (which is why no condition is needed on the type of the implementation to ensure type safety). Therefore, it makes good sense to additionally require that the restricted type is not $\mathbb{E}$. For instance, an implementation of `opp` with pattern `string` would be rejected, since `string` is not a subtype of `num`.

Formally, we assume given a total function $\mathsf{restrict}$. The type $\mathsf{restrict}(\tau, \overline{\pi})$ defines the restriction of a type τ to a list of patterns $\overline{\pi}$. It must satisfy the following requirement:

Requirement 44 (Restriction) *For all types τ, τ_i $(i = 1..n)$ and patterns $\overline{\pi}$,*

$$\frac{\tau_i \in \pi_i \ (i = 1..n)}{\mathsf{restrict}(\tau, \overline{\pi})\ \tau_1\ ...\ \tau_n \leq \tau\ \tau_1\ ...\ \tau_n}$$

As an example, we give possible definitions for $\mathsf{restrict}$ and $\in$ in the monomorphic type algebra of Section 1.1.2. Patterns are in this case simply algebraic types denoted by a. A type belongs to a pattern if it is a subtype of that pattern. That is, $a \in a'$ holds iff $a \prec a'$. The restriction of a functional type to a list of patterns is the functional type whose domain is the list of patterns and whose codomain is the original codomain. That is, $\mathsf{restrict}(a, b_1 ... b_n)$ is $b_1 \to ... \to b_n \to a_0$ when a is of the form $a_1 \to ... \to a_n \to a_0$ and $b_i \prec a_i$ holds for all i from 1 to n; it is $\mathbb{E}$ otherwise. It is easy to check that Requirement 44 (RESTRICTION)

is satisfied by these definitions. The rationale for the definition of restrict is that when a generic function of type $a_1 \to ... \to a_n \to a_0$ is applied to values of types $b_1 ... b_n$, it needs to be implemented by a function of type $b_1 \to ... \to b_n \to a_0$.

We can now define the validity of a generic function declaration, and of an implementation.

Definition 45 (Valid implementation) *An implementation $\overline{\pi} \Rightarrow e$ is* valid for type τ *if*

1. $\mathsf{restrict}(\tau, \overline{\pi}) \neq \mathbb{E}$
2. $\mathsf{type}(e) \leq \mathsf{restrict}(\tau, \overline{\pi})$

Definition 46 (Valid generic function) *The declaration of a generic function* ***generic*** g : τ = $\mathcal{I}_1, ..., \mathcal{I}_p$ *is valid if for all i, the implementation $\mathcal{I}_i$ is valid for type τ.*

Condition (1) in Definition 45 prevents from using patterns that are incompatible with the domain of the generic function and that would therefore never be used. This condition is not mandatory but makes good sense. Conversely, condition (2) is essential to ensure subject-reduction.

Now, we consider the second requirement on the implementations of a generic function, which is necessary to fulfill the Requirement Requirement 9.i, that is the "progress" part. To this end, one can notice that the type of the generic function determines its domain. The implementation branches, taken together, must cover this domain. Back to the introductory example, the generic function opp must possess an implementation for any non-abstract subtype of num. It is indeed not necessary that an implementation exists for types that have no runtime values. We therefore define run-time types and *covered generic functions*:

Definition 47 (Runtime type) *A type τ is a* run-time type *if there exists a value v such that* $\mathsf{type}(v) = \tau$.

Definition 48 (Covered generic) *A generic function* ***generic*** g : $\tau = \overline{\pi}_1 \Rightarrow e_1, ... \overline{\pi}_p \Rightarrow e_p$ *is* covered *if for all run-time types $\tau_1, ..., \tau_n$ such that $\tau\ \tau_1\ ...\ \tau_n \neq \mathbb{E}$, there exists an index i such that $\overline{\pi}_i$ matches $(\tau_1, ..., \tau_n)$.*

Chambers and Leavens present an efficient algorithm for testing coverage (and non-ambiguity, to ensure determinism) in [16], which is applicable in our context. Therefore, we do not address this point.

It is now possible to express a sufficient condition for programs with generic functions to be sound.

Theorem 49 (Generic functions) *The operators defined by valid and covered generic functions satisfy Requirement 9 (*Constants*).*

The first part of Requirement 9 (Constants) is directly implied by Definition 48 of covered generic functions together with the reduction rules. The second part is a consequence of Definition 45 of a valid implementation, using Requirement 44 (Restriction) on the restriction operator.

Proof of theorem 49 (Generic functions)

Since generic functions are operators, we need to prove parts i and ii of Requirement 9 (Constants).

i. By hypothesis, $g\ v_1\ ...\ v_n$ is well-typed, so by Definition 47 each $\mathsf{type}(v_i)$ is a runtime type. So, by Definition 48, there exists an implementation $\pi_1 ... \pi_n \Rightarrow e$ such that, for i from 1 to n, $\mathsf{type}(v_i) \in \pi_i$. Therefore, by definition of reduction rules, $g\ v_1\ ...\ v_n \longrightarrow e\ v_1\ ...\ v_n$

ii. Let I be an implementation $\overline{\pi} \Rightarrow e$ such that $\overline{\pi}$ matches $\mathsf{type}(v_1) ... \mathsf{type}(v_n)$, and τ_g be $\mathsf{constant\text{-}type}(g)$, the declared type of g. We need to prove that $\mathsf{type}(e\ v_1\ ...\ v_n)$ is a subtype of $\mathsf{type}(g\ v_1\ ...\ v_n)$.

$$
\begin{array}{lll}
 & \mathsf{type}(e\ v_1\ ...\ v_n) & \\
= & \mathsf{type}(e)\ \mathsf{type}(v_1)\ ...\ \mathsf{type}(v_n) & (\textsc{App}) \\
\leq & \mathsf{restrict}(\tau_g, \pi)\ \mathsf{type}(v_1)\ ...\ \mathsf{type}(v_n) & (\text{validity of } I \text{ and Covariance (Definition 4.}\mathbf{ii})) \\
\leq & \tau_g\ \mathsf{type}(v_1)\ ...\ \mathsf{type}(v_n) & (\text{Requirement 44 and Covariance (Definition 4.}\mathbf{ii})) \\
= & \mathsf{type}(g\ v_1\ ...\ v_n) & (\textsc{App})
\end{array}
$$

■

A program **let rec** $\overline{\mathcal{G} \mid \mathcal{C}}$ **in** e is well-typed if every generic function $\mathcal{G}$ is valid and covered, and if e is well-typed. The result of the program is the evaluation of e. By Theorem 12 (Soundness) and Theorem 49 (Generic functions), e reduces to a value v such that $\mathsf{type}(v) \leq \mathsf{type}(e)$.

We formalize type-checking for modular programs with generic functions in Chapter 8.

Chapter 5

Super

5.1 Super in class-based languages

Super is a construct that allows to reuse an existing implementation of a method inside another implementation of the method in a sub-class.

In the following example written in Java, a method performing side-effects (and therefore returning **void**), starts with performing the possible side-effects of the existing implementation.

```
class A {
  int x;
  void print(OutputStream s) {
    s.print(x);
  }
}

class B extends A {
  int y;
  void print(OutputStream s) {
    super.print(s);
    s.print(y);
  }
}
```

Here, `super.print` means the implementation of method **print** in the super-class of the current class. In particular, it is not mandatory to use the same name as the current method. It is therefore possible to call another method, implemented in the super-class. That is, in the body of a method `f` in class `B`, it is possible to use `super.g`. This semantics is tightly coupled with single-dispatch, in that it relies on the fact that dispatch is only made on the first argument only.

5.2 Super in multi-method languages

5.2.1 Dylan

The Dylan language features multi-methods. It allows to call a more general method from a method, with the **next-method** keyword. A higher-level description of its semantics is: **next-method** calls the method that would have been called if the current method had not existed. This call can fail at runtime in case of ambiguity, for the same reasons as for method calls. Dylan allows such failures at runtime, which is probably unavoidable given the possibility in Dylan to add method implementations at runtime.

The following program is the translation in the syntax of Dylan of the example of Section 5.1.

```
define class <A> (<object>)
  slot x;
end class <A>;

define class <B> (<A>)
  slot y;
end class <B>;

define generic print (a :: <A>);

define method print (a :: <A>)
  format-out("%d\n", a.x);
end method print;

define method print (b :: <B>)
  next-method();
  format-out("%d\n", b.y);
end method print;
```

Besides the syntactic differences with the code in the previous section, the main point to note is that a call to **next-method** does not specify a method name. One way to interpret this fact is that **next-method** is a high-level construct specifying how an implementation of a method is related to the implementations it overrides, while **super** in the previous section is a lower-lever construct specifying a special way to perform dispatch for a certain method call.

Furthermore, the **next-method** statement does not specify arguments. This is possible since, by default, **next-method** is called with the same arguments as the current method. It is possible in Dylan to pass other arguments. The specification requires that the new arguments lead to the same sequence of applicable methods as the original arguments. Otherwise, the semantics is undefined. The underlying problem is that dispatch has already been performed. In particular, by passing arguments with greater types, they might not be compatible with the called method, even though they are compatible with the generic function. This case is illustrated in the following example:

```
define class <A> (<object>)
  slot x, init-value: 0;
end class <A>;

define class <B> (<A>)
  slot y, init-value: 0;
end class <B>;

define class <C> (<B>)
end class <C>;

define generic print (a :: <A>);

define method print (a :: <A>)
  format-out("%d\n", a.x);
end method print;

define method print (b :: <B>)
  next-method();
  format-out("%d\n", b.y);
end method print;
```

```
define method print (c :: <C>)
  next-method(make(<A>));
end method print;
```

In the implementation of `print` for class `C`, **next-method** is called with a newly built instance of class `A`. However, the method called is the implementation of `print` for class `B`. That method can not handle the instance of class `A` that it receives in this call, which will provoke an error at runtime.

5.2.2 Cecil

Cecil features the **resend** keyword. While serving the same role as **next-method** in Dylan, it is better in two ways. First, it is possible to explicitly resolve ambiguities by directing the call: every argument of the current method specialized for a class `C` can be specialized in the **resend** to an ancestor of class `C`. For instance, if a class `C` has two super-classes `B1` and `B2` that are both subclasses of class `A`, the implementation of a method for class `C` can specify that a **resend** targets the implementation of that method for either `B1` or `B2`. Second, Cecil restricts passing a different argument than the original one to the case where that argument is not specialized. This allows to give a safe formal semantics in every case, unlike in Dylan.

The reference manual of Cecil includes typing rules for **resend**. However, they do not take into account polymorphic types, which are presented later as an extension. Furthermore, they require the explicit typing of every method implementation. In particular, the type of **resend** is the declared return type of the implementation that it calls.

5.3 Formalization

The syntax of method implementations is extended to allow the following notation:

$$\textbf{implementation}\,\mathtt{m}\,\overline{\pi} \Rightarrow \lambda x_1 ... x_n . e(\textbf{super})$$

where $e(\textbf{super})$ is an expression that can contain one or more occurrences of the **super** keyword.

This notation is syntactic sugar for:

$$\textbf{implementation}\,\mathtt{m}\,\overline{\pi} \Rightarrow \lambda x_1 ... x_n . e(\textbf{super}_{m,\overline{\pi}}\ x_1\ ...\ x_n)$$

We define the target of **super** as the most precise implementation of m that is less precise that the current implementation, or $\perp$ if such implementation does not exist. The ordering of implementations is defined in Definition 43.

Definition 50 (Target of super) *Let m be a method, and $\overline{\pi}$ be a list of patterns. Then*

$$\mathsf{target}(\boldsymbol{super}_{m,\overline{\pi}}) = \max\{\overline{\pi}' \Rightarrow e' \in \mathsf{implementations}(m) \mid \overline{\pi}' < \overline{\pi}\}$$

If there is no implementation of m with patterns less precise than $\overline{\pi}$, or none that is a maximum, the max is undefined, and $\mathsf{target}(\textbf{super}_{m,\overline{\pi}})$ is $\perp$. In that second case, the use of **super** is ambiguous, and therefore results in a typechecking error.

One can then define the semantics of **super**:

Definition 51 (Reduction of super) *Let m be a method, and $\overline{\pi}$ be a list of patterns. If* $\mathsf{target}(\boldsymbol{super}_{m,\overline{\pi}})$ *is $\overline{\pi}' \Rightarrow e'$, then $\boldsymbol{super}_{m,\overline{\pi}} \longrightarrow e'$.*

The semantics of **super** is therefore completely specified by the $\mathsf{target}(\cdot)$ predicate.

5.3.1 Typing

The type given to $\mathbf{super}_{m,\overline{\pi}}$ is the restriction of the type of m to the patterns of the target implementation:

Definition 52 (Type of super) *Let m be a method, and $\overline{\pi}$ be a list of patterns. Then the type of $\boldsymbol{super}_{m,\overline{\pi}}$ is defined by:*

$$\mathsf{constant\text{-}type}(\boldsymbol{super}_{m,\overline{\pi}}) = \begin{cases} \mathbb{E} & \text{if } \mathsf{target}(\boldsymbol{super}_{m,\overline{\pi}}) = \bot \\ \mathsf{restrict}(\mathsf{constant\text{-}type}(m), \overline{\pi}') & \text{if } \mathsf{target}(\boldsymbol{super}_{m,\overline{\pi}}) = \overline{\pi}' \Rightarrow e' \end{cases}$$

Theorem 53 (Super) *For any well typed method m and list of patterns $\overline{\pi}$, the operator $\boldsymbol{super}_{m,\overline{\pi}}$ verifies Requirement 9 (*Constants*).*

Proof of theorem 53 (Super)

1. Since $\mathbf{super}_{m,\overline{\pi}}$ is well-typed by hypothesis, $\mathsf{target}(\mathbf{super}_{m,\overline{\pi}})$ is well defined by Definition 52. Let then $\overline{\pi}' \Rightarrow e'$ be $\mathsf{target}(\mathbf{super}_{m,\overline{\pi}})$. By Definition 51, $\mathbf{super}_{m,\overline{\pi}} \longrightarrow e'$.
2. Let $\overline{\pi}' \Rightarrow e'$ be the implementation $\mathsf{target}(\mathbf{super}_{m,\overline{\pi}})$. The only reduction rule for **super** is $\mathbf{super}_{m,\overline{\pi}} \longrightarrow e'$. In this case, we have by Definition 52 $\mathsf{type}(\mathbf{super}_{m,\overline{\pi}}) = \mathsf{restrict}(\mathsf{type}(m), \overline{\pi}')$ (**1**). Since m is well typed, the implementation $\overline{\pi}' \Rightarrow e'$ must be valid by Definition 46. That is, by Definition 45, $\mathsf{type}(e') \leq \mathsf{restrict}(\mathsf{type}(m), \overline{\pi}')$. This shows, together with (1) that $\mathsf{type}(e') \leq \mathsf{type}(\mathbf{super}_{m,\overline{\pi}})$ holds.

■

5.4 Example

Let us see how **super** can be used in practice, and how the typing rules and the target resolution are applied. We consider a class hierarchy that models buttons in a graphical user interface toolkit:

```
class Button {
  String text;

  // A closure executed when the button is pressed.
  () → void action;
}
void draw(Button);
draw(Button this) { ... }

void clicked(Button);
clicked(Button this) {
  (this.action)();
}

class ImageButton extends Button {
  Image image;
}
draw(ImageButton this) { ... }

class OnOffButton extends Button
{
```

```
  boolean disabled;
}
clicked(OnOffButton this) {
  if (! this.disabled)
    super;
}
```

The `OnOffButton` class adds to the `Button` class the ability to deactivate the button. The use of **super** in the `clicked` method on a `OnOffButton` allows to abstract over the behavior of the parent class. If the parent was modified, this modification will also affect the sub-classes.

The target of **super** in `clicked(OnOffButton` is `clicked(Button)`. Since the method is monomorphic, the type of **super** is simply the type of the method.

Suppose now that we want to use a button that has both an image, and the ability to be deactivated, and that we want to monitor clicks, for instance to count them:

```
class MyButton extends ImageButton, OnOffButton {
  int nbClicks;
}
clicked(MyButton this) {
  this.nbClicks = this.nbClicks + 1;
  super;
}
```

The set of implementations that are less precise than `clicked(MyButton)` contains both `clicked(Button)` and `clicked(OnOffButton)`. Since the latter is more precise than the former, the target of **super** is `clicked(OnOffButton)`.

Subclassing is sometimes presented as similar to a textual copy of the implementations of the parent class that are not redefined in the child class. According to this presentation, one could be worried that the target resolution gives priority to the implementation inherited from `OnOffButton` over the one inherited from `Button` through `ImageButton`. However, the redefined implementation has priority because it indicates that the implementation that it replaces is not wanted in the concerned case. Thus, `clicked(Button)` is not valid for class `OnOffButton` since it does not take into account the activation state. It is therefore not valid either for the subclass `MyButton`, and so must not be considered in the resolution of the target. This corresponds exactly to the notion of most precise implementation, which is used for the dispatch of method calls. Indeed, the same reasoning about the invalidity of overridden implementations also justifies the semantics of dispatch.

Chapter 6

Kinds

6.1 Introduction

This chapter illustrates how the type system can be extended by augmenting the language of constraints of $ML_{\leq}$. We present two typing challenges that arise in practice from the interplay of polymorphism and subtyping, and we propose a single solution to solve them both.

The first challenge, which has already been pointed out [32], is the typing of homogeneous methods, that is, methods that accept several (but not all) types for their arguments, while these types cannot be intermixed. A typical example is the comparison operator less, which can be applied to two strings, two integers, two dates, etc, but not to two values of different types, and neither to types that have no canonical ordering like graphical widgets.

The second challenge, introduced in this thesis, has arisen from our experience with programming in languages with multi-methods and based on the polymorphic constrained type system $ML_{\leq}$ [6]. We found out that many useful methods are partially polymorphic: their types lie in precision in between a monomorphic and a bounded polymorphic type. For instance, considering the hierarchy `Integer` $\leq$ `Rational` $\leq$ `Float`, and the inverse operation $x \mapsto \frac{1}{x}$: the type `Float` $\rightarrow$ `Float` is correct for inverse, but it is too coarse since it does not show that the inverse of a `Rational` is a `Rational`; on the other hand, the type $\forall$`T` $\leq$ `Float`. `T` $\rightarrow$ `T`, which would correctly map a `Rational` to a `Rational`, is incorrect because the inverse of an integer might not be an integer but a rational in general. Our solution is to give inverse the type $\forall\alpha.\, \alpha : \texttt{Field} \Rightarrow \alpha \rightarrow \alpha$, meaning that for every class α that can be given the mathematical structure of a field, the inverse operation maps α to an α. Since integers do not form a field, but a mere subset of the field of rational numbers, our type indeed specifies that the inverse of an `Integer` is a `Rational`. Indeed, the best instantiation for α is then `Rational`. In our proposal `Field` is not a type but a *kind*, that is, a property that some types possess. An extra benefit of our solution is to be *modular* in the sense that new classes with the kind `Field` can be added to an existing class hierarchy without changing the type of the inverse operation.

In this thesis, we propose an extension of $ML_{\leq}$ with *kinds* that allows to typecheck both homogeneous methods and partially polymorphic methods.

In Section 6.2 and Section 6.3, we present several typing challenges that motivate the introduction of kinds in type systems with subtyping; in Section 6.5, we propose a simple solution that however fails to achieve separate type-checking of modules; In Chapter 10, we will show that kinds interact well with modular typechecking: new classes can be added in the domain of an existing method without changing the type of the method. Moreover, our extension of $ML_{\leq}$ also preserves all the essential properties of the system.

We start by examining several challenging type-checking situations of practical importance. Throughout this chapter, we consider type systems with nominal subtyping. Specifically, our examples use class declarations to declare new type names, for which subtyping is determined by the inheritance hierarchy.

6.2 Typing homogeneous operations

We first consider the problem of typing homogeneous operations. Homogeneous operations are a specific sort of binary (or n-ary) operations, characterized by the shape of their domain. They accept several types as arguments. However, values of different types cannot be mixed in the same call [32]. For instance, consider the comparison operator less. Its type must express two properties. First, only some types have a natural ordering. Therefore, comparing graphical widgets should not be possible. Second, even for comparable types like strings, integers, or dates, it should be ill-typed to mix any two of these types in a call: deciding whether a string is smaller than an integer does not make sense in general. We shall examine how it is possible (or not) to express this requirement in several type systems.

Monomorphic type system

The following program is an attempt to type less in a monomorphic type system, using Java syntax.

```
abstract class Comparable {
  boolean less(Comparable other);
}

class String extends Comparable {
  boolean less(Comparable other)
  { /* compares a string to a Comparable */ }
}

class Date extends Comparable {
  boolean less(Comparable other) { ... }
}
```

A monomorphic type system cannot prevent the intermixing of arguments of different types. Consequently, it is necessary to handle the case where a `String` is compared with an arbitrary value of type `Comparable`. Typically, this is done by runtime type inspection. In this case, it would be possible to return false, but since the comparison of a `String` and, say, a `Date` never makes sense, it is probably better to raise a runtime exception if values of different "kinds" are compared. Our aim is precisely to statically rule out these cases.

Monomorphic type system with multi-methods

Because we are still using an inexpressive type system, we cannot express the homogeneity of less. However, a first improvement is that the ability to dispatch on several arguments allows for the separate definition of comparison on pairs on objects of the same "kind", while with mono-methods, each specialized version of less had to handle an arbitrary (second) argument.

```
abstract class Comparable {}

boolean less(Comparable, Comparable); /* Multi-method */

less(Comparable s1, Comparable s2)
{ throw new Error("Trying to compare objects of unrelated classes"); }

class String extends Comparable;
less(String s1, String s2) { ... }

class Date extends Comparable;
less(Date d1, Date d2) { ... }
```

The declaration of the less multi-method makes any couple of two subclasses of `Comparable` valid arguments. The default implementation `less(Comparable, Comparable)` is therefore needed to handle the invalid cases. In the valid implementations, both arguments are statically known to be instances of subclasses of the concerned class.

F-bounded polymorphic type system

F-bounded polymorphism [8] extends bounded polymorphism by allowing the bound of a variable to refer to the type variable being bound. It offers the following solution to type homogeneous operations [32].

```
abstract class Comparable<T> {
  boolean less(T other);
}

class String extends Comparable<String> {
  boolean less(String other) { ... }
}

class Date extends Comparable<Date> {
  boolean less(Date other) { ... }
}
```

The idea is that `x.less(y)` is well typed only if `x` has type `Comparable<T>` for some `T`, and y has type `T`. It is therefore possible to make sure the operation is homogeneous by declaring `class String extends Comparable<String>` and `class Date extends Comparable<Date>` only, so that, for instance, `Date` is not a subtype of `Comparable<String>`.

We shall now propose our solution to this typing problem. The comparison of our system with F-bounded polymorphism will be made in Section 15.3.

Introducing kinds

Between the monomorphic type $(\texttt{Comparable}, \texttt{Comparable}) \to \texttt{boolean}$ for less, which is too loose, and the F-bounded polymorphic type $\forall \texttt{T} \leq \texttt{Comparable<T>}.\ (\texttt{T}, \texttt{T}) \to \texttt{boolean}$, which is unintuitive, we could have considered the simpler bounded polymorphic type $\forall \texttt{T} \leq \texttt{Comparable}.\ (\texttt{T}, \texttt{T}) \to \texttt{boolean}$. However, this does not work: it is in fact equivalent to the monomorphic type, since intuitively, `T` can be instantiated by the type `Comparable`.

In fact, `String` and `Date` share the property of being comparable, without having a common super-type. Therefore `Comparable` should not be a type, but a *property* possessed by some types. In other words, `Comparable` is a *kind*. We shall write `Date : Comparable` or `Date implements Comparable` to express that `Date` is a type of kind `Comparable`. We first present our solution informally.

```
kind Comparable;
<T : Comparable> boolean less(T, T);

class String implements Comparable {
  boolean less(String other) { ... }
}

class Date implements Comparable {
  boolean less(Date other) { ... }
}
```

Since `T` can be instantiated to either `String` or `Date`, pairs of strings or pairs of dates can be compared. Furthermore, no valid instantiation for `T` is a super-type of both `String` and `Date`, which prevents intermixing.

The type $\forall$T : Comparable. (T, T) $\rightarrow$ boolean therefore fulfills the two desired properties for less. In addition, we believe it is less involved than the F-bounded solution.

Inheritance without intermixing problem

Kinds are also appropriate to typecheck more complex cases of homogeneous operations. For instance, Litvinov [32] argues that it is sometimes useful to have a class inherit from another, while not allowing homogeneous operations to accept intermixing the super and the subclass (typically, Points are ColorPoints when it does not makes sense to compare one of each class).

We present here their solution, which is to parameterize both classes and use F-bounded quantification. We use our own syntax when it eases the comparison.

```
class PointF<Pt extends PointF<Pt>> {
  int x = 0; int y = 0;
  int area() = this.x * this.y;

  eqPoint(PointF<Pt> other);
  eqPoint(PointF other) = this.x == other.x && this.y == other.y;
}

class Point is PointF<Point> {}

class ColorPointF extends PointF {
  Color color;
  eqPoint(ColorPoint other) = super && this.color == other.color;
}

class ColorPoint is ColorPointF<ColorPoint> {}
```

Since type parameters are invariant, ColorPointF<ColorPoint> is *not* a subtype of PointF<Point>. Therefore, mixed calls to eqPoint are not well-typed. On the other hand, the area method can as desired be used for both Point and ColorPoint.

Our solution, using kinds, is instead to create a common superclass AbstractPoint containing the features to inherit (the x and y fields), and two classes Point and ColorPoint that implement the kind Comparable. Since AbstractPoint does not implement Comparable, intermixing is prevented.

```
kind Comparable;
<T : Comparable> boolean eqPoint(T, T);

abstract class AbstractPoint
{
  int x = 0; int y = 0;
  int area() = this.x * this.y;
  eqPoint(Point other) = this.x == other.x && this.y == other.y;
}

class Point extends AbstractPoint implements Comparable {
}

class ColorPoint extends AbstractPoint implements Comparable {
  int color = 0;
  eqPoint(ColorPoint other) = super && this.color == other.color;
}
```

Both versions solve the problem as expected: they only allow comparing instances of the same class.

```
void test() {
  eqPoint(new Point(), new Point());
  eqPoint(new ColorPoint(), new ColorPoint());
  //eqPoint(new ColorPoint(), new Point()); // Type Error
}
```

Our approach avoids again the "fake" and cumbersome parameterization. Furthermore, the addition of the class `AbstractPoint` allows to make obvious the fact that a `ColorPoint` is not a `Point`. In the F-bounded version, this fact is not immediately apparent: one must actually try to prove the subtyping —and fail— to conclude that it does not hold. We believe that this makes F-bounded quantification too complex for a widespread use in programming languages.

6.3 Partially polymorphic functions

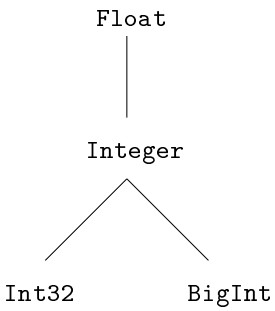
So far, we used kinds to describe a common property of unrelated types. One question immediately follows: how do kinds interact with subtyping? Given a class `A` of kind `K` and a subclass `B` of `A`, should then `B` always be of kind `K`? Actually, a function of type $\forall \mathtt{T} : \mathtt{K}.\ \mathtt{T} \to \mathtt{T}$ can always take an argument whose type `B` is a subtype of type `A` of kind `K`, since by subsumption the argument is also of type `A`. However, using subsumption, we can only conclude that the type of the result is `A`. Conversely, if `B` itself was of kind `K`, then we could type the application by instantiation of `T` by `B`, which would give the result the more precise type `B`.

In this section, we shall show that it is sometimes desired to have the less precise result type: many functions have type that are more precise than $\mathtt{A} \to \mathtt{A}$, but less precise than $\forall \mathtt{T} \leq \mathtt{A}.\ \mathtt{T} \to \mathtt{T}$. We call these functions "partially polymorphic", and now give several examples.

Numerical operations

Consider the following numerical hierarchy:

```
class Float {...}
class Integer extends Float {...}
class Int32 extends Integer {...}
class BigInt extends Integer {...}
```



What is the type of the addition on numbers? The sum of two floats is a float, the sum of two integers is an integer, the sum of a float and an integer is a float. More generally, *the type of the sum of two numbers is their least upper bound* (**1**).

The monomorphic type system and the Hindley-Milner type system do not allow to capture all possible types described by (1) in a single type expression. This explains why arithmetic operators are usually treated apart. However, as we shall see below, this situation also occurs with user defined types, for which *ad hoc* typing is not possible. With bounded polymorphism, it is possible to type `plus` with $\forall \mathtt{T} \leq \mathtt{Float}.\ (\mathtt{T}, \mathtt{T}) \to \mathtt{T}$. This expression correctly captures all possible types described by (1).

However, this type is, in a way, *too precise*: we don't want the sum of two `Int32` to be an `Int32`, but just an `Integer`, because this sum can overflow, in which case the result should be a `BigInt`. Thus, we refine (1), by requiring that the type of the sum always be above `Integer`: *the type of the sum of two numbers is the upper bound of* `Integer` *and of their least upper bound* (**2**). However, bounded polymorphism can not capture all types described by (2) anymore. Intuitively, (2) constrains a type variable with both an upper-bound and

a lower-bound, while bounded polymorphism only allows upper-bounds. Conversely, (2) can be expressed in a constrained polymorphic type system with the type expression $\forall \texttt{Integer} \leq \texttt{T} \leq \texttt{Float}.(\texttt{T},\texttt{T}) \rightarrow \texttt{T}$.

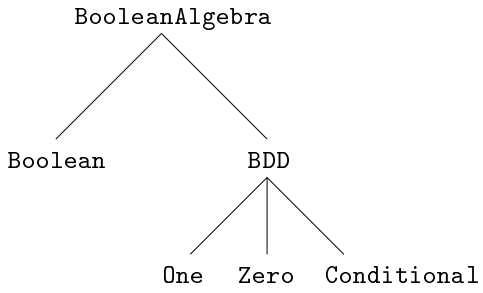
However, this type has the disadvantage that it can only be given once one knows the complete numerical hierarchy. Suppose we now want to subclass `Float` with two implementations that differ with respect to the number of bits used to store the float — `Float32` and `Float64`. Since the new classes are not above `Integer`, the previous type given to `plus` asserts that the sum of two `Float32` may be any `Float`. It seems legitimate to specify that addition of two floats does not change their representation, but this cannot be expressed. Tuning the type using more constraints to match the requirements needs a complete knowledge of the type hierarchy. Therefore, this approach prevents extending the type hierarchy in a flexible way, that is, a modular development of classes. Furthermore, even when the complete hierarchy is known, typing `plus` would require disjunctive constraints, like $\forall \texttt{T}.\, \texttt{Integer} \leq \texttt{T} \vee \texttt{Float32} \leq \texttt{T} \vee \texttt{Float64} \leq \texttt{T} \Rightarrow (\texttt{T},\texttt{T}) \rightarrow \texttt{T}$.

We call *partially polymorphic* the functions that behave like `plus` with respect to types: their types is living somewhere in-between monomorphic types and fully bounded polymorphic types. We believe that they occur rather frequently. Therefore, it is an important issue to handle them appropriately. Please, note that the above situation is very similar to the typing of numerical operators in Java [25]: the sum of two `float` is a `float`, the sum of two `int` is an `int`, but the sum of two `short` (or `byte`) is an `int`. Java handles this situation by *ad hoc* typing rules. Let us give a few more examples.

User defined methods

A similar situation occurs with the typing of, for instance, the negation operator $\neg$ on boolean algebras. This shows that partial polymorphism occurs not only in possibly predefined functions, but also in user defined code. Consequently, this rules out *ad hoc* or non-modular solutions that do not solve the general case.

Indeed, what is the type of this negation operator with respect to a hierarchy that includes both the algebra of booleans and the algebra of binary decision diagrams (BDD) ?



The type `BooleanAlgebra`→`BooleanAlgebra` is very imprecise. It would lead to a big loss of typing information, for instance by having `not(x < y)` be an expression of type `BooleanAlgebra`, given two integers `x` and `y`. The polymorphic type $\forall \texttt{T} \leq \texttt{BooleanAlgebra}.\ \texttt{T} \rightarrow \texttt{T}$ is not correct, since the negation of a constant BDD instance of class `One` is not a `One` but a `Zero`; an union type like $\forall \texttt{T}.\, \texttt{Boolean} \leq \texttt{T} \vee \texttt{BDD} \leq \texttt{T} \Rightarrow \texttt{T} \rightarrow \texttt{T}$ is correct and precise, but disallows the introduction of a new boolean algebra and thus breaks modularity. Furthermore, the introduction of disjunctions in constraints would significantly increase the complexity of the type-checking.

As a last example, consider a hierarchy representing a source program tree inside a compiler or an interpreter. If the source language distinguishes between expressions and statements, it makes sense to declare that class `Expression` is a subclass of class `Statement` because an expression can be considered as a statement that computes and then forgets a value. Many useful functions take a statement and possibly auxiliary arguments, and return a statement: the resolution function that replaces identifiers with a reference to their definition, optimization functions, a macro-expansion function, *etc.* None of these functions are fully polymorphic: name resolution maps identifiers, represented by some class in the hierarchy, to variable definitions, which are of a different class; macro-expansion replaces macro-calls by their definition, which may be arbitrary expression. On the other hand, typing these functions as monomorphic is too coarse: since only expressions are accepted at certain places in a syntax tree (for instance, as the right-hand-side of an

assignment) it is useful to reflect in the type of these functions that expressions are mapped to expressions and not to arbitrary statements.

6.4 Using kinds to type partially polymorphic functions

Let us try to find a common solution for all these situations. Since the problem of finding a satisfactory type to these functions seems difficult to solve, it might be that the problem itself is not formulated properly. Let us reconsider the boolean algebra situation. A `Conditional` is indeed a subclass of `BDD`, because any value of type `Conditional` is a BDD. However, a value of type `BDD` is not itself a `BooleanAlgebra`. It is the *set of all* BDDs that forms a boolean algebra. Therefore, `BooleanAlgebra` is not a super-type of BDD, it is a property of the type BDD.

This situation already occurred in Section 6.2, and motivated the introduction of kinds. This new example additionally involves the interaction of kinding with subtyping. The property of forming a boolean algebra is not true for an arbitrary subset of all BDDs. For example, neither the sets of all `Conditional` BDDs nor the two single-element subsets containing respectively `One` and `Zero` are boolean algebras. Thus, it is crucial that kinding is not inherited. All these observations can be summarized as follows: The property of forming a boolean algebra is represented by the kind `BooleanAlgebra`. Class `BDD` is of kind `BooleanAlgebra`. Class `Zero` is a subclass of `BDD`, and is not of kind `BooleanAlgebra`. The operation `not`, for any class `T` of kind `BooleanAlgebra`, takes a parameter of type `T` and return a value of type `T`. This translates naturally to the following declarations:

```
kind BooleanAlgebra;
<T : BooleanAlgebra> T not(T);

class BDD implements BooleanAlgebra;
class Zero extends BDD;
...

class Boolean implements BooleanAlgebra;
```

Numerical operations can be typed in a similar way. We introduce the kind `Num` to express the property of being a number and give the type $\forall$`T` : `Num`. (`T`, `T`) $\rightarrow$ `T` to `plus`. This type captures all properties of `plus` described above. In particular, all forms of integers equivalently. The type `Int32` does not have kind `Num`. Hence, the "best" solution for `T` when `plus` is applied to an `Int32` is "`T=Integer`". Thus, the only guarantee for the return type is to be below `Integer`.

Kinds can also be viewed as an open set of classes with names. This approach allows for new classes to be added to a kind without having to modify the type of methods operating on the classes of this kind. One reason that makes this solution more modular than an approach based on disjunctive constraints is that whenever we introduce a new class in the numerical hierarchy, we are able to determine its behavior relatively to the kind `Num`. Additionally, we believe that the types are also shorter to write, easier to understand, and easier to handle in a type-checking algorithm.

This solution also gives an arguably more intuitive type to `plus`. We believe this is an important issue to ensure that powerful type systems can be used in wide-spread programming languages. Using kinds, the type can be explained in simple words: "`plus` has type (`T`,`T`)$\rightarrow$`T` for every numerical class `T`". In our view, `Int32` is *not* a numerical class (that is, a class of kind `Num`), but an *implementation* of a numerical class.

6.5 Closed-world formalization

We present a first attempt to formalize a type system with kinds. In this section, we will make the *closed-world* assumption. That is, we will consider that type-checking is made for whole programs only, so that there is no difference between the type structure in which an expression is type-checked and the global type-structure of the running program. This has two purposes. First, it allows the use of simpler typing rules,

that are useful for an intuitive understanding of types involving kinds. Second, it serves as a motivation for the more complex rules of Chapter 10, where typing will take place in an open world. Hence, results in this sections are subsumed by Chapter 10.

We extend in Figure 6.1 the $\text{ML}_{\leq}$ type algebra of Section 2.2. Type structures now also include a set $\mathcal{K}$ of kinds, and a new relation denoting kinding ($c_V : K$). Constraints include kinding constraints. In the previous examples, we used the notations $\forall T \leq \theta.\, \theta'$ and $\forall T : K.\, \theta'$ as shorthands for $\forall T.\, T \leq \theta \Rightarrow \theta'$ and $\forall T.\, T : K \Rightarrow \theta'$ respectively.

Type structure	$\mathcal{T}$	::=	$(\mathcal{C}, \mathcal{K}, \leq, :)$
Constraint	κ	::=	$\theta \leq \theta \mid \phi_V \leq \phi_V \mid \phi_V : K$

Figure 6.1: Extensions to $\text{ML}_{\leq}$

A type can be interpreted as the upward-closing of the set of its ground instances that satisfy the constraint. Given a type, we define its denotation as:

$$\mathsf{den}(\forall \overline{\alpha}.\, \overline{\kappa} \Rightarrow \theta) = \{\theta', \exists \sigma.\ \sigma(\theta) \leq \theta' \text{ and } \sigma(\overline{\kappa}) \text{ hold}\}$$

where θ' ranges over ground types and σ ranges over mappings from type variables to ground types. Each constraint in $\sigma(\overline{\kappa})$ is of the form $\theta_1 \leq \theta_2$ or $\theta : K$ and can be readily interpreted as true or false in the type structure.

For instance, the type of plus in Section 6.3, $\forall T.\, T : \texttt{Num} \Rightarrow (T, T) \to T$ is denoted by the upward closing of set $\{(\texttt{Float}, \texttt{Float}) \to \texttt{Float}, (\texttt{Integer}, \texttt{Integer}) \to \texttt{Integer}\}$. In particular, the closure contains super-types of the above two types that describe how any pair of two types is mapped to a result type: $(\texttt{Float}, \texttt{Integer}) \to \texttt{Float}$, $(\texttt{Int32}, \texttt{Int64}) \to \texttt{Integer}$, ... This corresponds to our intuition of the type behavior of the addition.

Given this interpretation of types, it is easy to define sub-typing and type-checking. Type τ_1 is a subtype of τ_2 if $\mathsf{den}(\tau_2) \subseteq \mathsf{den}(\tau_1)$. Instantiation and generalization rules ensure that an expression has the polymorphic type τ if and only if it has all the monomorphic types in the denotation of τ.

This formalization is only correct and safe in a closed-world. Therefore, it could be used to type-check entire programs, but not program modules taken separately. We present in Chapter 10 a variation on the theory that accommodates with modular type-checking. We provide there a complete formalization and proofs, and a comparison with related work.

Part III

Modularity

Chapter 7

Modular type algebras

When a program is made of modules, it must be possible to typecheck each module independently. Furthermore, a module can import another module and add class declarations, which modifies the type structure of the type algebra. It must therefore be possible to guarantee that code typechecked in the original algebra is still well-typed in the new algebra.

To this end, we define the notion of a type algebra extension.

Definition 54 (Type algebra extension) *A type algebra $(A', \leq')$ is an extension of a type algebra $(A, \leq)$ if and only if $A \subseteq A'$ and for all syntactic types τ_1 and τ_2 in $S(A)$ such that $\tau_1 \leq \tau_2$ holds, $\tau_1 \leq' \tau_2$ also holds.*

Note that the condition that A is contained in A' ensures that $S(A)$ is contained in $S(A')$, which in turn makes the inequality $\tau_1 \leq' \tau_2$ well-formed.

7.1 $\mathrm{ML}_{\leq}$

We now give conditions that guarantee that an $\mathrm{ML}_{\leq}$ type algebras is an extension of another. Since we present a variant of $\mathrm{ML}_{\leq}$ in Chapter 10, we first formalize the possibility to create a variant of $\mathrm{ML}_{\leq}$ by extending the constraint language.

7.1.1 Variants of $\mathrm{ML}_{\leq}$

A variant of $\mathrm{ML}_{\leq}$ can extend, compared to the original version of Section 2.2,

- the syntax of constraints;
- the set of axioms defining constraint implication;
- the notion of type structure $\mathcal{T}$.

Furthermore, for enabling modular typing, we assume given a transitive predicate allowing to state that a type structure $\mathcal{T}'$ is an extension of another type structure $\mathcal{T}$, written $\mathcal{T}' \geq \mathcal{T}$.

The only condition required for a variant of $\mathrm{ML}_{\leq}$ is that the axiomatization of constraint implication be correct and complete, as we define below in Definition 55. Indeed, we already proved in Section 2.2 that $\mathrm{ML}_{\leq}$ is a type algebra. This proof uses only the axioms of constraint implication, which are also present in the considered variant of $\mathrm{ML}_{\leq}$. Therefore, the proof remains valid, and the variant is also a type algebra. On the other hand, the axiomatization is extended, and must therefore be proved again correct and complete.

Definition 55 (Correction and completeness) *A* $\mathrm{ML}_{\leq}$ *axiomatization* $\models$ *is correct and complete if the following property holds: for all type structure* $\mathcal{T}$*, variable list* ϑ *and constraints* κ_1 *and* κ_2 *in* $\mathcal{T}$*, the constraint implication* $\forall\vartheta.\ \kappa_1 \models \kappa_2$ *is provable if and only if for all extension* $\mathcal{T}'$ *of* $\mathcal{T}$*, for all ground substitution* σ_1 *such that* $\mathcal{T}' \vdash \sigma_1(\kappa_1)$*, there exists a ground substitution* σ_2 *such that* $\sigma_2 \overset{\vartheta}{=} \sigma_1$ *and* $\mathcal{T}' \vdash \sigma_2(\kappa_2)$*.*

The notation $\sigma_1 \overset{\vartheta}{=} \sigma_2$ means, as in Section 2.1, that substitutions σ_1 and σ_2 are equal for all elements of ϑ.

In correct and complete axiomatizations, one can offer an interpretation of subtyping between polytypes.

Corollary 56 (Interpretation) *Let* τ_1 *and* τ_2 *be two closed types. Then* $\tau_1 \leq \tau_2$ *holds in a type structure* $\mathcal{T}$ *if and only if in every extension* $\mathcal{T}'$ *of* $\mathcal{T}$*, for every ground instance* θ_2 *of* τ_2 *in* $\mathcal{T}'$ *there is a ground instance* θ_1 *of* τ_1 *such that* $\mathcal{T}' \vdash \theta_1 \leq \theta_2$*.*

Proof of corollary 56 (Interpretation)

Let $\forall\vartheta_1.\ \kappa_1 \Rightarrow \theta_1$ be τ_1 and $\forall\vartheta_2.\ \kappa_2 \Rightarrow \theta_2$ be τ_2. By Definition 27, $\tau_1 \leq \tau_2$ is equivalent to the constraint implication $\forall t.\ \kappa_2 \wedge \theta_2 \leq t \models \kappa_1 \wedge \theta_1 \leq t$ where t is a fresh variable, that is not free in $\kappa_1, \kappa_2, \theta_1, \theta_2$ (**1**).

By Definition 55, this constraint implication is equivalent to

$$\begin{array}{l} \forall \mathcal{T}' \geq \mathcal{T}, \forall \sigma_1 \mid \mathcal{T}' \vdash \sigma_1(\kappa_2 \wedge \theta_2 \leq t), \\ \exists \sigma_2 \mid \mathcal{T}' \vdash \sigma_2(\kappa_1 \wedge \theta_1 \leq t) \text{ and } \sigma_2(t) = \sigma_1(t) \end{array} \tag{7.1}$$

On the other hand, the target proposition is:

$$\begin{array}{l} \forall \mathcal{T}' \geq \mathcal{T}, \forall \sigma \mid \mathcal{T}' \vdash \sigma(\kappa_2), \\ \exists \sigma' \mid \mathcal{T}' \vdash \sigma'(\kappa_1) \text{ and } \mathcal{T}' \vdash \sigma'(\theta_1) \leq \sigma(\theta_2) \end{array} \tag{7.2}$$

Since all constraints are evaluated in the extension structure $\mathcal{T}'$, we may leave $\mathcal{T}'$ implicit.

Let us prove that 7.1 implies 7.2. For $\mathcal{T}' \geq \mathcal{T}$ and σ such that $\mathcal{T}' \vdash \sigma(\kappa_2)$ (**2**), let σ_1 be $\sigma + \{t \mapsto \sigma(\theta_2)\}$ (**3**). Since $\sigma(\kappa_2)$ holds by (2), $\sigma_1(\theta_2) = \sigma(\theta_2)$ by (1) and $\sigma_1(t) = \sigma(\theta_2)$ by (3), the premise $\sigma_1(\kappa_2 \wedge \theta_2 \leq t)$ of 7.1 holds. Therefore, by 7.1, there exists a substitution σ_2 such that $\sigma_2(\kappa_1 \wedge \theta_1 \leq t)$ (**4**) and $\sigma_2(t) = \sigma_1(t) = \sigma(\theta_2)$ (**5**) hold. One can therefore take $\sigma' = \sigma_2$, since $\sigma_2(\kappa_1)$ holds by (4) and $\sigma_2(\theta_1) \leq \sigma_2(t) = \sigma(\theta_2)$ holds by (4) and (5).

Conversely, for given $\mathcal{T}' \geq \mathcal{T}$ and σ_1 such that $\sigma_1(\kappa_2 \wedge \theta_2 \leq t$ (**6**), we have in particular by hypothesis $\sigma_1(\kappa_2)$. Therefore, by 7.2, there exists σ' such that $\sigma'(\kappa_1)$ holds (**7**) and $\sigma'(\theta_1) \leq \sigma_1(\theta_2)$ holds (**8**). Let σ_2 be $\sigma' + \{t \mapsto \sigma_1(t)\}$ (**9**). We then have $\sigma_2(\kappa_1)$ by (7) and (1). Furthermore, $\sigma_2(\theta_1 \leq t)$ is equivalent by (9) and (1) to $\sigma'(\theta_1) \leq \sigma_1(t)$, which is true by transitivity on (8) and hypothesis $\sigma_1(\theta_2) \leq \sigma_1(t)$ (6). ■

Theorem 57 (Extension of a ML-Sub type algebra) *If an* $\mathrm{ML}_{\leq}$ *type structure* $\mathcal{T}'$ *is an extension of a type structure* $\mathcal{T}$*, then the type algebra* $\mathcal{A}(\mathcal{T}')$ *is an extension of* $\mathcal{A}(\mathcal{T})$*.*

Proof of theorem 57 (Extension of a ML-Sub type algebra)

By corollary 56 (Interpretation), in every extension $\mathcal{T}_0$ of $\mathcal{T}$, for every ground instance θ' of τ' in $\mathcal{T}_0$ there is a ground instance θ of τ such that $\mathcal{T}_0 \vdash \theta \leq \theta'$. Every extension of $\mathcal{T}'$ is also an extension of $\mathcal{T}$ since extension is transitive. Therefore this also holds for every extension of $\mathcal{T}'$. Therefore by corollary 56 (Interpretation) $\mathcal{T}' \vdash \tau \leq \tau'$. ■

7.1.2 Original $\mathrm{ML}_{\leq}$

In particular, the original version of $\mathrm{ML}_{\leq}$ defined in [5] has a correct and complete axiomatization. In that setting, the extension of a type structure is defined in the following way:

Definition 58 (Extension of a ML-Sub type structure) *A type structure* $\mathcal{T}'$ *is an extension of* $\mathcal{T}$ *if all type constructors of* $\mathcal{T}$ *are in* $\mathcal{T}'$ *and if for all type constructors* c_1 *and* c_2 *of* $\mathcal{T}$*,* $\mathcal{T}' \vdash c_1 \leq c_2$ *if and only if* $\mathcal{T} \vdash c_1 \leq c_2$*.*

These conditions correspond to the usual extension of the type structure found in object oriented system for a module importing other modules: the existing types are not modified, but new types can be freely added.

The proof that the original version of $ML_{\leq}$ verifies Definition 55 is done in the $ML_{\leq}$ report [5].

Chapter 8

Open generic functions

It has already been recognized [19] that the activity of programming has two main sides: defining operations and defining data structures. The functional paradigm mainly uses sum and product types as its data structures, and functions defined by pattern-matching on data-types as its operations. The object-oriented paradigm provides classes to structure data, and methods to operate on it. However, both paradigms introduce an asymmetry between the two concepts. In a functional program, data-types can be defined independently of functions, while functions need knowledge about the data-type constructors. Conversely, methods are defined locally to a class, while classes include the list of all their methods.

This asymmetry is problematic when it comes to modular programming, that is, programming reusing pre-compiled libraries, without changing them [26]. Following the above dualism, modular programming is thus both defining new operations on existing data structures and defining new data structures to be handled by existing operations. In the functional paradigm, defining new functions is straightforward. On the other hand, extending existing datatypes is not possible since it would break existing functions defined by pattern matching on this datatype, which would miss the new cases. Conversely, extending data structures in an object-oriented setting amounts to writing new classes, while defining new methods on existing classes is not allowed.

In Chapter 4, all implementations of a generic function had to be syntactically present together with the definition of the function. This is similar to pattern matching in a functional language, and thus fails in the same way to provide support for modular programming. The solution is to *open* generic functions, so that implementation branches can be defined independently, which brings back symmetry. An open generic `opp` can thus be defined as:

generic `opp` : $\forall t \leq$ `num`. $t \to t$
implementation `opp float` $\Rightarrow$ `opp_float`
implementation `opp int` $\Rightarrow$ `opp_int`

If the new type of complex numbers, subtype of `num`, is added in a different module that imports the above one, a new implementation of `opp` can —and must— be provided:

implementation `opp complex` $\Rightarrow$ ...

We illustrate the two situations encountered with modular programming by taking the example of a small programming language implementation. The structure of this implementation is presented in Figure 8.1. Modularity is expressed by the fact that declarations are grouped inside modules. Each module is represented by **module** `NAME` {...} and is typically written separately from other modules.

```
module CORE {
  abstract class Expression {}
  class Apply extends Expression
    { f : Expression, arg : Expression }

  generic eval : Expression → Expression
  generic print : Expression → String

  implementation eval Apply ⇒ ...
  implementation print Apply ⇒ ...
}

module NUMERIC imports CORE {
  class IntegerLiteral extends Expression ...

  implementation eval IntegerLiteral ⇒ ...
  implementation print IntegerLiteral ⇒ ...
}

module COMPILER imports CORE and NUMERIC {
  generic compile : Expression → Code

  implementation compile Apply ⇒ ...
  implementation compile IntegerLiteral ⇒ ...
}
```

Figure 8.1: Programming with open generic functions

8.1 Syntax and semantics

The syntax for programs with open generic functions is:

Generic function	$\mathcal{G}$	::=	**generic** g : τ
Implementation	$\mathcal{I}$	::=	**implementation** g $\overline{\pi} \Rightarrow e$
Class	$\mathcal{C}$	::=	**class** C **extends** $\overline{C}$ { ... }
Declaration	$\mathcal{D}$	::=	$\mathcal{G} \mid \mathcal{I} \mid \mathcal{C}$
Module	$\mathcal{M}$	::=	**module** M **imports** $\overline{M}$; **let rec** $\overline{\mathcal{D}}$
Program	$\mathcal{P}$	::=	$\overline{\mathcal{M}}$; **eval** e

A module **module** M **imports** $\overline{M'}$; **let rec** $\overline{\mathcal{D}}$ declares a module with name M that imports the modules whose names are in the list $\overline{M'}$. A program $\mathcal{M}_0; \mathcal{M}_1 ... \mathcal{M}_n$; **eval** e consists of a main module $\mathcal{M}_0$, additional modules $\mathcal{M}_1 ... \mathcal{M}_n$ that can be imported by $\mathcal{M}_0$, and an expression e which expresses the desired behavior of the program.

Since modules can be considered independently, they refer to each other – in their import lists – by name. Therefore, we need a way to map a module name to its definition. We will call *module repository* a function from module names into modules. We can associate a module repository to each program:

Definition 59 (Module repository) *Let $\mathcal{P}$ be the program $\mathcal{M}_0; \mathcal{M}_1 ... \mathcal{M}_n$; **eval** e. For i from 0 to n, let M_i be the name of module $\mathcal{M}_i$. The module repository for $\mathcal{P}$, written* repository($\mathcal{P}$)*, is then the function $\{M_0 \mapsto \mathcal{M}_0, ..., M_n \mapsto \mathcal{M}_n\}$.*

A natural semantics of a program $\mathcal{P}$ is defined by translation into the generic functions of Chapter 4. The generic functions found in all modules are reconstructed by grouping the implementations with their respective declarations.

Definition 60 (Closure of a modular program) *Let $\mathcal{P}$ be the modular program $\mathcal{M}_0; \mathcal{M}_1 ... \mathcal{M}_n$;* ***eval*** *$e$ and let R be* repository($\mathcal{P}$). *The closure $\mathcal{P}$ is the non-modular program* ***let rec*** (classes($\mathcal{M}_0$)) $\cup$ generics($\mathcal{M}_0, \mathcal{M}_0 ... \mathcal{M}_n$) ***in*** *$e$, written* closure($\mathcal{P}$), *where*

$$\text{classes}(\textbf{\textit{module}}\ M\ \textbf{\textit{imports}}\ M_1 ... M_n; \textbf{\textit{let rec}}\ \overline{\mathcal{G} \mid \mathcal{I} \mid \mathcal{C}}) = \overline{\mathcal{C}} \cup \bigcup_{i=1}^{n} \text{classes}(R(M_i))$$

$$\text{generics}(\textbf{\textit{module}}\ M\ \textbf{\textit{imports}}\ M_1 ... M_p; \textbf{\textit{let rec}}\ \overline{\mathcal{G} \mid \mathcal{I} \mid \mathcal{C}}, \mathcal{M}_0 ... \mathcal{M}_n)$$
$$=$$
$$\{\textbf{\textit{generic}}\ \mathsf{g}\ :\ \tau\ = \bigcup_{i=0}^{n} \text{implementations}(g, \mathcal{M}_i) \mid \textbf{\textit{generic}}\ \mathsf{g}\ :\ \tau \in \overline{\mathcal{G}}\} \cup \bigcup_{i=1}^{p} \text{generics}(R(M_i), \mathcal{M}_0 ... \mathcal{M}_n)\}$$

$$\text{declarations}(\textbf{\textit{module}}\ M\ \textbf{\textit{imports}}\ \overline{M}; \textbf{\textit{let rec}}\ \overline{\mathcal{G} \mid \mathcal{I} \mid \mathcal{C}}) = \overline{\mathcal{G}}$$

$$\text{implementations}(g, \textbf{\textit{module}}\ M\ \textbf{\textit{imports}}\ \overline{M}; \textbf{\textit{let rec}}\ \overline{\mathcal{G} \mid \mathcal{I} \mid \mathcal{C}})$$
$$=$$
$$\{\overline{\pi} \Rightarrow e \mid \textbf{\textit{implementation}}\ \mathsf{g}\ \overline{\pi} \Rightarrow e \in \overline{\mathcal{I}}\}$$

The program can then be evaluated as before. The advantage of open generic functions is that programs can now be decomposed into modules, and that modules can be typechecked separately.

8.2 Modular type-checking

A module type-checks if all its implementations are *valid* according to Definition 45. Since this definition is independent of other implementations, modules can be type-checked separately.

Formally, we need to define the aspects of a module that are relevant for type-checking other modules. These can be divided in two: the declarations of classes, which affect the type-algebra in which type-checking occurs, and the implementations of generic functions, which are needed to check the coverage and non-ambiguity of generic functions.

Definition 61 (Module signature)
Let $\mathcal{M}$ be the module ***module*** *M* ***imports*** *$M_1, ..., M_n$;* ***let rec*** *$\overline{\mathcal{D}}$, and let R be a module repository. The* type signature *of $\mathcal{M}$ in R is*

$$\text{typesig}(\mathcal{M}, R) = \text{classes}(\mathcal{M}) \cup \bigcup_{i=1}^{n} \text{typesig}(R(M_i))$$

The generic function signature *of $\mathcal{M}$ in R is*

$$\text{gensig}(\mathcal{M}, R) = \text{declarations}(\mathcal{M}) \cup\ \{g\ \overline{\pi} \mid \textbf{\textit{implementation}}\ \mathsf{g}\ \overline{\pi} \Rightarrow e \in \overline{\mathcal{D}}\} \cup \bigcup_{i=1}^{n} \text{gensig}(R(M_i))$$

Finally, the signature of $\mathcal{M}$ in R is the union of both:

$$\text{sig}(\mathcal{M}, R) = \text{typesig}(\mathcal{M}, R) \cup \text{gensig}(\mathcal{M}, R)$$

Basically, the signature of a module erases the body of method implementations and keeps only toplevel declarations. It is apparent from this definition that the imported definitions are searched recursively in the signature of imported modules.

We require a function that associate to a type signature S a type algebra $A(S)$. Given a module $\mathcal{M}$, we then define the associated type algebra $A(\mathcal{M})$ as $A(\text{typesig}(\mathcal{M}))$. To guarantee type-safety, we require that this algebra is an extension of the algebras of all imported modules.

Requirement 62 (Module import) *Let $\mathcal{M}$ be a module that imports modules $\mathcal{M}_1, ..., \mathcal{M}_n$. Then, for all i from 1 to n, the type algebra $A(\mathcal{M})$ must be an extension of the type algebra $A(\mathcal{M}_i)$, as specified by Definition 54.*

Definition 63 (Type correct module)
Let ***module*** *M* ***imports*** *$M_1, ..., M_n$;* ***let rec*** *$\overline{\mathcal{D}}$ be a module $\mathcal{M}$. Module $\mathcal{M}$ is* type correct *if every implementation* ***implementation*** $\mathsf{g}\ \overline{\pi} \Rightarrow e$ *in $\overline{\mathcal{D}}$ is valid for τ (as specified in Definition 45) in the type algebra $A(\mathcal{M})$, where τ is the declared type of g in* $\mathrm{sig}(\mathcal{M})$.

The following coverage condition is similar to Definition 48, except that it can now be checked using only the information contained in the signature of modules.

Definition 64 (Coverage)
A generic signature GS is covered *in a type algebra A if for all* ***generic*** $\mathsf{g} : \tau$ *in GS, for all run-time types $\tau_1, ..., \tau_n$ in A such that $\tau\ \tau_1\ ...\ \tau_n \neq \mathbb{E}$, there exists an implementation $g\ \overline{\pi}$ in GS such that $\overline{\pi}$ matches $(\tau_1, ..., \tau_n)$.*

Theorem 65 (Modular type-checking) *Let $\mathcal{P}$ be the program $\mathcal{M}_0; \mathcal{M}_1 ... \mathcal{M}_n$;* ***eval*** *$e$ and R be* repository$(\mathcal{P})$. *The equivalent non-modular program* closure$(\mathcal{P})$ *is well-typed if*

- *every module in $\mathcal{M}_0 ... \mathcal{M}_n$ is type correct;*
- *e is well typed in $A(\mathcal{M}_0)$;*
- *and* $\mathrm{gensig}(\mathcal{M}_0, R)$ *is covered in $A(\mathcal{M}_0)$.*

The proof relies mainly on the fact that the type algebra associated with the whole program is an extension of the type algebra of every module, in which their code is typechecked.

Proof of theorem 65 (Modular type-checking)

By Theorem 49 (Generic functions), we only need to prove that every generic function in closure$(\mathcal{P})$ is covered and valid.

First, the type algebra $A(\mathcal{M}_0)$ is the same as that of closure$(\mathcal{P})$ since by Definition 60 and Definition 61 they have exactly the same classes. Let **generic** $\mathsf{g} : \tau = \mathcal{I}_1, ... \mathcal{I}_p$ be a generic function in closure$(\mathcal{P})$ (**1**) and let $\tau_1, ..., \tau_n$ be run-time types in $A(\mathcal{M}_0)$ such that $\tau\ \tau_1\ ...\ \tau_n \neq \mathbb{E}$ (**2**). By (1) and Definition 60, there exists a module $\mathcal{M}$ transitively imported by $\mathcal{M}_0$ such that **generic** $\mathsf{g} : \tau$ belongs to the generic declarations of $\mathcal{M}$. Therefore, by Definition 61, **generic** $\mathsf{g} : \tau$ belongs to $\mathrm{gensig}(\mathcal{M}_0, R)$. By hypothesis, $\mathrm{gensig}(\mathcal{M}_0, R)$ is covered in $A(\mathcal{M}_0)$. Therefore, by Definition 64 and (2), there exists an implementation $g\ \overline{\pi}$ in $\mathrm{gensig}(\mathcal{M}_0, R)$ such that $\overline{\pi}$ matches $(\tau_1, ..., \tau_n)$ (**3**). By Definition 61, there exists a module $\mathcal{M}'$ transitively imported by $\mathcal{M}_0$ and there exists an expression e such that **implementation** $\mathsf{g}\ \overline{\pi} \Rightarrow e$ belongs to the declarations of $\mathcal{M}'$. Therefore, by Definition 60, $\overline{\pi} \Rightarrow e$ is an implementation of g in closure$(\mathcal{P})$ that matches $(\tau_1, ..., \tau_n)$ by (3). This shows by Definition 48 that generic function g is covered in closure$(\mathcal{P})$.

We also need to prove that an implementation of a generic function g that is valid in a module $\mathcal{M}$ is valid in a program $\mathcal{P}$ that contains that module. Let $\mathcal{M}_0$ be the main module of $\mathcal{P}$. Let the implementation be $\overline{\pi} \Rightarrow e$, τ be $\mathsf{type}(g)$, τ' be $\mathsf{restrict}(\tau, \overline{\pi})$, and τ_e be $\mathsf{type}(e)$. Since $\mathcal{M}$ is type correct by hypothesis, the implementation is valid by Definition 63. Therefore, by Definition 45, $\tau_e \leq \tau'$ holds in $A(\mathcal{M})$. By applying Requirement 62 (Module import) to the chain of imports from $\mathcal{P}$ to $\mathcal{M}$, $A(\mathcal{M}_0)$ is an extension of $A(\mathcal{M})$. Therefore, by Definition 54, $\tau \leq \tau'$ also holds in $A(\mathcal{M}_0)$, and the implementation is valid for the whole program $\mathcal{P}$. ■

8.3 Early detection of errors

Checking coverage and non-ambiguity must be done for the (whole) program, that is when all the implementations and types are known. It is therefore necessary to postpone these checks until the whole program

is known. However, this scheme for checking coverage is problematic because it leaves much freedom about the module in which to define a certain implementation. This is arguably a problem from the software-engineering point of view. For example, it would be possible in our compiler example of Figure 8.1 to omit the implementation of `eval` in module `NUMERIC`. At the point of linking the modules together, the coverage test would fail for `eval`. Solving this failure would amount to adding the missing implementation. But since it logically belongs to module `NUMERIC`, this solution breaks modularity by forcing the update of an already compiled module.

Therefore, we believe it is a good design to check coverage in every module. This check is done in the typing context made of all declarations present in this module and all the modules it imports transitively. Thus, apart from errors local to the module, the coverage test can only fail when a module imports a generic function, and defines new types in its domain without providing the corresponding implementation, or when it imports two modules leading to the same situation. These are indeed the situation where an implementation logically belong to this module. The error can then be solved locally by including this implementation in the module, without breaking any existing code. This coverage testing scheme also enforces an intuitive organization of code:

- when a module extends an existing data structure, it must define the implementations for all the generic functions that operate on this data structure;
- when a module defines a new generic function, it must define its implementations for all the data structures in its domain.

The list of all generic functions and data structures is drawn from the typing context defined at the beginning of this paragraph. An example of the first point is the implementation of `eval` in module `NUMERIC`; the implementations of `compile` in module `COMPILER` illustrates the second point.

Additionally, the location where a method implementation is placed can be further constrained even in the case where it could have been omitted. This is done by requiring that the implementation is written as early as possible, instead of being delayed to client modules. This requirement is formalized in the following definition. In addition to improving the organization of programs, this rule is important in the presence of super calls, as we illustrate in Chapter 9.

Definition 66 (Precocity rule) *An implementation* ***implementation*** $\mathrm{g}\ \overline{\pi} \Rightarrow \ldots$ *is not valid in a module* $\mathcal{M}$ *if* g *and* $\overline{\pi}$ *are visible together in a single module imported by* $\mathcal{M}$*.*

Furthermore, it is possible to make the coverage test unnecessary, at the price of a loss of generality for generic functions. A well-known case is mono-methods, used in object-oriented languages with single dispatch. These methods select an implementation based on the runtime type of a distinguished "receiver" argument. Furthermore they are always defined together with the class for which they provide an implementation. The coverage test then reduces to checking that an implementation is present if none is inherited from the parent classes—which could happen if the parents are abstract. This check can therefore be done with the sole knowledge of the class definition and of its parents. In our presentation, this corresponds to restrict implementations to only one pattern on the first argument. It is then sufficient to check that whenever a new type is defined, all necessary implementations are also provided. More elaborate restrictions for multi-methods are studied in [33].

8.4 Type inference for open generic functions

It is an open problem to infer types for the generic functions of Chapter 4. However, we claim here that it does not make sense to try to infer types for *open* generic functions. In short, the idea is that inference requires the knowledge of the implementation of the function. For open generic functions, only some implementations might be known in a module. The type inferred might then be too precise to capture the intent of the functions, which would either prevent proper implementation in client modules, or break modularity by requiring the imported module to be re-type-checked.

As a degenerate case, it is perfectly legal to declare an open generic function with no implementation at all. This makes sense if the function operates on a hierarchy rooted at an abstract class, with no known subclasses. That is to say that the module could only contain:

generic g

How can we possibly infer a type for g? If we choose $\forall t.t$, one will not be able to implement g in a client module. Conversely, we could pick an arbitrary type for g, but it would in general not match the usage of the generic. One possibility would be to infer the type from the usage of g in the current module. But it could happen that g is not used either, in which case the problem remains. Consequently, we believe that it is natural for users to declare the types of generics, since they are toplevel definitions exported to the clients of the module and since their type is therefore the specification of their behavior.

8.5 $\mathrm{ML}_{\leq}$

In this section, we show that $\mathrm{ML}_{\leq}$ is an appropriate type algebra for the modular typechecking of multi-methods. To this end, we need to define the type algebra associated to a module, and show that it meets Requirement 62 (Module import).

An $\mathrm{ML}_{\leq}$ type-structure is associated to each module $\mathcal{M}$ with

$$\mathcal{T}_M = (\{C \mid \textbf{class } C \ ... \in \mathrm{typesig}(\mathcal{M})\}, \{C_1 \leq C_2 \mid \textbf{class } C_1 \textbf{ extends } ..., \ C_2, \ ... \in \mathrm{typesig}(\mathcal{M})\})$$

The type algebra $A(\mathcal{M})$ associated to module $\mathcal{M}$ is then simply the $\mathrm{ML}_{\leq}$ type algebra based on $\mathcal{T}_M$.

Lemma 67 (Module import in ML-Sub) *If module $\mathcal{M}$ imports module $\mathcal{M}'$, then $\mathcal{T}_{\mathcal{M}}$ is an extension of $\mathcal{T}_{\mathcal{M}'}$.*

Proof of lemma 67 (Module import in ML-Sub)
By Definition 61, all type constructors (classes) of $\mathcal{M}'$ are in $\mathcal{M}$, and the sub-constructor relations over type constructors are preserved. Moreover, every new sub-constructor declarations concern new type constructors. Therefore, by Definition 54, $T_{\mathcal{M}}$ is an extension of $T_{\mathcal{M}'}$. Theorem 57 (Extension of a ML-Sub type algebra) then shows that $A(\mathcal{M})$ is an extension of $A(\mathcal{M}')$. ■

The proof of Requirement 62 (Module import) is a direct consequence of this lemma and Theorem 57 (Extension of a ML-Sub type algebra).

8.6 $\mathrm{ML}_{\leq}$ multi-methods

We present here multi-methods as a particular case of open generic functions. Multi-methods are generic functions whose patterns match values depending on the class these values are an instance of. We therefore define a language of patterns to express such matching. Furthermore, we define the restriction predicate $\mathsf{restrict}(\cdot,\cdot)$ that guarantees the type-correctness of multi-methods in the case of the $\mathrm{ML}_{\leq}$ type algebra.

8.6.1 Syntax

The abstract syntax for a multi-method declaration is the same as for open generic functions. However, for clarity, we replace the **generic** keyword by **method**:

method $m : \tau_m$

Method implementations are identical to generic function implementations, prefixed by the **implementation** keyword.

Patterns, the $\in$ predicate and the restriction function remain to be defined. The language of patterns is defined by:

$$\pi \ ::= \ \_ \mid @C \mid \#C$$

The intent is that the pattern $\_$ matches any expression. The pattern $@C$ matches any instance of either class C or of one of its subclasses. This means that an implementation with pattern $@C$ can also be used for sub-classes of C, which is usual in object-oriented languages. On the other hand, the pattern $\#C$ only matches instances of class C. This pattern is necessary to implement some polymorphic multi-methods, for which no implementation can have a precise enough type to be valid for any subclass. We shall illustrate this situation with an example in Section 8.6.3. Definitions 68 and 69 formalize this informal presentation of patterns.

8.6.2 Type-checking

We define multi-methods as an instance of open generic functions by providing the matching and restriction predicates, and we check that they meet their requirements.

Definition 68 (Pattern constraint) *For any pattern π and type τ we define the* pattern constraint $\pi(\tau)$ *by:*

$$\begin{array}{rcl} \_\,(\forall\vartheta.\,\kappa \Rightarrow \theta) & = & \mathit{true} \\ @C(\forall\vartheta.\,\kappa \Rightarrow \theta) & = & \kappa \wedge \theta \leq C[\overline{t}] \\ \#C(\forall\vartheta.\,\kappa \Rightarrow \theta) & = & \kappa \wedge \theta = C[\overline{t}] \end{array}$$

where $\overline{t}$ are lists of fresh type variables of length $\mathit{arity}(C)$.

Definition 69 (Matching) *The relation $\tau \in \pi$ holds if and only if $\mathit{true} \models \pi(\tau)$ holds.*

Definition 70 (Restriction) *The restriction*

$$\mathsf{restrict}(\forall\vartheta.\,\kappa \Rightarrow \theta_1 \to ... \to \theta_n \to \theta, \pi_1 ... \pi_n)$$

is equal to

$$\forall\vartheta t_1 ... t_n.\,\kappa \wedge t_i \leq \theta_i \wedge \pi_i(t_i) \Rightarrow t_1 \to ... \to t_n \to \theta$$

and $\mathsf{restrict}(\tau, \pi_1 ... \pi_n)$ *is the error type $\mathbb{E}$ if τ is not a functional type of textual arity n.*

Note that if $\mathsf{restrict}(\tau, \overline{\pi})$ is not well-formed it is equal to $\mathbb{E}$ by Definition 27 and the fact that $\mathbb{E}$ is maximal.

We can then prove that the Requirement 44 (Restriction) holds:

Lemma 71 (Restriction)

$$\frac{\tau'_i \in \pi_i \quad (i = 1..n)}{\mathsf{restrict}(\tau_m, \overline{\pi})\ \tau'_1 \ ...\ \tau'_n \equiv \tau_m\ \tau'_1 \ ...\ \tau'_n}$$

Proof of lemma 71 (Restriction)

Let τ_m be $\forall\vartheta.\,\kappa \Rightarrow \theta_1 \to ... \to \theta_n \to \theta$, and τ'_i be $\forall\vartheta'_i.\,\kappa'_i \Rightarrow \theta'_i$.
By Definition 70 and App:

$$\begin{array}{l} \mathsf{restrict}(\tau_m, \pi_i)\ \tau'_i = \forall\vartheta t_i \vartheta'_i.\,\kappa \wedge t_i \leq \theta_i \wedge \pi_i(t_i) \wedge \kappa'_i \wedge \theta'_i \leq t_i \Rightarrow \theta \\ \tau_m\ \tau'_1 \ ...\ \tau'_n = \forall\vartheta\vartheta'_i.\,\kappa \wedge \theta'_i \leq \theta_i \wedge \kappa'_i \Rightarrow \theta \end{array}$$

The proof of $\tau_m\ \tau'_1 \ ...\ \tau'_n \leq \mathsf{restrict}(\tau_m, \pi_i)\ \tau'_i$ is straightforward:

$\forall t.$
$\quad \kappa \wedge t_i \leq \theta_i \wedge \pi_i(t_i) \wedge \kappa'_i \wedge \theta'_i \leq t_i \wedge \theta \leq t$
$\models \quad \kappa \wedge \theta'_i \leq \theta_i \wedge \kappa'_i \wedge \theta \leq t \qquad$ (MTrans)

Conversely, $\mathsf{restrict}(\tau_m, \pi_i)\ \tau_i' \leq \tau_m\ \tau_1'\ ...\ \tau_n'$ holds because:

$$
\begin{array}{ll}
\forall t. & \\
\quad & \kappa \wedge \theta_i' \leq \theta_i \wedge \kappa_i' \wedge \theta \leq t \\
\models & \kappa \wedge \theta_i' \leq \theta_i \wedge \kappa_i' \wedge \pi_i(\theta_i') \wedge \theta_i' \leq \theta_i' \wedge \theta \leq t \quad (\tau_i' \in \pi_i) \\
\models & \kappa \wedge t_i \leq \theta_i \wedge \kappa_i' \wedge \pi_i(t_i) \wedge \theta_i' \leq t_i \wedge \theta \leq t \quad (\textsc{VarIntro} : t_i \mapsto \theta_i')
\end{array}
$$

■

Therefore, we can apply Theorem 49 (Generic functions), which shows that valid and covered multi-methods verify Requirement 9 (Constants). Using Theorem 65 (Modular type-checking), the type soundness of modular programs with valid and covered multi-methods is therefore guaranteed.

8.6.3 Examples

This section illustrates the implementation and typing aspects of multi-methods. Firstly, we consider the definition of a generic method `equals` for structural equality of values. We assume given a function `identical` for testing reference equality, to handle the general case, and a record type `Point` with a component x.

$$
\begin{array}{l}
\textbf{method}\ \texttt{equals} : \forall t.\ t \to t \to \texttt{bool} \\
\textbf{implementation}\ \texttt{equals}\ \_\ \_ = \texttt{identical} \\
\textbf{implementation}\ \texttt{equals}\ \texttt{@Point}\ \texttt{@Point} = \\
\quad \lambda p.\lambda q.\ \texttt{equals}\ p.x\ q.x
\end{array}
$$

The first implementation is valid since `identical` has type $\forall t.\ t \to t \to \texttt{bool}$. The second implementation is valid since it has inferred type $\texttt{Point} \to \texttt{Point} \to \texttt{bool}$ and calculation in the $\mathrm{ML}_{\leq}$ type algebra shows that:

$$
\begin{array}{l}
\mathsf{restrict}(\forall t.\ t \to t \to \texttt{bool}, \texttt{@Point}\ \texttt{@Point}) \\
\equiv \forall t, t_1, t_2.\, t_1 \leq t \wedge t_2 \leq t \wedge t_1 \leq \texttt{Point} \wedge t_2 \leq \texttt{Point} \Rightarrow t_1 \to t_2 \to \texttt{bool} \\
\equiv \forall t_1, t_2.\, t_1 \leq \texttt{Point} \wedge t_2 \leq \texttt{Point} \Rightarrow t_1 \to t_2 \to \texttt{bool} \\
\equiv \texttt{Point} \to \texttt{Point} \to \texttt{bool}
\end{array}
$$

We can now implement the `opp` example of Chapter 4 in the concrete case of multi-methods with:

$$
\begin{array}{l}
\textbf{method}\ \texttt{opp} : \forall t.\, t \leq \texttt{num} \Rightarrow t \to t \\
\textbf{implementation}\ \texttt{opp}\ \#\texttt{float} = \texttt{opp\_float} \\
\textbf{implementation}\ \texttt{opp}\ \#\texttt{int} = \texttt{opp\_int}
\end{array}
$$

This illustrates that # patterns are useful to implement methods with precise types like `opp`, which could not be implemented using @ patterns: if we try to use pattern `@float` we find:

$$\mathsf{restrict}(\forall t.\, t \leq \texttt{num} \Rightarrow t \to t, \texttt{@float}) \equiv \forall t.\, t \leq \texttt{float} \Rightarrow t \to t$$

It would thus be invalid to use pattern `@float` with a function that returns a `float`. That would be unsound indeed, since this implementation could be called with a strict subtype (say `rational`) of `float`, in which case `opp`'s type requires that the result be type `rational`. On the other hand,

$$\mathsf{restrict}(\forall t.\, t \leq \texttt{num} \Rightarrow t \to t, \#\texttt{float}) \equiv \texttt{float} \to \texttt{float}$$

The reason why the implementation with pattern `#float` is sound is that this pattern does not match the strict subtypes of `float`. The coverage test will therefore ensure that an implementation exists for any such subtype, and each of these implementations will be required to return a result of the same type.

The # patterns are also useful outside base type and operators. Consider the case of a generic container hierarchy. Container classes `List`, `Vector`, etc, derive from an abstract `Container` class. It is natural to

define functional operations on containers that return the same kind of container as their argument. These can be implemented using # patterns, and not by @ patterns:

method `map` : $\forall t, u, c \leq$ `Container`. $(c[t],\ t \to u) \to c[u]$
implementation `map` #`List` `_` = ...**new** `List`...
implementation `map` #`Vector` `_` = ...**new** `Vector`...

8.6.4 Semantics

The semantics is implied by the matching relation, as defined in Chapter 4. We can now follow up on the discussion about the fact that this semantics depends on types. With the patterns described in this section, only the head type constructor is needed to specify reductions. In an object-oriented language, this tagging information is already commonly present at run-time. It corresponds to the `new_C` data constructor of Chapter 3. Multi-methods can thus be implemented without run-time typing.

Furthermore, it is not fundamental to give an eager semantics. The patterns presented in this section only require the head type constructor to be known in order to choose the method implementation. Object fields could therefore be computed lazily. This is much similar to the semantics of pattern matching in lazy languages, which would make the introduction of such multi-methods fit well in lazy languages.

Algorithms for efficient reduction of multi-method operators (that is, implementation of multiple-dispatch) can be found in [17, 15, 20].

Chapter 9

Super in a modular setting

In Section 5, we formalized the semantics of **super** in a whole program. In particular, the definition of the target of a super call uses the set of implementations of a method. As soon as a program has more than one module, it is possible that this set is not completely known in the current module. It is therefore relevant that the target be resolved in the current module. We therefore start by formalizing the semantics of **super** in a modular setting.

9.1 Formalization

The **super** keyword defined in Section 5 was annotated by the method it relates to and by the patterns of that method's implementation in which it appears. We now additionally add an annotation for the module in which it appears. The definitions of its target and its semantics are then modified to specify that the set of implementation of the method is taken from that module's signature, as defined in Definition 61.

Definition 72 (Modular target of super) *Let m be a method, $\overline{\pi}$ be a list of patterns, $\mathcal{M}$ a module and R a module repository. Then the target of $\boldsymbol{super}_{m,\overline{\pi},\mathcal{M}}$ in R is*

$$\mathsf{target}(\boldsymbol{super}_{m,\overline{\pi},\mathcal{M}}) = \max\{\overline{\pi}' \Rightarrow e' \in \mathrm{gensig}(m,R) \mid \overline{\pi}' < \overline{\pi}\}$$

Again, if there is no implementation of m with patterns less precise than $\overline{\pi}$, or none that is a maximum, the max is undefined, and $\mathsf{target}(\mathbf{super}_{m,\overline{\pi},\mathcal{M}})$ is $\bot$. In that second case, the use of **super** is ambiguous, and therefore results in a typechecking error.

One can then define the semantics of **super**:

Definition 73 (Modular reduction of super) *Let m be a method, $\overline{\pi}$ be a list of patterns and $\mathcal{M}$ a module. If $\mathsf{target}(\boldsymbol{super}_{m,\overline{\pi},\mathcal{M}})$ is $\overline{\pi}' \Rightarrow e'$, then $\boldsymbol{super}_{m,\overline{\pi},\mathcal{M}} \longrightarrow e'$.*

The type given to $\mathbf{super}_{m,\overline{\pi},\mathcal{M}}$ is the restriction of the type of m to the patterns of the target implementation:

Definition 74 (Modular type of super) *Let m be a method, $\overline{\pi}$ be a list of patterns, and $\mathcal{M}$be a module. Then the type of $\boldsymbol{super}_{m,\overline{\pi},\mathcal{M}}$ is defined by:*

$$\mathsf{constant\text{-}type}(\boldsymbol{super}_{m,\overline{\pi},\mathcal{M}}) = \begin{cases} \mathbb{E} & \text{if } \mathsf{target}(\boldsymbol{super}_{m,\overline{\pi},\mathcal{M}}) = \bot \\ \mathsf{restrict}(\mathsf{constant\text{-}type}(m), \overline{\pi}') & \text{if } \mathsf{target}(\boldsymbol{super}_{m,\overline{\pi},\mathcal{M}}) = \overline{\pi}' \Rightarrow e' \end{cases}$$

Theorem 75 (Modular super) *For any well typed method m, module $\mathcal{M}$ and list of patterns $\overline{\pi}$, the operator $\boldsymbol{super}_{m,\overline{\pi},\mathcal{M}}$ verifies Requirement 9 (*Constants*).*

The proof is mostly identical to the non-modular version. The only difference lies in the fact that typing inequality guaranteed by the validity of the method implementation is known in the type algebra associated with the module declaring the implementation. One has to use the argument that the type algebra of the whole program is an extension of that type algebra.

Proof of theorem 75 (Modular super)

1. Since $\mathbf{super}_{m,\overline{\pi},\mathcal{M}}$ is well-typed by hypothesis, $\mathsf{target}(\mathbf{super}_{m,\overline{\pi},\mathcal{M}})$ is well defined by Definition 74. Let then $\overline{\pi}' \Rightarrow e'$ be $\mathsf{target}(\mathbf{super}_{m,\overline{\pi},\mathcal{M}})$. By Definition 73, $\mathbf{super}_{m,\overline{\pi},\mathcal{M}} \longrightarrow e'$.

2. Let $\overline{\pi}' \Rightarrow e'$ be the implementation $\mathsf{target}(\mathbf{super}_{m,\overline{\pi},\mathcal{M}})$. Let $\mathcal{M}_0$ the main module of the program. Let $(A_0, \leq_0)$ be the type algebra $A(\mathcal{M}_0)$ and $(A_1, \leq_1)$ be the type algebra $A(\mathcal{M})$.

   The only reduction rule for **super** is $\mathbf{super}_{m,\overline{\pi},\mathcal{M}} \longrightarrow e'$. In this case, we have by Definition 74 $\mathsf{type}(\mathbf{super}_{m,\overline{\pi},\mathcal{M}}) = \mathsf{restrict}(\mathsf{type}(m), \overline{\pi}')$ (**1**). Since m is well typed, the implementation $\overline{\pi}' \Rightarrow e'$ must be valid by Definition 46. That is, by Definition 45, $\mathsf{type}(e') \leq_1 \mathsf{restrict}(\mathsf{type}(m), \overline{\pi}')$. By Requirement 62 (MODULE IMPORT), $A(\mathcal{M}_0)$ is an extension of $A(\mathcal{M})$. Therefore, by Definition 54, $\mathsf{type}(e') \leq_0 \mathsf{restrict}(\mathsf{type}(m), \overline{\pi}')$. This shows, together with (1) that $\mathsf{type}(e') \leq_0 \mathsf{type}(\mathbf{super}_{m,\overline{\pi},\mathcal{M}})$ holds.

■

In practice, the choice to resolve the target in the signature of the occurring module only is convenient for the purpose of separate compilation, since the super call can be resolved to the specific target implementation which is known at the time the module is compiled. This choice also ensures that no ambiguity error for super calls arises when combining independent modules together, which could be the case if the target of the call was resolved taking into consideration the whole program.

9.2 Consequences of the precocity rule

It is interesting to note that many cases where the resolution of the target of a **super** operator would be different in the non-modular program are in fact ruled out by the precocity rule (Definition 66). That is, when an implementation is omitted although it could have been defined, but it is present in another module, as in the following example:

```
module M0 {
  class A
}
module M1 imports M0 {
  method m : A -> void
  implementation m @A = eA;
}
module M2 import M0 {
  class B extends A
}
module M1' imports M1,M2 {
  class C extends B
  implementation m @C = eC;   // super in eC refers to m @A
}
module M2' imports M1,M2 {
  implementation m @B = eB;
}

program imports M1',M2'
```

This situation is ruled out by Definition 66. The implementation of `m` for class `B` is incorrect in module `M2'` because of the precocity rule. Indeed, it could have been done in module `M1'`, where both `m` and `B` are visible.

Chapter 10

Modular kinds

In Chapter 6, we have motivated the introduction of kinding constraints, and presented a formalization that works in a closed-world setting. In this chapter, we argue that this formalization is not suited to modular typechecking and we propose a revised solution that supports modular programming.

10.1 The open world problem

Rules of Section 6.5 were designed with a closed-world assumption. They are indeed sound in a closed world. However, we now show with a simplistic example that using the same rules in an open-world would be unsound.

```
module M1 {
  interface
    kind K
    class A : K
    method f : <T : K> T->T
  implementation
    f @A = fun x -> (new A)
}

module M2 imports M1 {
  interface
    class B : K
    var b : B
  implementation
    let b = M1.f (new B)
}
```

That is, if we typecheck the module `M1` in its type structure with a single class `A` implementing `K` and then typecheck `M2` in the type structure induced by `M1` (since it is imported by `M2`) and `M2`, both modules typecheck successfully. However, taken as a whole, the program would fail. This shows that new rules have to be given for typechecking in an open world.

Indeed, in module `M1`, the implementation of `f @A` must be a subtype of $\mathsf{restrict}(\forall T : K.\ T \to T, @A)$, which is by definition $\forall X, T.\ X \leq A, X \leq T, T : K \Rightarrow X \to T$. Its denotation is $\{A \to A\}$ since the only class implementing the kind K is A. The type of `fun x -> (new A)` is $\forall U.\ U \to A$, so its denotation includes $A \to A$ and it is a correct implementation for `f @A`. In module `M2`, the type of `f` which is known from the interface of `M1` to be $\forall T.T : K \Rightarrow T \to T$ can now be instantiated to $B \to B$ since $B : K$. Therefore b has

indeed type B and the module `M2` is well typed. However, `M1.f (new B)` reduces to `(fun x -> (new A)) (new B)`, which reduces to `(new A)`, which is not of type B. Subject-reduction does not hold.

The problem obviously comes from the closed-world assumption: in `M1` we assumed that the only class implementing `K` is `A`, however that is not true in the client module `M2`. There are at least two possible places where an error could have been reported. As a first solution, the declaration of class `B` in module `M2` could be invalid. However, this requires knowledge of the implementation of module `M1`: if we had written `f @A = fun x -> x`, class `B` would raise no problem. Therefore, this first solution is non-modular, and we rule it out. Our solution is instead to take the open-world assumption. The implementation of `f` in module `M1` will then be declared incorrect on the ground that a subclass of `A` of kind `K` might be declared later. Formally, this amounts to the fact that $\forall U.\ U \to A$ is not a subtype of $\forall X, T.\ X \leq A, X \leq T, T : K \Rightarrow X \to T$.

Is it still possible to implement method `f` at all? The identity function is a correct implementation. However, it might indeed be the case that a new object must be returned. This is possible using our `#` pattern. Had module `M1` implemented f with `f #A = fun x -> (new A)`, this implementation would indeed have a subtype of $\mathsf{restrict}(\forall T : K.\ T \to T, \#A)$, which is by definition $\forall X, T.\ X = A, X \leq T, T : K \Rightarrow X \to T$, that is $\forall T.\ A \leq T, T : K \Rightarrow A \to T$. This implementation would be sound because it is only applicable to instances of class `A`. The coverage test would require that an implementation be provided for `B`. The use of `#` patterns gives the flexibility of the first solution. It retains modularity by using only information from the patterns of the method implementations contained in imported modules, which have to be present in module signatures in order to be able to implement the coverage test of multi-methods.

However, it might be tedious to use `#` patterns in certain cases, as it prevents to share the same implementation for several classes. For instance, it might be known by design that no subclass of `A` can be of kind `K`. In that case, it would be useful to be able to declare this fact, and consequently allow to implement `f` homogeneously for `A` and all its subclasses, using the `@A` pattern. To support this situation, we introduce the possibility to add *abstracts* annotations: class `C` *abstracts* kind `K` if no subclass of `C` is allowed to have kind `K`. This is especially useful to reason in an open-world setting, since it tells us about all possible subclasses of `C`, even those that might be declared in other, unknown modules. In our previous example, class `BDD` abstracts `BooleanAlgebra`, and class `Integer` abstracts `Num`.

Let us consider how to use this information for typechecking. We reuse module `M1` from the previous example, adding `class A abstracts K` so that the implementation of `f` becomes legal. The restricted type of `f` to `@A` is, by definition, $\forall X, T.\ X \leq A, X \leq T, T : K \Rightarrow X \to T$. We would like its constraint to imply $A \leq T$. Consequently, $\forall U.\ U \to A$ would be a subtype of the restricted type, and the implementation would typecheck. Note that this can be done safely only if we know that no subclass of `A` will ever have kind `K`, which is exactly what the declaration `class A abstracts K` ensures. Otherwise, an object `c` of such a class `C` could be passed to `f`, and `f(c)` would have static type `C` (by instantiating `T` to `C`), while this expression would reduce to `new A` of type `A`, thus breaking subject-reduction.

For each kind `K`, we will define the view of class `C` for kind `K`, and write it $\mathsf{view}_{\mathtt{K}}(\mathtt{C})$. The class $\mathsf{view}_{\mathtt{K}}(\mathtt{C})$, if it exists, must a class to which type `C` is mapped when viewed as an instance of kind `K`. In particular, $\mathsf{view}_{\mathtt{K}}(\mathtt{C})$ is always a superclass of `C` and always implements kind `K`. In that case, and under the assumptions $X \leq C, X \leq T, T : K$, we want to be able to conclude that $\mathsf{view}_{\mathtt{K}}(\mathtt{C}) \leq T$. We formally define views in the next section, and show that this conclusion is indeed valid. In our example, $\mathsf{view}_{\mathtt{K}}(\mathtt{A})$ is `A` itself and the implementation of `f` typechecks.

A fruitful way to look at views is that they define, for each kind, an abstract view of the class hierarchy. For instance, kind `Num` defines a view of the numerical classes where `Int32` and `BigInt` are mapped to `Integer`. This view is more abstract than the original hierarchy because it hides the details of the implementation of `Integer`. Note that $\mathsf{view}_{\mathtt{Num}}(\mathtt{Float})$ does not exist, since `Float` does not abstract `Num`. This leaves the possibility to extend the hierarchy with a new subclass of `Float` that implements `Num`.

10.2 Open-world formalization

In this section we formally present an extension of $\mathrm{ML}_{\leq}$ with kinds. We use the framework introduced in Chapter 7 to show that this extension leads to a modular type algebra.

As in Section 6.5, type structures include a set of kinds, a partial order on kinds and predicates for implementation and abstraction of kinds by classes, and constraints include the implementation of a kind by a type constructor.

Type structure	$\mathcal{T}$	::=	$(\mathcal{C}, \mathcal{K}, \leq, :, ::)$
Constraint	κ	::=	$\theta \leq \theta \mid \phi_V \leq \phi_V \mid \phi_V : K$

A kind is a name in $\mathcal{K}$ that represents a characteristic shared by a set of classes. In a sense, *kinds are the type of classes* (technically speaking, of type constructors).

When a class `C` has the characteristic of a kind `K`, we will say that it *implements* this kind, and write `C` : `K`. If a class `C` *abstracts* a kind `K`, written `C` :: `K`, no subclass of `C` is allowed to implement that kind. While in many cases classes that abstract a kind also implement it, there is no technical reason to enforce this property.

It is also possible to express that a kind `K'` *extends* kind `K`, written `K'` $\leq$ `K`. This states that whenever a class implements `K'`, it also implements `K`. If we interpret views as an abstraction relation, `K'` is then more abstract than `K`. We assume the existence of a kind called `Top` that is a super-kind of all other kinds, implemented by every class and abstracted by every *final* class. [1]

We impose three requirements on type structures to ensure the coherence of kind annotations:

Requirement 76 (Kinds) *A type structure* $(\mathcal{C}, \mathcal{K}, \leq, :, ::)$ *is valid if:*

i. *for all kinds* `K` *and* `K'` *such that* `K'` $\leq$ `K`*, for all class* `C`*, if* `C` : `K'` *then* `C` : `K`*.*

ii. *for all kinds* `K` *and* `K'` *such that* `K'` $\leq$ `K`*, for all class* `C`*, if* `C` :: `K` *then* `C` :: `K'`*.*

iii. *for all kind* `K`*, for all classes* `C` *and* `C'` *such that* `C'` $\leq$ `C`*, if* `C` :: `K` *then* `C'` :: `K`*.*

The first requirement simply expresses the notion of kind extension. The second requirement is the dual of the first one. Since no subclass of `C` can implement `K` (`C` :: `K`), it follows from the first requirement that no subclass of `C` can implement `K'`. The third requirement saturates the abstracts annotations: since no subclass of `C` can implement `K` (`C` :: `K`), this is in particular true for any subclass `C'` of `C`. Therefore, we might as well require `C'` to abstract `K`. Note that a concrete programming language would therefore not require abstracts annotations to be repeated on all subclasses, but would infer the annotations automatically.

We can now define views.

Definition 77 (View) *If a class* `C` *abstracts kind* `K` *and if there exists a class* `C'` *such that, for all class* `C''` *below* `C`, `C'` *is the lowest class above* `C''` *implementing* `K`*, then the view of class* `C` *for kind* `K`*, written* $\mathsf{view}_{\mathtt{K}}(\mathtt{C})$*, is the class* `C'`*. Otherwise,* $\mathsf{view}_{\mathtt{K}}(\mathtt{C}) = \bot$*.*

A particular case concerns final classes. When `C` is a final class, then $\mathsf{view}_{\mathtt{K}}(\mathtt{C})$ is the least class above it that implements `K`. The intuition here is that since `C` is final, it cannot have sub-classes, so there is no difficulty to satisfy the downward-closing property of $\xrightarrow{\mathtt{K}}$. For instance, with a *final* class `NativeFloat` below `Float`, we would have `NativeFloat` $\xrightarrow{\mathtt{Num}}$ `Float`.

Definition 78 (Type structure extension) *A type structure* $\mathcal{T}'$ *extends a type structure* $\mathcal{T}$ *if the following conditions are respected:*

1. *the conditions of Definition 58*
2. *if* $c_V : K$ *in* $\mathcal{T}$*, then* $c_V : K$ *in* $\mathcal{T}'$
3. *if* $c_V :: K$ *in* $\mathcal{T}$*, then* $c_V :: K$ *in* $\mathcal{T}'$

[1] `Top` is special in the sense that we do not want the user to have to assert the above properties, but is has no special treatment in the theory. It can thus been seen as syntactic sugar.

4. *if* $K \leq K'$ *in* $\mathcal{T}$*, then* $K \leq K'$ *in* $\mathcal{T}'$

5. *if* $\mathsf{view}_K(c_V) = c'_V$ *in* $\mathcal{T}$ *with* $c'_V \neq \bot$*, then* $\mathsf{view}_K(c_V) = c'_V$ *in* $\mathcal{T}'$

We now introduce new axioms to deal with kinds.

Definition 79 (Constraint implication with kinds) *The constraint implication relation* $\models$ *is the least relation that satisfies the axioms of Figure 10.1 and of Figure 2.5.*

$$\textsc{TrivImp}\ \frac{c_V : K}{\forall\vartheta.\ \kappa \models \kappa \wedge c_V : K} \qquad \textsc{Abs}\ \frac{\phi_V \leq c_V \in \kappa \qquad \phi_V \leq \phi'_V \in \kappa \qquad \phi'_V : K \in \kappa}{\forall\vartheta.\ \kappa \models \kappa \wedge \mathsf{view}_K(c_V) \leq \phi'_V}\ \mathsf{view}_K(c_V) \neq \bot$$

$$\textsc{EqImp}\ \frac{\phi_V = \phi'_V \in \kappa \qquad \phi_V : K \in \kappa}{\forall\vartheta.\ \kappa \models \kappa \wedge \phi'_V : K} \qquad \textsc{ExtImp}\ \frac{\phi_V : K' \in \kappa \qquad K' \leq K}{\forall\vartheta.\ \kappa \models \kappa \wedge \phi_V : K}$$

Figure 10.1: Axioms for kinds

The main axiom, Abs, defines how views augment the constraint implication relation. When a class c_V has a view for kind K, and there exists a type constructor ϕ_V below both c_V and a type constructor ϕ'_V that implements K, then we can conlude that $\mathsf{view}_K(c_V)$ is below ϕ'_V. This axiom is correct because of the minimality condition in Definition 77.

Using the framework of Chapter 7, we just need to prove the correction and completeness of this axiomatization, as defined by Definition 55. By Theorem 57 (Extension of a ML-Sub type algebra), this ensures that extensions of the type structure lead to extensions of the type algebra.

Theorem 80 (Correction and completeness of the axiomatization of kinds) *For all type structure* $\mathcal{T}$*, variable list* ϑ *and constraints* κ_1 *and* κ_2 *in* $\mathcal{T}$*, the constraint implication* $\forall\vartheta.\ \kappa_1 \models \kappa_2$ *holds if and only if for all extension* $\mathcal{T}'$ *of* $\mathcal{T}$*, for all ground substitution* σ_1 *such that* $\mathcal{T}' \vdash \sigma_1(\kappa_1)$*, there exists a ground substitution* σ_2 *such that* $\sigma_2 \overset{\vartheta}{=} \sigma_1$ *and* $\mathcal{T}' \vdash \sigma_2(\kappa_2)$*.*

Proof of theorem 80 (Correction and completeness of the axiomatization of kinds)
We first consider soundness, that is, the "if" part of the proposition. The original proof of [5] is by case on the axioms. Since we only strengthened the requirements on type structures, the proofs for the existing axioms carry on unchanged. We thus only need to prove that the newly introduced axioms are also sound.

For all axioms, we choose $\sigma_2 = \sigma_1$, and we write it σ. By Definition 78, all the hypotheses on the type structure $\mathcal{T}$ are also valid on $\mathcal{T}'$.

case TrivImp
By hypothesis, $\sigma(\kappa)$ holds in $\mathcal{T}'$. Furthermore, $c_V : K$ in $\mathcal{T}$, and thus also in $\mathcal{T}'$ by Definition 78. Therefore, $\sigma(\kappa \wedge c_V : K)$ holds in $\mathcal{T}'$.

case Abs
Since by hypothesis $\mathsf{view}_K(c_V) \neq \bot$ in $\mathcal{T}$, $\mathsf{view}_K(c_V)$ has the same value in $\mathcal{T}'$ as in $\mathcal{T}'$ by Definition 78. By hypothesis, class $\sigma(\phi'_V)$ verifies $\sigma(\phi'_V) \geq \sigma(\phi_V)$, and $\sigma(\phi_V) \leq c_V$. By Definition 77, $\mathsf{view}_K(c_V)$ is the least such class in $\mathcal{T}'$. Therefore $\mathsf{view}_K(c_V) \leq \sigma(\phi'_V)$.

case EqImp
This is trivial, since $\leq$ is a partial order over the type constructors of $\mathcal{T}'$.

case ExtImp
$K' \leq K$ holds in $\mathcal{T}'$ by Definition 78. By hypothesis, $\sigma(\phi_V) : K'$. So by Requirement 76.**i**, $\sigma(\phi_V) : K$.

We now prove completeness, that is, the "only if" part of the proposition. As in [5], we first consider the case where κ_1 is a well-formed constraint in base form, that is without constructed monotypes. In that case, we can construct a specific extension of the type structure $\mathcal{T}^1$, which assigns a constant to every variable in ϑ. We shall prove that $\mathcal{T}^1$ extends $\mathcal{T}$ and that for all ϑ-closed constraint κ, the constraint implication $\forall\vartheta.\ \kappa_1 \models \kappa$ holds if and only if κ holds in $\mathcal{T}^1$. This corresponds to lemma 18 in [5]. The rest of the proof is then unchanged, from their lemma 19 to 21, which proves completeness.

Let us define $\mathcal{T}^1$. Let V^1 be the set of V type constructors in $\mathcal{T}$ and of V type constructors variables in ϑ. Let T^1 be the set of type variables in ϑ. We define partial orders $\leq$ and equivalence relations $=$ on T^1 and all V^1 by:

$$\begin{array}{rcl} t \leq t' & \Leftrightarrow & \forall\vartheta.\ \kappa_1 \models t \leq t' \\ \phi_V \leq \phi'_V & \Leftrightarrow & \forall\vartheta.\ \kappa_1 \models \phi_V \leq \phi'_V \\ t = t' & \Leftrightarrow & t \leq t' \text{ and } t' \leq t \\ \phi_V = \phi'_V & \Leftrightarrow & \phi_V \leq \phi'_V \text{ and } \phi'_V \leq \phi_V \end{array}$$

We build the type structure $\mathcal{T}^1$ by adding to $\mathcal{T}$ a constant type constructor for each equivalence class of V^1 and T^1. Noting $[\phi_V]$ the class of ϕ_V, we define $\mathcal{T}^1$ with:

$$\begin{array}{rcl} [\phi_V] : K \text{ in } \mathcal{T}^1 & \Leftrightarrow & \forall\vartheta.\ \kappa_1 \models \phi_V : K \\ [\phi_V] :: K \text{ in } \mathcal{T}^1 & \Leftrightarrow & \text{there exists } c_V \text{ in } \mathcal{T} \text{ such that } \forall\vartheta.\ \kappa_1 \models \phi_V \leq c_V \text{ and } c_V :: K \text{ hold in } \mathcal{T} \\ K' \leq K \text{ in } \mathcal{T}^1 & \Leftrightarrow & K' \leq K \text{ in } \mathcal{T} \end{array}$$

These definitions do not depend on the choice of ϕ_V in its equivalence class thanks to axioms EqImp and CTrans.

Let us check that $\mathcal{T}^1$ is a type structure. By definition, the relations $\leq$ are partial orders. For Requirement 76 (Kinds):

i. By hypothesis, $K' \leq K$ and $[\phi_V] : K'$ hold in $\mathcal{T}^1$. That is, by definition of $\mathcal{T}^1$, $K' \leq K$ and $\forall\vartheta.\ \kappa_1 \models \phi_V : K'$ hold in $\mathcal{T}$. Therefore, by ExtImp, $\forall\vartheta.\ \kappa_1 \models \phi_V : K$ hold in $\mathcal{T}$. That is, by definition of $\mathcal{T}^1$, $[\phi_V] : K$ holds in $\mathcal{T}^1$.

ii. By hypothesis, $K' \leq K$ and $[\phi_V] :: K$ hold in $\mathcal{T}^1$. That is, by definition of $\mathcal{T}^1$, $K' \leq K$ and there exists a class c_V in $[\phi_V]$ such that $c_V :: K$ hold in $\mathcal{T}$. Since $\mathcal{T}$is a type structure, by Requirement 76 (Kinds), $c_V :: K'$ holds in $\mathcal{T}$. Therefore, by definition of $\mathcal{T}^1$, $[\phi_V] :: K'$ holds in $\mathcal{T}^1$.

iii. By hypothesis, $[\phi'_V] \leq [\phi_V]$ and $[\phi_V] :: K$ hold in $\mathcal{T}^1$. That is, by definition of $\mathcal{T}^1$, $\forall\vartheta.\ \kappa_1 \models \phi'_V \leq \phi_V$ holds in $\mathcal{T}$(**1**), and there exists a class c_V such that $\forall\vartheta.\ \kappa_1 \models \phi_V \leq c_V$ (**2**) and $c_V :: K$ hold in $\mathcal{T}$(**3**). By Lemma 26 (Conjunction) with (1) and (2), $\forall\vartheta.\ \kappa_1 \models \phi'_V \leq \phi_V \wedge \phi_V \leq c_V$ holds in $\mathcal{T}$. Therefore, by CTrans and Triv, $\forall\vartheta.\ \kappa_1 \models \phi'_V \leq c_V$ holds in $\mathcal{T}$. Therefore, by (3) and definition of $\mathcal{T}^1$, $[\phi_V] :: K'$ holds in $\mathcal{T}^1$.

Let us now check that $\mathcal{T}^1$ is an extension of $\mathcal{T}$, up to the [_] quotient, as defined in Definition 78.

1. True by CStruct.

2. If $c_V : K$ in $\mathcal{T}$, then by TrivImp $\forall\vartheta.\ \kappa_1 \models c_V : K$ holds in $\mathcal{T}$. Therefore, $[c_V] : K$ holds in $\mathcal{T}^1$.

3. By definition of abstraction in $\mathcal{T}^1$, since c_V belongs to $[c_V]$

4. By definition of $\mathcal{T}^1$

5. By hypothesis, $\mathsf{view}_K(c_V)$ exists and is c'_V in $\mathcal{T}$. Therefore, by Definition 77, c_V abstracts K. That is, $[c_V]$ abstracts K in $\mathcal{T}^1$. Moreover, for all $[\phi_V]$ such that $[\phi_V] : K$ and that there exists $[\phi'_V]$ such that $[\phi'_V] \leq [\phi_V]$ and $[\phi'_V] \leq [c_V]$, we have to prove that $[c'_V] \leq [\phi_V]$. We have by hypothesis $\forall\vartheta.\ \kappa_1 \models \phi'_V \leq \phi_V : K \wedge \phi'_V \leq c_V$. Therefore, by Abs, $\forall\vartheta.\ \kappa_1 \models c'_V \leq \phi_V$. That is, $[c'_V] \leq [\phi_V]$ in $\mathcal{T}^1$. Therefore, $\mathsf{view}_K([c_V])$ is $[c'_V]$ in $\mathcal{T}^1$.

It now remains to prove that for all ϑ-closed constraint κ, κ holds in $\mathcal{T}^1$ if and only if $\forall\vartheta.\ \kappa_1 \models \kappa$ holds in $\mathcal{T}$. First, suppose $\forall\vartheta.\ \kappa_1 \models \kappa$. By construction, κ_1 holds in in $\mathcal{T}^1$. Furthermore, we have proved that $\mathcal{T}^1$ is an extension of $\mathcal{T}$. So we can apply the soundness property, which proves, since κ is ϑ-closed, that κ holds in $\mathcal{T}^1$. For the converse proof, we proceed by induction on the structure of κ.

case $\kappa = \phi_V \leq \phi'_V$
By hypothesis, κ holds in $\mathcal{T}^1$. That is, $[\phi_V] \leq [\phi'_V]$. Thus by definition of $\leq$ in $\mathcal{T}^1$, $\forall\vartheta.\ \kappa_1 \models \phi_V \leq \phi'_V$.

case $\kappa = \theta \leq \theta'$
By hypothesis, $\theta \leq \theta'$ in $\mathcal{T}^1$. So by Definition 23, $\theta = \phi_V[\overline{\theta}]$ and $\theta' = \phi'_V[\overline{\theta'}]$, with $\phi_V \leq \phi'_V$ and $\overline{\theta} \leq_V \overline{\theta'}$ holding in $\mathcal{T}^1$. So by definition of $\leq$ in $\mathcal{T}^1$, $\forall\vartheta.\ \kappa_1 \models \phi_V \leq \phi'_V$, and by induction hypothesis and Lemma 26 (Conjunction), $\forall\vartheta.\ \kappa_1 \models \overline{\theta} \leq_V \overline{\theta'}$. So, again by Lemma 26 (Conjunction), $\forall\vartheta.\ \kappa_1 \models \phi_V \leq \phi'_V \wedge \overline{\theta} \leq_V \overline{\theta'}$. Therefore, by MIntro, $\forall\vartheta.\ \kappa_1 \models \theta \leq \theta'$.

case $\kappa = \phi_V : K$
By hypothesis, $\phi_V : K$ holds in $\mathcal{T}^1$. So by definition of $\mathcal{T}^1$, $\forall\vartheta.\ \kappa_1 \models \phi_V : K$.

case $\kappa = \kappa'_1 \wedge \kappa'_2$
By induction hypothesis, $\forall\vartheta.\ \kappa_1 \models \kappa'_1$ and $\forall\vartheta.\ \kappa_1 \models \kappa'_2$. So by Lemma 26 (Conjunction), $\forall\vartheta.\ \kappa_1 \models \kappa'_1 \wedge \kappa'_2$.

■

The extension of $\mathrm{ML}_\leq$ with kinds has been used as a type system of our programming language Nice. This has been very useful to spot interesting typing situations and check how they can be solved using kinds. A note on syntactic details is given in Appendix 13 to enable the reader to experiment with our implementation. We could also verify that type-checking can be implemented efficiently.

10.3 Language

In this section, we briefly describe a complete programming language that supports kinds. We base our presentation on the generic framework of Chapter 1, thus illustrating its interest for factoring a large part of the presentation and the proofs. This framework is extensible in two directions. First, an arbitrary type algebra — a language for types equipped with a subtyping relation — can be used, provided it meets four simple requirements. Second, new operators can be defined to add features to the language. In particular, multi-methods can be defined as operators.

For the type algebra, we take the extended version of $\mathrm{ML}_\leq$, as defined in Section 10.2. In particular, type constraints include kinding constraints. For operators, we can simply reuse the multi-methods defined in Chapter 8. Their expressivity is automatically augmented by the possibility to include kinding constraints in their types. Additionally, the surface language needs to include the possibility to declare new kinds, and to declare that a class implements an existing kind. These declarations have no evaluation semantics, but create the type structure in which subtyping is defined. The syntax for programs with multi-methods and kinding constraints is:

Declaration	$\mathcal{G}$	::=	**generic** g : τ
Implementation	$\mathcal{I}$	::=	**implementation** g $\overline{\pi} \Rightarrow e$
Class	$\mathcal{C}$	::=	**class** C **extends** $\overline{C}$ { ... }
Kind	$\mathcal{K}$	::=	**kind** K
Kinding	$\mathcal{KI}$	::=	**class** C **implements** K
Module	$\mathcal{M}$	::=	**module** M **imports** $\overline{M}$; **let rec** $\overline{\mathcal{G} \mid \mathcal{I} \mid \mathcal{C} \mid \mathcal{K} \mid \mathcal{KI}}$
Program	$\mathcal{P}$	::=	**imports** M **eval** e

Interestingly, there is no need to add specific rules to check the kind implementation declarations. They come as a particular case of multi-method typechecking and coverage test: if `class C implements K` and

method `f` has type `<T : K> T -> T`, then the coverage test will check that there exists an implementation of `f` that matches class `C`. Additionally, each implementation will be forced to be type-correct.

Since Theorem 80 (Correction and completeness of the axiomatization of kinds) holds, the modular typechecking scheme of Chapter 8 is sound for these programs.

10.4 Conclusion

In Chapter 6, we have identified the need to augment the expressiveness of type systems with polymorphism and nominal subtyping to handle two typing situations that occur in practice. Our solution is to introduce kinds that describe a property that types can declare to possess.

In this chapter, we have proposed an extension of the $ML_{\leq}$ type system that implements that solution while preserving the main properties of the system. The resulting system achieves modularity, since it allows modules to be type-checked independently and new classes to be added in a hierarchy containing partially polymorphic methods. We have implemented this type system in the Nice programming language, showing in particular that type-checking remains tractable.

Part IV

Practice

Chapter 11

Code generation

11.1 Monomorphic bytecode language

We define in this chapter a target language for compiling our high-level language with classes and multi-methods. One goal is to show how our language can be compiled to a low-level stack language that is explicitly typed and includes the verification of code before running it. This is useful, since such low-level languages are becoming common, in particular with Java bytecode [31] and the Common Language Infrastructure.

However, many features of these languages are not necessary for this presentation. For instance, since their dispatch is limited to one argument, it will be necessary to express multiple dispatch with more primitive operations. Therefore, we can ignore single dispatch for the sake of simplicity. We choose to formalize a subset of the Java bytecode. This choice makes the presentation more practical by allowing compilation to an existing and widespread architecture. Furthermore, it stays valid for similar targets that include the same subset of features that we use.

Bytecode type	T	::=	
Class name			C
Array		|	$T[]$
Variable name	x		
Method name	m		
Bytecode expression	B	::=	
			`load` x
		|	`store` x `in` B
		|	`call` m
		|	`cast` T
		|	`iftrue` B `else` B
		|	`instanceof` C
		|	`exactinstanceof` C
		|	`true` | `false`
		|	`new` C
		|	`field` $C.i$
Sequence		|	$B; B$
Bytecode value	V	::=	`true` | `false` | $\overline{V}$; `new` C
Function definition	F	::=	`static method` $m(\overline{T\ x}) : T\ \{B;$ `return`$\}$
Class definition	D	::=	`class` C `extends` $C'\ \{\ \overline{T field};\ \}$

Figure 11.1: Bytecode

The syntax of bytecode types, bytecode expressions and function definitions is given in Figure 11.1. Since all methods are static, only one type of call is used, which corresponds to `invokestatic` in the JVM. Therefore, we simply call it `call`. The fact that `store` has an explicit scope enables us to express more simply a reduction semantics. This scope corresponds to the lifetime associated to local variables in the JVM.

The objects that are instances of a class C are represented by a sequence of values for each field followed by the operator `new` C. This corresponds to pushing the values of the fields on the stack before calling the `new` C operator.

The bytecode expression `exactinstanceof` C is not part of the JVM. However, it can be emulated easily, for instance by using the instructions to retrieve the class of the value, then to get the name of that class and to compare it with the name of class C. This corresponds to the bytecode

`call Object.getClass; call Class.getName; nameOfC; call Object.equals`

where `nameOfC` is the fully qualified name of C.

We denote by C the types of the bytecode language. They correspond exactly to class names. They are ordered by the subtype ordering, which is declared by the heritage relation on classes.

$$\texttt{true}; \texttt{iftrue}\ B_1\ \texttt{else}\ B_2 \longrightarrow B_1 \qquad \texttt{false}; \texttt{iftrue}\ B_1\ \texttt{else}\ B_2 \longrightarrow B_2$$

$$\frac{m \text{ declared with } \texttt{static method}\ m(T_1\ x_1, ..., T_n\ x_n) : T\ \{B; \texttt{return}\}}{V_1; ...; V_n; \texttt{call}\ m \longrightarrow V_1; ...V_n; \texttt{store}\ x_n\ \texttt{in}\ ...\ \texttt{store}\ x_1\ \texttt{in}\ B}$$

$$V; \texttt{store}\ x\ \texttt{in}\ B \longrightarrow B\,[(\texttt{load}\ x) \leftarrow V] \qquad \frac{\text{class } C' \text{ extends } C}{\overline{V}; \texttt{new}\ C';\ \texttt{cast}\ C \longrightarrow \overline{V}; \texttt{new}\ C'}$$

$$\frac{\text{class } C' \text{ extends } C}{\overline{V}; \texttt{new}\ C';\ \texttt{instanceof}\ C \longrightarrow \texttt{true}} \qquad \frac{\text{class } C' \text{ does not extend } C}{\overline{V}; \texttt{new}\ C';\ \texttt{instanceof}\ C \longrightarrow \texttt{false}}$$

$$\overline{V}; \texttt{new}\ C;\ \texttt{exactinstanceof}\ C \longrightarrow \texttt{true} \qquad \frac{C' \neq C}{\overline{V}; \texttt{new}\ C';\ \texttt{exactinstanceof}\ C \longrightarrow \texttt{false}}$$

$$V_1; ...; V_n; \texttt{new}\ C'; \texttt{field}\ C.i \longrightarrow V_{\text{shift}(sc(C'),C)+i} \qquad \frac{B_1 \longrightarrow B_1'}{B_1; B_2 \longrightarrow B_1'; B_2}$$

Figure 11.2: Bytecode semantics

A reduction semantics on these expressions is given in Figure 11.2. We do not try to model side effects. The generalization to references is indeed orthogonal to our focus, which is the implementation of multi-methods and the translation of polymorphic code into verifiable monomorphic code. Therefore, we do not need an evaluation environment. Furthermore, we suppose for the sake of simplicity that every expression variable has a different name, in order to avoid explicit renamings.

The first reduction expresses that a true value followed by an `iftrue` operator reduces to the first branch. A function call reduces, when arguments are evaluated to values, to the expression storing the values in the formal parameters and evaluating the body of the function. Store expressions reduce by substituting the value for instances of loads for the corresponding variable in the body of the store expression. A `cast` C expression reduces provided that the instance is built on a subclass of C. Since we are interested in compiling type-safe programs to bytecode, we are only interested in showing that casts never fail. In that setting, it is sufficient to let the unsuccessful case be stuck, since we don't model exceptions. The `instanceof` and `exactinstanceof` expressions reduce accordingly to the class upon which the value is built. The reduction

of field accessed requires the computation of the rank difference of the field between the declaring super-class and the class the value is an instance of. This rank difference is defined as in Section 3.2 using the shift operator. The list of fields declared by a class, $Fields(\_)$, as well as the full list of fields, $AllFields(\_)$, are also used in this section. Finally, sequence expressions reduce if their prefix reduce.

11.1.1 Type checking

The bytecode is submitted to type-checking before being executed. Figure 11.4 defines the $\Gamma; S \vdash B : S'$ relation, which infers the bytecode types S' on the stack after evaluation of a bytecode expression B, given a type environment Γ and the types S on the stack before the evaluation.

$$C \leq C \qquad \frac{\texttt{class } C \texttt{ extends } C'}{C \leq C'} \qquad \frac{C \leq C' \qquad C' \leq C''}{C \leq C''} \qquad \frac{T \leq T'}{T[] \leq T'[]}$$

Figure 11.3: Subtyping

$$\Gamma, x : T; S \vdash \texttt{load } x : S, T \qquad \frac{\Gamma, x : T; S \vdash B : S'}{\Gamma; S, T \vdash \texttt{store } x \texttt{ in } B : S'}$$

$$\frac{m \text{ declared with } \texttt{static method } m(T_1\ x_1, ..., T_n\ x_n) : T\ \{...\} \qquad \forall i \in 1..n\ T'_i \leq T_i}{\Gamma; S, T'_1, ..., T'_n \vdash \texttt{call } m : S, T}$$

$$\Gamma; S, T' \vdash \texttt{cast } T : S, T \qquad \frac{\Gamma; S \vdash B_1 : S, T_1 \qquad \Gamma; S \vdash B_2 : S, T_2 \qquad (T_2 \leq T_1 = T) \text{ or } (T_1 \leq T_2 = T)}{\Gamma; S, \texttt{boolean} \vdash \texttt{iftrue } B_1 \texttt{ else } B_2 : S, T}$$

$$\Gamma; S, T \vdash \texttt{instanceof } C : S, \texttt{boolean} \qquad \Gamma; S, T \vdash \texttt{exactinstanceof } C : S, \texttt{boolean}$$

$$\Gamma; S \vdash \texttt{true} : S, \texttt{boolean} \qquad \Gamma; S \vdash \texttt{false} : S, \texttt{boolean} \qquad \frac{AllFields(C) = T_1, ..., T_n \qquad \forall i\ T'_i \leq T_i}{\Gamma; S, T'_1, ..., T'_n \vdash \texttt{new } C : S, C}$$

$$\frac{C' \leq C \qquad Fields(C) = T_1, ..., T_n}{\Gamma; S, C' \vdash \texttt{field } C.i : S, T_i} \qquad \frac{\Gamma; S \vdash B_1 : S' \qquad \Gamma; S' \vdash B_2 : S''}{\Gamma; S \vdash B_1; B_2 : S''}$$

Figure 11.4: Type checking

A `load` expression pushes the type of the loaded expression on the type stack. The expression `store` x `in` B produces the same type stack as the expression B in the context where x has the top type of the incoming stack. A function call pops the types of the arguments and pushes the return type, provided that the argument types are subtypes of the declared parameter types. The expression `cast` T changes the top type to T. An `iftrue` expression requires the top type to be `boolean`, and pushes the type produced by either branch provided it is greater than the one of the other branch. `instanceof` and `exactinstanceof` pop an arbitrary type and push the `boolean` type. `true` and `false` push the `boolean` type. The expression `new` C pushes type C provided the types on top of the stack are subtypes of the corresponding field types. A field access expression `field` $C.i$ pops a subtype C' of C and pushes the type of the i^{th} field of class C. Finally, a sequence $B_1; B_2$ produces the same stack as B_2 produces with the incoming stack being the one produced by B_1.

11.2 Monomorphic instances of polytypes

The type system defined in the previous section is very limited compared to $ML_{\leq}$. In particular, it does not include type constructors, functional types and polymorphic constrained types. $ML_{\leq}$ types must therefore be translated into bytecode types, in parallel with the translation of high-level expressions into bytecode expressions. This translation must verify the following constraints:

- the generated bytecode must have the same semantics as the source program;
- the generated bytecode must be well-typed according to the bytecode type system;
- as far as possible, the translation must allow fast execution of the generated bytecode.

The second point is debatable. One could also choose not to respect the bytecode type system and rely on the possibility of some virtual machines to switch off bytecode verification. Supposing the source program was well-typed, the first point still ensures that no error will occur at runtime. However, the user of a released compiled program would not have any guarantees that the program is safe. Furthermore, virtual machines can suppose that the executed bytecode is verifiable. It is therefore possible that they perform some optimizations that become incorrect on non-verifiable bytecode. Therefore, it seems that this approach would require a modified bytecode format, together with a modified virtual machine to execute it. In that case, the format could as well include other features, such as multiple dispatch. While this approach is also interesting, we do not explore it here.

In this section, we motivate our choices for the translation of $ML_{\leq}$ types into bytecode types, before formalizing that translation in Section 11.3. We first consider types build by the application of type constructors to other types, before tackling the more difficult case of polymorphic constrained types.

11.2.1 Type Constructors

Parameterized classes

The translation of a parameterized class definition is done by erasure, as in Pizza [37]: a non-parameterized bytecode class is defined, whose fields have the bytecode translation of their type.

The translation of a type $C[\overline{\theta}]$ constructed on a parameterized class C is simply C.

Functional types

Lambda-abstractions are compiled into objects that contain the captured variables of the environment and a final method with a canonical name (apply) that represents the function. A functional type is therefore translated into a bytecode interface type that declares a method `apply` with one argument for each argument of the lambda-abstraction.

At first sight, one could want to declare in this interface type the bytecode type of the arguments and the return type of the function. This would allow the bytecode of the lambda-expression and of its calls to use a better approximation of their type, therefore avoiding in some cases to need a cast. However, this would make impossible the direct use of a functional expression as parameter of a higher order function, as soon as the bytecode type of this expression is not exactly the one expected. It would in particular be the case for polymorphic functions and for functions whose domain is larger, or whose codomain is smaller, than the expected type. These cases are valid in $ML_{\leq}$ but would not be in the bytecode, since it has only invariance rules for the type of methods in subclasses. For instance, given types $\texttt{C} \leq \texttt{B} \leq \texttt{A}$, type $\texttt{B} \to \texttt{B}$ would be translated into an interface containing the method `B apply(B)`, and a value of type $\texttt{A} \to \texttt{C}$ would consist of an instance of a class with a method `C apply(A)`. However, that latter class cannot be made to implement the interface. Therefore, in the bytecode, the value cannot be used directly where type $\texttt{B} \to \texttt{B}$ is expected, even though that is valid in the source language.

One could consider adding a cast towards the expected functional interface type. This cast would only succeed if the class of the functional value explicitly declares implementing this interface. This is impractical:

for a value of type $\forall T.\ T \to T$, this would amount to implementing the functional interface type of every type occurring in the program. Besides the sheer number of those, this would pose a problem for separate compilation.

Another solution is to insert problematic functional value inside another one declaring the expected functional interface type, redirecting calls to the original function and cast the result to return the expected type. This solution has a runtime cost of one closure creation for every functional value passed as an argument with a different type, plus one indirection and a cast per call of this functional value.

A simpler solution for functional types is to use only one interface per arity, every parameter and return type being `Object`. The runtime cost is of one cast for each argument and for the return type per call, except when used polymorphically. Compared to the previous solution, this one is advantageous if lambda-expressions are often polymorphic and have typically few arguments. Furthermore, it avoids the creation of numerous functional interface types, each one incurring a cost at its first use in the virtual machine. Furthermore, it is ideal in the case where casts can be deactivated when the bytecode is trusted, since typing guarantees that they will not fail.

Only this last solution has been implemented. A comparison of the two approaches on concrete examples would be necessary to conclusively decide which one is most efficient.

Arrays

Arrays are the only parameterized types in Java bytecode. They are covariant, which imposes a runtime verification during writes inside arrays, possibly failing with an `ArrayStoreException`. In Nice, arrays are invariant. Typing is therefore more restrictive (although polymorphism allows to express naturally polymorphic functions on arrays, provided they respect their types). By translating a $\mathrm{ML}_{\leq}$ array type to the bytecode array type whose elements are the translation of the type parameter, the bytecode type system is therefore automatically respected. Furthermore, generated programs never lead to runtime errors while writing arrays elements because of the type of the element.

11.2.2 Constrained polymorphic types

Indeed, those types represent in general a set of monomorphic type instances that are equivalent to no single monomorphic type. The translation is therefore only an approximation. Our goal is to find an optimal monomorphic approximation for any polytype.

Example

Let f be a function of type $\forall T \leq A.\ T \to T$, where A is a class that has a subclass B. We need to find a bytecode type for the domain D and the codomain C of f. The choice of those two types is constrained both by the bytecode implementation of the function and by the call sites of f.

Two requirements have to be met, since they originate from reasoning on the runtime types of values. Ignoring them would lead to a possible cast error at runtime. Firstly, the values returned by f might be instances of both A and B. The bytecode codomain must therefore verify $C \geq A$ (and $C \geq B$, but this last requirement is weaker than the previous one). Secondly, the argument passed during a call to f can be an instance of A or B, which constraints the domain with $D \geq A$.

Two other contexts show what bytecode type is expected, and therefore allow to know what precise value to choose for C and D to minimize the number of casts: the code of f is typed under the hypothesis $T \leq A$, and T is the type of the parameter of f. Therefore, all the operations done on this parameter must be valid for $T = A$. If we choose $D = A$, we therefore guarantee that no cast is necessary in the code generated for f. On the other hand, the value returned by a call to f can in general be used with type T, instantiated for this particular call, either in A or in B. Since the choice of the codomain is constrained by $C \geq A$, we cannot avoid a cast for the case $T = B$. On the other hand, by choosing $C = A$ rather than $C =$ `Object`, we avoid a cast in the case $T = A$.

This example suggests that the optimal valid translation for a polymorphic function is obtained when choosing the greatest instance of the type parameters. We shall now formalize and prove this rule.

11.3 Compilation

11.3.1 Types

A closed monotype is translated by erasure into a bytecode type. Only array types are parameterized by the type of their components.

Definition 81 (Erasure for monomorphic types) *Given a ground monotype θ, its erasure $BC(\theta)$ is defined by:*

$$\begin{array}{lrcl} \textit{Array type} & BC(\texttt{Array}[\theta]) & = & BC(\theta)[] \\ \textit{Constructed monotype} & BC(c_V[\overline{m}]) & = & BC(c_V) \quad (c_V \neq \texttt{Array}) \\ \textit{Function type} & BC(\,\theta_1 \to \theta_2) & = & \texttt{Fun} \end{array}$$

where $c_V \mapsto BC(c_V)$ is a one-to-one mapping from type constructors to bytecode class names, and `Fun` *is a class with a single method* `Object apply(Object)`.

Lemma 82 (Covariance of the bytecode translation) *Let θ and θ' be two ground monotypes such that $\theta \leq \theta'$. Then $BC(\theta) \leq BC(\theta')$.*

Proof of lemma 82 (Covariance of the bytecode translation)

If either θ or θ' are function types, then both are since $\theta \leq \theta'$ holds. Therefore, both of their translations are `Fun`, and the result holds.

Otherwise, let θ be $c_V[\overline{\theta}]$ and θ' be $c'_V[\overline{\theta}']$. By Definition 23, since $\theta \leq \theta'$, we have $c_V \leq c'_V$ and $\overline{\theta} \leq_V \overline{\theta'}$.

case $c'_V = \texttt{Array}$
Since `Array` is a final class and since $c_V \leq$ `Array`, we also have $c_V =$ `Array`. Furthermore, `Array` has signature $V = (\otimes)$ (that is, it is an invariant type constructor), so $\overline{\theta} = \overline{\theta'} = \theta_0$. Thus, $BC(\theta) = BC(\theta') = BC(\theta_0)[]$.

case c'_V is a class different from `Array`
Since $c_V \leq c'_V$, c_V is also a class different from `Array`. Therefore, $BC(\theta) = c_V$ and $BC(\theta') = c'_V$. Furthermore, the type constructors verify $c_V \leq c'_V$, so the bytecode classes also verify $BC(c_V) \leq BC(c'_V)$.

■

The translation of polymorphic type is done by instantiation into the most general monotype. For method types, the domain and the codomain are translated independently.

Definition 83 (Erasure for polymorphic types)

$$\begin{array}{rcl} BC_\kappa(\theta) & = & \mathsf{mub}\ \{BC(\sigma(\theta)) \mid \sigma(\kappa)\} \\ BC(\forall \overline{t}.\, \kappa \Rightarrow \theta) & = & BC_\kappa(\theta) \\ dom_i(\forall \overline{t}.\, \kappa \Rightarrow \theta_1 \to ... \to \theta_n \to \theta) & = & BC_\kappa(\theta_i) \\ codom(\forall \overline{t}.\, \kappa \Rightarrow \theta_1 \to ... \to \theta_n \to \theta) & = & BC_\kappa(\theta) \end{array}$$

where mub *is a function mapping a set of bytecode types to one of their minimal upper bounds.*

For a set of bytecode types, `Object` is always an upper bound. With multiple inheritance, there can be several minimal upper bounds. For our purposes, it does not matter which one is chosen, since any upper bound would be correct. The minimality just helps keeping the bytecode type as informative as possible, and potentially reduces the number of casts needed inside the implementation of methods.

This computation is stable by extension of the set of types, so it is compatible with separate compilation. Indeed, the new bytecode types are always smaller than the existing ones, since only subclasses of existing classes can be added. Thus, by Lemma 82 (Covariance of the bytecode translation), they do not modify the result of the computation of the minimal upper bound.

11.3.2 Programs

The goal of this section is to define a translation $BC(\cdot)$ from source programs of Chapter 4 into bytecode that verifies the following theorem. We use a call-by-value semantics.

Theorem 84 (Compilation) *Let p be* ***let rec*** $\overline{\mathcal{D}}$ ***in*** e *and $BC(p)$ be its bytecode translation. If p is well-typed, then $BC(p)$ is a well-formed bytecode program. Furthermore, let v be a value such that $e \longrightarrow v$ and $\tau_v = \mathsf{type}(v)$. Then $BC(e) \longrightarrow BC(v)$ and the bytecode value $BC(v)$ has bytecode type $BC(\tau_v)$.*

Definition 85 (Bytecode translation) *Given an expression e, we define its translation $BC(e)$ by case on e. For a variable x, its translation is* `load` x.

Given the expression `let` x_1 `be` e_1 `in` e_2, *let* $\forall\vartheta_1.\kappa_1 \Rightarrow \theta_1$ *be* $\mathsf{type}(e_1)$ *and* $\forall\vartheta_2.\kappa_2 \Rightarrow \theta_2$ *be* $\mathsf{type}(\texttt{let } x_1 \texttt{ be } e_1 \texttt{ in } e_2)$. *Then* $BC(\texttt{let } x_1 \texttt{ be } e_1 \texttt{ in } e_2)$ *is defined as* $BC(e_1);\texttt{cast } BC_{\kappa_2}(\theta_1);\texttt{store } x_1 \texttt{ in } BC(e_2)$.

A lambda-abstraction $\lambda x.e$ is translated into an object with a single method: `new Fun(){ Object apply(Object x) {` $BC(e)$ `}}`.

For an application of the form $c\ e_1\ ...\ e_n$, where c is an operator of arity n, its translation is $\overline{BC(e_i);\texttt{cast } dom_i(\mathsf{type}(c))};BC_{Call}(c)$, where

$$\begin{aligned} BC_{Call}(m) &= \texttt{call } m \\ BC_{Call}(\textbf{\textit{new}}\, C) &= \texttt{new } C \\ BC_{Call}(C.i) &= \texttt{field } C.i \end{aligned}$$

Functional constants that are not directly applied to all of their arguments are compiled in their eta-expansed form. Let n be arity(c) and p be a number, with $0 \leq p < n$. Then $BC(c\ e_1\ ...\ e_p)$ is equal to $BC(\lambda x_{p+1}...x_n.c\ e_1...e_p, x_{p+1}...x_n)$.

An application $e_1\ \ e_2$ where e_1 is not a constant is translated into $BC(e_1);\texttt{cast Fun};BC(e_2);\texttt{call Fun.apply}$.

Finally, $BC($`true`*$)$ is* `true` *and $BC($*`false`*$)$ is* `false`.

For a method definition `method` $m : \tau\ \overline{(\pi_{i,1},...,\pi_{i,n}) \Rightarrow \lambda x_1...x_n.e_i}$, *its bytecode translation is*

```
static method m(dom_1(τ) x_1, ..., dom_n(τ) x_n) : codom(τ)
{
   BC_{π_{1,1}}(load x_1); BC_{π_{1,2}}(load x_2); and; ...; BC_{π_{1,n}}(load x_n); and; iftrue BC(e_1)
   else ...
   else BC_{π_{p-1,1}}(load x_1); BC_{π_{p-1,2}}(load x_2); and; ...; BC_{π_{p-1,n}}(load x_n); and; iftrue BC(e_{p-1})
   else BC(e_p);
   cast codom(τ);return
}
```

where

$$\begin{aligned} BC_{@C}(B) &= B;\texttt{instanceof } C \\ BC_{\#C}(B) &= B;\texttt{exactinstanceof } C \\ BC_{@\_}(B) &= \texttt{true} \end{aligned}$$

For a class declaration `class` $C[\overline{t}]$ `extends` $C_1,...,C_m\ \{f_1 : F_1(\overline{t}),...,f_p : F_p(\overline{t})\}$, *its bytecode translation is*

$$\texttt{class } C \texttt{ extends } C_1,...,C_m\ \{\ BC_{true}(F_1(\overline{t}))\ f_1;...;BC_{true}(F_p(\overline{t}))\ f_p;\ \}$$

For compiling methods, we suppose that the implementations are ordered with respect to the specificity of patterns. That is, such that for all numbers i and j, $i < j \Longrightarrow \pi_j \not\leq \pi_i$. This is always possible, since the source program is well-typed, while two equal patterns would lead to an ambiguity error.

For let expressions, we cast the bound value to be of type $BC_{\kappa_2}(\theta_1)$. It would be correct to cast it to be of type $BC_{\kappa_1}(\theta_1)$ instead. However, since κ_2 is the constraint for the whole expression, it is a superset of κ_1, as can be checked in Definition 28. Therefore, $BC_{\kappa_1}(\theta_1)$ is a more precise (that is, smaller) bytecode type, which takes into account the way the bound variable is used in the body of the let expression. By giving a more precise bytecode type to x_1, this potentially reduces the number of casts needed in the translated body. At the same time, this initial cast is guaranteed to succeed since the whole source expression is well-typed, as we show in the proof of Theorem 84 (Compilation).

We can observe that many casts can be erased, as soon as the bytecode typing ensures that the expression on the stack has a subtype of the desired type. This happens often in practice[1], both in fully monomorphic code and in fully polymorphic code. Casts are only necessary at the border between these two kinds of code, that is, when a value with a known monomorphic type is passed to a polymorphic function and the result of the call is used with its statically known monomorphic type.

In our translation, casts are performed before method calls and before storing the value of a let-bound variable. It would be possible to place them differently. For instance, one could cast the result of method calls and of loads, which would guarantee that the types on the stack are always as precise as they can be. The translation we chose has two advantages. Firstly, it easily guarantees that bytecode is well-typed, since a cast is inserted if necessary before each checked instruction, that is method calls and field accesses. Secondly, this translation reduces the number of casts that are needed. For instance, if the result of a method call is not used, no cast is performed. On the other hand, if this result is bound by a let to be used several times, the cast is made before the binding, to avoid a possible cast at each use.

We assign methods a return type which is the monomorphic approximation of their codomain. This requires in general a cast before the `return` instruction since the implementation of the method can use polymorphically typed functions, which entails a loss of typing information. Alternatively, one could give all methods a return type of `Object` (and thus avoid the cast) and rely on the other casts to be performed as needed. This would reduce the number of casts when the result is not used with a specific type, but would increase it if the cast inside the method was redundant and the approximation was sufficient for the way the result was used. Furthermore, our solution has the advantage of giving methods a more intuitive type, which is useful when looking at the generated code or using it in a call from a different language using the same bytecode.

For classes, the translation of field types is, given Definition 83, a minimal upper bound of the translation of all possible instantiations for the class type parameters. This implies that the type parameters, if they appear at all in the translated type, are translated to the bytecode type `Object`.

Lemma 86 (Bytecode pattern test) *Let v be a well-typed value and π be a pattern. Then $v \in \pi$ holds if and only if $BC_\pi(BC(v)) \longrightarrow^*$ `true` holds.*

Proof of lemma 86 (Bytecode pattern test)
The proof is by case on π. Let $\mathsf{type}(v)$ be $\forall\vartheta.\,\kappa \Rightarrow \theta$.

case $\pi = @C$

Then $v \in \pi$ amounts to $\mathit{true} \models \kappa \wedge \theta \leq C[\overline{t}]$. Therefore, by Requirement 38 (Class type), $v =$ **new** $C'\ v_1\ ...\ v_n$, where C' is a subclass of C (**1**). Thus, $BC(v)$ is $BC(v_1);...;BC(v_n);$`new` C'. Thus, $BC_\pi(BC(v))$ is $BC(v_1);...;BC(v_n);$`new` $C';$`instanceof` C. So, by the semantics of Figure 11.2 and (1), $BC_\pi(BC(v)) \longrightarrow$ `true`.

Conversely, if $BC(v);$`instanceof` $C \longrightarrow$ `true`, then by Figure 11.2 $BC(v) = V_1;...;V_n;$`new` C' where C' is a subclass of C. Therefore v is of the form **new** $C'\ v_1\ ...\ v_n$. By definition, $\mathsf{constant\text{-}type}($**new** $C') = \forall\overline{t}.\,F'_1(\overline{t}) \to ... \to F'_n(\overline{t}) \to C'[\overline{t}]$ where $[F'_1, ..., F'_n] = \mathrm{AllFields}(C')$. So by App, θ is of the form $C'[\overline{t'}]$. Let V be the variance of C and C' and $\overline{t}$ be a list of $\mathit{arity}(V)$ fresh type variables. Then,

[1] In the source code of the compiler for Nice, from 90 to 95% of the casts theoretically needed are unnecessary for this reason.

$$
\begin{array}{lll}
 & \mathit{true} & \\
\models & \kappa & (v \text{ is well-typed}) \\
\models & \kappa \wedge C' \leq C & (C' \text{ subclass of } C) \\
\models & \kappa \wedge C' \leq C \wedge \overline{t'} \leq_V \overline{t'} & (\textsc{MRef}) \\
\models & \kappa \wedge C' \leq C \wedge \overline{t'} \leq_V \overline{t} & (\textsc{VarIntro} \text{ with } \sigma(\overline{t}) = \overline{t'}) \\
\models & \kappa \wedge C'[\overline{t'}] \leq C[\overline{t}] & (\textsc{MIntro})
\end{array}
$$

That is to say that $v \in @C$ holds.

case $\pi = \#C$

Then $v \in \pi$ amounts to $\mathit{true} \models \kappa \wedge \theta = C[\overline{t}]$, and therefore in particular $\mathit{true} \models \kappa \wedge \theta \leq C[\overline{t}]$. So by Requirement 38 (Class type), $v = \textbf{new}\ C'\ v_1\ ...\ v_n$, where C' is a subclass of C. Since $\mathit{true} \models \kappa \wedge \theta = C[\overline{t}]$, C' is actually equal to C. So $BC(v)$ is equal to $BC(v_1); ...; BC(v_n); \texttt{new}\ C$. Furthermore, $BC_\pi(BC(v)) = BC(v_1); ...; BC(v_n); \texttt{new}\ C; \texttt{exactinstanceof}\ C$. So, by the semantics of Figure 11.2, $BC_\pi(BC(v)) \longrightarrow \texttt{true}$.

Conversely, if $BC(v); \texttt{exactinstanceof}\ C \longrightarrow \texttt{true}$, then by Figure 11.2 $BC(v) = V_1; ...; V_n; \texttt{new}\ C$. So v is of the form $\textbf{new}\ C\ v_1\ ...\ v_n$. By definition, $\mathsf{constant\text{-}type}(\textbf{new}\ C) = \forall \overline{t}.\, F_1(\overline{t}) \to ... \to F_n(\overline{t}) \to C[\overline{t}]$ where $[F_1, ..., F_n] = \text{AllFields}(C)$. So by App, θ is of the form $C[\overline{t}]$. Let V be the variance of C and $\overline{t'}$ be a list of $\mathit{arity}(V)$ fresh type variables. Then,

$$
\begin{array}{lll}
 & \mathit{true} & \\
\models & \kappa & (v \text{ is well-typed}) \\
\models & \kappa \wedge C = C & (\textsc{CRef}) \\
\models & \kappa \wedge C = C \wedge \overline{t} =_V \overline{t} & (\textsc{Ref}) \\
\models & \kappa \wedge C = C \wedge \overline{t} =_V \overline{t'} & (\textsc{VarIntro} \text{ with } \sigma(\overline{t'}) = \overline{t}) \\
\models & \kappa \wedge C[\overline{t}] = C[\overline{t'}] & (\textsc{MIntro})
\end{array}
$$

That is to say that $v \in \#C$ holds.

case $\pi = \_$

Then by definition $BC_\pi(BC(v)) = \texttt{true}$.

Conversely, $\mathit{true} \models \mathit{true}$, therefore $v \in \_$.

■

We prove special properties about values. Firstly, we show that the type of a value is special in that the bytecode translation of all its instances are the same bytecode type.

Lemma 87 (Value types) *Let v be a value of type $\tau_v = \forall \vartheta_v.\, \kappa_v \Rightarrow \theta_v$. Then for all ground substitution σ, $BC(\sigma(\theta_v))$ is equal to $BC(\tau_v)$.*

Proof of lemma 87 (Value types)

It is sufficient to show that the type $BC(\sigma(\theta_v))$ does not depend on σ. By Definition 83, this will show that that type is $BC(\tau_v)$. The proof is by case on v.

case $v = \texttt{true}, v = \texttt{false}$

Then τ_v is `Boolean`. Therefore, for all substitution σ, $BC(\sigma(\theta_v))$ is `Boolean`.

case $v = \textbf{new}\ C\ v_1\ ...\ v_n$

Then τ_v is $\forall t_1, ..., t_n.\, \kappa_1 \wedge \theta_1 \leq t_1, ..., \kappa_n \wedge \theta_n \leq t_n \Rightarrow C[t1, ..., t_n]$. Therefore, for all substitution σ, $BC(\sigma(\theta_v))$ is $BC(C)$.

case $v = \lambda x.e$

By Definition 28, for any σ, $\sigma(\theta_v)$ is a functional type. Therefore, by Definition 81 and Definition 83, $BC(\sigma(\theta_v))$ is the bytecode type `Fun`.

■

We now show that in a well-typed application of a function to values, the bytecode translation of the type of the values is a subtype of the translated domain of the function. This implies that no cast will be needed to translate the application itself, which will be used to show in Lemma 89 (VALUES) that the translation of a value is a bytecode value.

Lemma 88 (Bytecode application) *Let $e\ v_1\ ...\ v_n$ be a well-typed expression. Let τ by the type of e and, for all i from 1 to n, τ_{v_i} be the type of v_i. Then, for all i from 1 to n, $BC(\tau_{v_i})$ is a subtype of $dom_i(\tau)$*

Proof of lemma 88 (Bytecode application)

Let $\forall\vartheta.\,\kappa \Rightarrow \theta_1 \to ... \to \theta_n \to \theta'$ be τ, and $\forall\vartheta_v.\,\kappa_{v_i} \Rightarrow \theta_{v_i}$ be τ_{v_i}. Since $e\ v_1\ ...\ v_n$ is well-typed, the constraint $\kappa \wedge \overline{\kappa_{v_i} \wedge \theta_{v_i} \leq \theta_i}$ is satisfiable. Let therefore σ_0 be a ground substitution such that $\sigma_0(\kappa \wedge \overline{\kappa_{v_i} \wedge \theta_{v_i} \leq \theta_i})$ holds (**2**). By Lemma 87 and (2), $BC(\tau_{v_i})$ is equal to $BC(\sigma_0(\theta_{v_i}))$. Furthermore, by (2), $\sigma_0(\theta_{v_i}) \leq \sigma_0(\theta_i)$ holds. Therefore, by Lemma 82 (COVARIANCE OF THE BYTECODE TRANSLATION), $BC(\sigma_0(\theta_{v_i})) \leq BC(\sigma_0(\theta_i))$ holds. By Definition 83 and since σ_0 satisfies κ by (2), $BC(\sigma_0(\theta_i))$ must be smaller than the upper bound $BC_\kappa(\theta_i)$, which is also $dom_i(\tau)$. That is, by transitivity, $BC(\tau_{v_i})$ is a subtype of $dom_i(\tau)$. ■

We now show two further properties of values. Firstly, the translation of a source value is a bytecode value. Secondly, subtyping on polytypes must be preserved by translation, if the smaller type can be the type of a value. This property is required so that a polymorphic value can be used directly as an expression of a less general type. This corresponds in particular to the observation made in Section 11.2.1 about functional types. For instance, this property would be violated if $\forall T.\ T \to T$ was translated into object with a method of type `Object` → `Object` and `String` → `String` was translated into an object with a method of type `String` → `String`.

Lemma 89 (Values) *Let v be a well-typed value of type τ_v. Then $BC(v)$ is a bytecode value whose bytecode type is $BC(\tau_v)$.*

Furthermore, for all type τ such that $\tau_v \leq \tau$, $BC(\tau_v)$ is smaller than $BC(\tau)$.

Proof of lemma 89 (Values)

The proof is by induction and case on v.

case $v = \texttt{true}, v = \texttt{false}$

Then $\mathsf{type}(v) = \texttt{Boolean}$, and $BC(v)$ is by definition a bytecode value of type `boolean` $= BC(\texttt{Boolean})$.

Furthermore, let τ such that $\texttt{Boolean} \leq \tau$. By corollary 56 (INTERPRETATION), every instance θ of τ verifies $\texttt{Boolean} \leq \theta$, so by Lemma 82, $BC(\texttt{Boolean}) \leq BC(\theta)$. Therefore, $BC(\texttt{Boolean}) \leq BC(\tau)$.

case $v = \textbf{new}\ C\ v_1\ ...\ v_n$

By induction hypothesis, each $BC(v_i)$ has bytecode type $BC(\mathsf{type}(v_i))$. Since v is well-typed, Lemma 88 (BYTECODE APPLICATION) shows that $BC(\mathsf{type}(v_i))$ is a subtype of $\mathrm{dom}_i(\tau)$. Therefore, in Definition 85 for $BC(v)$, the casts are redundant, and $BC(v)$ is equal to $BC(v_1); ...; BC(v_n); \texttt{new}\ C$, which is a bytecode value of bytecode type C.

By definition, $\mathsf{constant\text{-}type}(\textbf{new}\ C) = \forall\overline{t}.\,F_1(\overline{t}) \to ... \to F_n(\overline{t}) \to C[\overline{t}]$ where $[F_1, ..., F_n] = \mathrm{AllFields}(C)$. Let $\mathsf{type}(v_i)$ be $\forall\vartheta_{v_i}.\,\kappa_{v_i} \Rightarrow \theta_{v_i}$. Then by APP, $\tau_v = \mathsf{type}(\textbf{new}\ C\ v_1\ ...\ v_n) = \forall\overline{t\vartheta_{v_i}}.\,\overline{\kappa_{v_i} \wedge \theta_{v_i} \leq F_i(\overline{t})} \Rightarrow C[\overline{t}]$. Therefore, by Definition 81 and Definition 83, $BC(\tau_v) = C$.

Let $\tau_0 = \forall\vartheta_0.\,\kappa_0 \Rightarrow \theta_0$ be a type such that $\tau_v \leq \tau_0$. Let κ' be $\overline{\kappa_{v_i} \wedge \theta_{v_i} \leq F_i(\overline{t})}$. By corollary 56 (INTERPRETATION), for every ground substitution σ such that $\sigma(\kappa_0)$, there exists σ' such that $\sigma'(\kappa')$ and $\sigma'(C[\overline{t}]) \leq \sigma(\theta_0)$. So by Lemma 82, $BC(\sigma'(C[\overline{t}])) \leq BC(\sigma(\theta_0))$. Furthermore, $BC(\sigma'(C[\overline{t}]))$ is always equal to C by Definition 81. So $C \leq BC(\sigma(\theta_0))$ holds for all σ. Thus, by Definition 83, $BC(\tau_v) = C \leq BC(\tau_0)$.

case $v = \lambda x.e$

By Definition 28, Definition 81 and Definition 83, $BC(\tau_v)$ is `Fun`. By Definition 85, $BC(v)$ has bytecode type `Fun` as well.

Furthermore, for all type τ such that $\tau_v \leq \tau$, τ is also a functional type by MELIM, and therefore $BC(\tau) = BC(\tau_v) = \texttt{Fun}$.

■

The following lemma shows that the bytecode translation is a morphism for substitution.

Lemma 90 (Bytecode substitution) *For all expression e, value v and variable x, $BC(e\,[x \leftarrow v]) = BC(e)\,[\texttt{load}\ x \leftarrow BC(v)]$ holds.*

Note that the validity of this lemma lies in the fact that x is bound by a let and therefore is not modified in $BC(e)$.

Proof of lemma 90 (Bytecode substitution)

The proof is by induction on e.

case $e = x$

Then $BC(e)\,[\texttt{load}\ x \leftarrow BC(v)] = (\texttt{load}\ x)\,[\texttt{load}\ x \leftarrow BC(v)] = BC(v)$ and $BC(e\,[x \leftarrow v]) = BC(v)$.

case $e = x'$ **with** $x' \neq x$

Then $BC(e)\,[\texttt{load}\ x \leftarrow BC(v)] = (\texttt{load}\ x')\,[\texttt{load}\ x \leftarrow BC(v)] = \texttt{load}\ x'$ and $BC(e\,[x \leftarrow v]) = BC(x') = \texttt{load}\ x'$.

case $e = c\ e'_1\ ...\ e'_n$ **with** $arity(c) = n$

Let τ be $\mathsf{type}(c)$.

$$
\begin{array}{lll}
 & BC(e)\,[\texttt{load}\ x \leftarrow BC(v)] & \\
= & \overline{BC(e'_i)\,[\texttt{load}\ x \leftarrow BC(v)]; \texttt{cast}\ dom_i(\tau)}; BC_{Call}(c) & \text{(Definition 85)} \\
= & \overline{BC(e'_i\,[x \leftarrow v]); \texttt{cast}\ dom_i(\tau)}; BC_{Call}(c) & \text{(Induction hypothesis)} \\
= & BC(c\ (e'_1\,[x \leftarrow v])\ ...\ (e'_n\,[x \leftarrow v])) & \text{(Definition 85)} \\
= & BC((c\ e'_1\ ...\ e'_n)\,[x \leftarrow v]) &
\end{array}
$$

case $e = e_1\ e_2$ **with** e **not of the form** $c\ e'_1\ ...\ e'_n$

$$
\begin{array}{lll}
 & BC((e_1\ e_2)\,[\texttt{load}\ x \leftarrow BC(v)] & \\
= & (BC(e_1); \texttt{cast Fun}; BC(e_2); \texttt{call Fun.apply})\,[\texttt{load}\ x \leftarrow BC(v)] & \text{(Definition 85)} \\
= & BC(e_1)\,[\texttt{load}\ x \leftarrow BC(v)]; \texttt{cast Fun}; BC(e_2)\,[\texttt{load}\ x \leftarrow BC(v)]; \texttt{call Fun.apply} & \\
= & BC(e_1\,[x \leftarrow v]); \texttt{cast Fun}; BC(e_2\,[x \leftarrow v]); \texttt{call Fun.apply} & \text{(Ind. hyp.)} \\
= & BC(e_1\,[x \leftarrow v]\ e_2\,[x \leftarrow v]) & \text{(Definition 85)} \\
= & BC((e_1\ e_2)\,[x \leftarrow v]) &
\end{array}
$$

case $e = \texttt{let}\ x'\ \texttt{be}\ e_1\ \texttt{in}\ e_2$

$$
\begin{array}{lll}
 & BC(e)\,[\texttt{load}\ x \leftarrow BC(v)] & \\
= & (BC(e_1); \texttt{cast}\ BC_{\kappa_2}(\theta_1); \texttt{store}\ x_1\ \texttt{in}\ BC(e_2))\,[\texttt{load}\ x \leftarrow BC(v)] & \text{(Definition 85)} \\
= & BC(e_1)\,[\texttt{load}\ x \leftarrow BC(v)]; \texttt{cast}\ BC_{\kappa_2}(\theta_1); & \\
 & \texttt{store}\ x_1\ \texttt{in}\ BC(e_2)\,[\texttt{load}\ x \leftarrow BC(v)] & \\
= & BC(e_1\,[x \leftarrow v]); \texttt{cast}\ BC_{\kappa_2}(\theta_1); \texttt{store}\ x_1\ \texttt{in}\ BC(e_2)\,[x \leftarrow v] & \text{(Induction hypothesis)} \\
= & BC(\texttt{let}\ x'\ \texttt{be}\ e_1\,[x \leftarrow v]\ \texttt{in}\ e_2\,[x \leftarrow v]) & \text{(Definition 85)} \\
= & BC((\texttt{let}\ x'\ \texttt{be}\ e_1\ \texttt{in}\ e_2)\,[x \leftarrow v]) &
\end{array}
$$

case $e = \lambda x'.e'$

By alpha-conversion, we can assume that x' is different from x (**1**).

$$
\begin{array}{lll}
& BC(e)\,[\texttt{load}\ x \leftarrow BC(v)] & \\
= & (\texttt{new Fun()\{ Object apply(Object x') \{ } BC(e')\ \texttt{\}\})}\,[\texttt{load}\ x \leftarrow BC(v)] & \text{(Definition 85)} \\
= & \texttt{new Fun()\{ Object apply(Object x') \{ } BC(e')\,[\texttt{load}\ x \leftarrow BC(v)]\ \texttt{\}\})} & \\
= & \texttt{new Fun()\{ Object apply(Object x') \{ } BC(e'\,[x \leftarrow v])\ \texttt{\}\})} & \text{(Induction hypothesis)} \\
= & BC(\lambda x'.(e'\,[x \leftarrow v])) & \text{(Definition 85)} \\
= & BC((\lambda x'.e')\,[x \leftarrow v]) & (1)
\end{array}
$$

■

We can now prove our main result.

Proof of theorem 84 (Compilation)

It is sufficient to show that $BC(e) \longrightarrow BC(v)$. Indeed, by Lemma 89 (Values), we then have that $BC(v)$ has type $BC(\mathsf{type}(v))$.

The proof is induction on the length of the reduction, and by case on e.

case $e = v$

By Lemma 89, $BC(v)$ is a bytecode value, so $BC(v) \longrightarrow BC(v)$.

case $e = c\ e'_1\ ...\ e'_n$ **with** $arity(c) = n$

Since e is well-typed, all e'_i are also well-typed by property **i** of Definition 4 (Error). Therefore, we know by subject reduction that, for all i in $1..n$, there exists a value v_i such that $e'_i \longrightarrow v_i$. Let τ be $\mathsf{type}(c)$, and τ_{v_i} be $\mathsf{type}(v_i)$.

By induction hypothesis, we know that $BC(e'_i) \longrightarrow BC(v_i)$. Therefore,

$$
\begin{array}{lll}
& BC(c\ e'_1\ ...\ e'_n) & \\
= & BC(e'_1); \texttt{cast}\ dom_1(\tau); ...; BC(e'_n); \texttt{cast}\ dom_n(\tau); BC_{Call}(c) & \\
\longrightarrow & BC(v_1); \texttt{cast}\ dom_1(\tau); ...; BC(v_n); \texttt{cast}\ dom_n(\tau); BC_{Call}(c) &
\end{array}
$$

Since $c\ v_1\ ...\ v_n$ is well-typed by Theorem 10 (Subject reduction), we can apply Lemma 88 (Bytecode application), which shows that, for all i, $BC(\tau_{v_i})$ is a subtype of $dom_i(\tau)$ (**1**). Therefore, all the cast succeed, and $BC(v_1); \texttt{cast}\ dom_1(\tau); ...; BC(v_n); \texttt{cast}\ dom_n(\tau); BC_{Call}(c) \longrightarrow BC(v_1); ...; BC(v_n); BC_{Call}(c)$.

We now reason by case on c.

case $c = \textbf{new}\ C$

Then $e \longrightarrow v' = \textbf{new}\ C\ v_1\ ...\ v_n$.

By definition, $BC(v_1); ...; BC(v_n); BC_{Call}(c) = BC(v_1); ...; BC(v_n); \texttt{new}\ C$. This expression is equal to $BC(\textbf{new}\ C\ v_1\ ...\ v_n)$, since by (1) the casts are redundant. This shows that $BC(e)$ reduces to $BC(v')$.

case $c = C.i$

Then, field access operators being unary, we have $n = 1$. By Theorem 40 (Field access soundness), $e \longrightarrow v'_{\text{shift}(sc(C),C')+i}$, with $e'_1 \longrightarrow \textbf{new}\ C'\ v'_1\ ...\ v'_n$. Therefore, by induction hypothesis, $BC(e'_1) \longrightarrow BC(\textbf{new}\ C'\ v'_1\ ...\ v'_n)$, which is equal to $BC(v'_1); ...; BC(v'_n); \texttt{new}\ C'$ by Definition 85, since Lemma 88 (Bytecode application) shows casts in $BC(\textbf{new}\ C'\ v'_1\ ...\ v'_n)$ are redundant. So $BC(e) \longrightarrow BC(v'_1); ...; BC(v'_n); \texttt{new}\ C'; \texttt{field}\ C.i$, and by the reduction for `field`, $BC(e) \longrightarrow BC(v'_{\text{shift}(sc(C),C')+i})$.

case $c = m$

Since $m\ v_1\ ...\ v_n$ is well-typed, we know by subject reduction that there exists a value v' and an index I such that

$$m\ v_1\ ...\ v_n \longrightarrow (\lambda x_1 ... x_n.e_I)\ v_1\ ...\ v_n \longrightarrow e_I\,[x_1 \leftarrow v_1] ... [x_n \leftarrow v_n] \longrightarrow v'$$

where $m\ \overline{\pi}_I = \lambda x_1 ... x_n.e_I$ is the most precise implementation of m such that $\overline{\pi}_I$ matches $(v_1, ..., v_n)$. Let τ and $\forall\vartheta.\kappa \Rightarrow \theta_1 \rightarrow ... \rightarrow \theta_n \rightarrow \theta'$ be the type of m. Let τ' be the type of v'.

The semantics of the bytecode given in Figure 11.2 then implies that:

$$\begin{array}{ll} & BC(v_1); ...; BC(v_n); \texttt{call}\ m \\ \longrightarrow & BC(v_1); \texttt{store}\ x_1\ \texttt{in}\ ... BC(v_n); \texttt{store}\ x_n\ \texttt{in} \\ & \overline{\overline{BC_{\pi_{i,j}}(BC(\texttt{load}\ x_j))}\ ; \texttt{iftrue}\ BC(e_i); \texttt{cast}\ codom(\tau)} \\ \longrightarrow & \overline{\overline{BC_{\pi_{i,j}}(BC(v_j))}; \texttt{iftrue}\ BC(e_i)\,[\texttt{load}\ x_1 \leftarrow BC(v_1), ..., \texttt{load}\ x_n \leftarrow BC(v_n)]; \texttt{cast}\ codom(\tau)} \end{array}$$

We know that $\overline{\pi}_I$ matches $(v_1, ..., v_n)$. Since the patterns are ordered by specificity, we know furthermore that for all j strictly smaller than I, $(\pi_{j,1}, ..., \pi_{j,n})$ does not match $(v_1, ..., v_n)$. So by Lemma 86, the first test that succeeds is for the index I. Therefore, $BC(v_1); ...; BC(v_n)\texttt{call}\ m$ reduces to $BC(e_I)\,[\texttt{load}\ x_1 \leftarrow BC(v_1), ..., \texttt{load}\ x_n \leftarrow BC(v_n)]; \texttt{cast}\ codom(\tau)$.

Since $e_I\,[x_1 \leftarrow v_1, ..., x_n \leftarrow v_n] \longrightarrow v'$, we know by induction hypothesis that $BC(e_I\,[x_1 \leftarrow v_1, ..., x_n \leftarrow v_n]) \longrightarrow BC(v')$. Moreover, by Lemma 90, $BC(e_I\,[x_1 \leftarrow v_1, ..., [\leftarrow\leftarrow x]_n]v_n) = BC(e_I)\,[\texttt{load}\ x_1 \leftarrow BC(v_1), ..., \texttt{load}\ x_n \leftarrow BC(v_n)]$. Therefore,

$$BC(e_I)\,[\texttt{load}\ x_1 \leftarrow BC(v_1), ..., \texttt{load}\ x_n \leftarrow BC(v_n)]; \texttt{cast}\ codom(\tau) \longrightarrow BC(v'); \texttt{cast}\ codom(\tau)$$

It remains to be shown that this last cast succeeds. By Definition 28, the type of $m\ v_1\ ...\ v_n$ is $\tau_s = \forall\vartheta\vartheta_v.\kappa \wedge \kappa_v \wedge \overline{\theta_{v_i} \leq \theta_i} \Rightarrow \theta'$. By Theorem 10 (Subject reduction), we know that $\tau' \leq \tau_s$ holds. Furthermore, by Definition 27 and Triv, it is easy to see that $\tau_s \leq \forall\vartheta.\kappa \Rightarrow \theta'$. Therefore, by transitivity, $\tau' \leq \forall\vartheta.\kappa \Rightarrow \theta'$. So, by Lemma 89, $BC(v')$ has type $BC(\tau')$, and $BC(\tau') \leq BC(\forall\vartheta.\kappa \Rightarrow \theta')$. By Definition 83, $BC(\forall\vartheta.\kappa \Rightarrow \theta')$ is equal to $BC_\kappa(\theta')$, which is also $codom(\tau)$. That is, $BC(v')$ has a smaller bytecode type than $codom(\tau)$, so the cast succeeds. Thus, $BC(m\ e'_1\ ...\ e'_n) \longrightarrow BC(v')$.

case $e = e_1\ e_2$ **with** e **not of the form** $c\ e'_1\ ...\ e'_n$

Then by Definition 85, $BC(e_1\ e_2)$ is equal to $BC(e_1); \texttt{cast Fun}; BC(e_2); \texttt{call Fun.apply}$.

By Theorem 10 (Subject reduction), we know that there exists values v_1 and v_2 such that $e_1 \longrightarrow v_1$, $e_2 \longrightarrow v_2$, and $v_1\ v_2$ is well typed (**1**). Therefore, by induction hypothesis, $BC(e_1) \longrightarrow BC(v_1)$ and $BC(e_2) \longrightarrow BC(v_2)$. Therefore, $BC(e_1); \texttt{cast Fun}; BC(e_2); \texttt{call Fun.apply} \longrightarrow BC(v_1); \texttt{cast Fun}; BC(v_2); \texttt{call Fun.apply}$.

We now reason by case on v_1. By (1), v_1 must be a functional value. There are therefore two cases:

case $v_1 = \lambda x'_1.e'_1$

Then, by Definition 85, $BC(v_1)$ is $\texttt{new Fun()\{ Object apply(Object x}'_1\texttt{) \{}\ BC(e'_1)\ \texttt{\}\}}$. Therefore, the cast succeeds, and $BC(v_1); \texttt{cast Fun}; BC(v_2); \texttt{call Fun.apply}$ reduces to $\texttt{new Fun()\{ Object apply(Object x}'_1\texttt{) \{}\ BC(e'_1)\ \texttt{\}\}}; BC(v_2); \texttt{call Fun.apply}$, which reduces to $BC(v_2); \texttt{store}\ x'_1\ \texttt{in}\ BC(e'_1)$, which reduces to $BC(e'_1)\,[\texttt{load}\ x'_1 \leftarrow BC(v_2)]$. By Lemma 90 (Bytecode substitution), this last expression is equal to $BC(e'_1\,[x'_1 \leftarrow v_2])$. Since e reduces to $v_1\ v_2$ which reduces to $e'_1\,[x'_1 \leftarrow v_2]$, the property holds in this case.

case $v_1 = c\ v'_1\ ...\ v'_p$ **with** $0 \leq p < arity(c)$

Let n be $arity(c)$. Then, by Definition 85, $BC(v_1)$ is $\texttt{new Fun()\{ Object apply(Object x}_{\texttt{p+1}}\texttt{) \{}\ BC(\lambda x_{p+2}...x_n.c\ \ v'_1\ \ ...\ \ v'_p\ \ x_{p+1}\ \ ...\ \ x_n)\ \texttt{\}\}}$. Therefore, the cast succeeds, and $BC(v_1); \texttt{cast Fun}; BC(v_2); \texttt{call Fun.apply}$ reduces to $\texttt{new Fun()\{ Object apply(Object x}_{\texttt{p+1}}\texttt{) \{}\ BC(\lambda x_{p+2}...x_n.c\ \ v'_1\ \ ...\ \ v'_p\ \ x_{p+1}...x_n)\ \texttt{\}\}}; BC(v_2)$; $\texttt{call Fun.apply}$, which reduces to $BC(\lambda x_{p+2}...x_n.c\ \ v'_1\ \ ...\ \ v'_p\ \ x_{p+1}...x_n)\,[\texttt{load}\ \ x_{p+1}\ \leftarrow BC(v_2)]$. By Lemma 90 (Bytecode substitution), this last expression is equal to $BC(\lambda x_{p+2}...x_n.c\ v'_1\ ...\ v'_p\ x_{p+1}...x_n\,[x_{p+1} \leftarrow v_2])$, that is $BC(\lambda x_{p+2}...x_n.c\ v'_1\ ...\ v'_p\ v_2\ x_{p+2}...x_n$. By Definition 85, this is equal to $BC(c\ v'_1\ ...\ v'_p\ v_2)$. We therefore have shown that $BC(e)$ reduces

to $BC(c\ v'_1\ ...\ v'_p\ v_2)$, with e reducing to $c\ v'_1\ ...\ v'_p\ v_2$. If $c\ v'_1\ ...\ v'_p\ v_2$ a value, the property is proved. Otherwise, by Theorem 10 (SUBJECT REDUCTION), there exists a value v such that $c\ v'_1\ ...\ v'_p\ v_2$ reduces to v. Since e is not of the form $c\ e_1\ ...\ e_n$, there has been at least one step of reduction. Therefore, we can apply the induction hypothesis to $c\ v'_1\ ...\ v'_p\ v_2$, which shows that $BC(c\ v'_1\ ...\ v'_p\ v_2)$ reduces to $BC(v)$, which finishes the proof.

case $e = \texttt{let}\ x_1\ \texttt{be}\ e_1\ \texttt{in}\ e_2$

We know by subject reduction that there exist values v_1 and v such that $\texttt{let}\ x_1\ \texttt{be}\ e_1\ \texttt{in}\ e_2 \longrightarrow \texttt{let}\ x_1\ \texttt{be}\ v_1\ \texttt{in}\ e_2 \longrightarrow e_2\,[x_1 \leftarrow v_1] \longrightarrow v$.

Let τ_1 and $\forall\vartheta_1.\,\kappa_1 \Rightarrow \theta_1$ be $\mathsf{type}(e_1)$, Let τ_2 and $\forall\vartheta_2.\,\kappa_2 \Rightarrow \theta_2$ be $\mathsf{type}(\texttt{let}\ x_1\ \texttt{be}\ e_1\ \texttt{in}\ e_2)$, and τ_{v_1} and be $\mathsf{type}(v_1)$. We therefore have $\tau_{v_1} \leq \tau_1$ (**1**) by Theorem 10 (SUBJECT REDUCTION). By definition, $BC(\texttt{let}\ x_1\ \texttt{be}\ e_1\ \texttt{in}\ e_2) = BC(e_1); \texttt{cast}\ BC_{\kappa_2}(\theta_1); \texttt{store}\ x_1\ \texttt{in}\ BC(e_2)$. By induction hypothesis, $BC(e_1) \longrightarrow BC(v_1)$ and $BC(v_1)$ has type $BC(\tau_{v_1})$. Therefore, $BC(\texttt{let}\ x_1\ \texttt{be}\ e_1\ \texttt{in}\ e_2) \longrightarrow BC(v_1); \texttt{cast}\ BC_{\kappa_2}(\theta_1); \texttt{store}\ x_1\ \texttt{in}\ BC(e_2)$.

We now show that the cast succeeds. By Definition 28, constraint κ_1 is included in κ_2: this is immediate from the definition if x_1 is not free in e_2, and follows from a straightforward induction otherwise. Therefore, by TRIV, the constraint implication $\forall FV(\tau_1), FV(\tau_2), t.\ \kappa_2 \wedge \theta_1 \leq t \models \kappa_1 \wedge \theta_1 \leq t$ holds. That is, by Definition 29 and Definition 27, type $\forall\vartheta_1.\,\kappa_1 \Rightarrow \theta_1$ is a subtype of $\forall\vartheta_1\vartheta_2.\,\kappa_2 \Rightarrow \theta_1$. Therefore, by transitivity with (1), we have $\tau_{v_1} \leq \forall\vartheta_1\vartheta_2.\,\kappa_2 \Rightarrow \theta_1$. By Lemma 89, this shows that $BC(\tau_{v_1}) \leq BC(\forall\vartheta_1\vartheta_2.\,\kappa_2 \Rightarrow \theta_1)$ holds. Furthermore, $BC(\forall\vartheta_1\vartheta_2.\,\kappa_2 \Rightarrow \theta_1)$ is equal to $BC_{\kappa_2}(\theta_1)$ by Definition 83. Therefore, $BC(\tau_{v_1})$ is a bytecode subtype of $BC_{\kappa_2}(\theta_1)$. This shows that the cast always succeeds, and

$$BC(\texttt{let}\ x_1\ \texttt{be}\ e_1\ \texttt{in}\ e_2) \longrightarrow BC(v_1); \texttt{store}\ x_1\ \texttt{in}\ BC(e_2)$$

By definition, $BC(v_1); \texttt{store}\ x_1\ \texttt{in}\ BC(e_2) \longrightarrow BC(e_2)\,[\texttt{load}\ x_1 \leftarrow BC(v_1)]$. By Lemma 90, $BC(e_2)\,[\texttt{load}\ x_1 \leftarrow BC(v_1)]$ is equal to $BC(e_2\,[x_1 \leftarrow v_1])$. By induction hypothesis, $BC(e_2\,[x_1 \leftarrow v_1])$ reduces to $BC(v)$. Therefore, by transitivity, we have $BC(\texttt{let}\ x_1\ \texttt{be}\ e_1\ \texttt{in}\ e_2) \longrightarrow BC(v)$.

■

Chapter 12

Typing kinds

In this chapter, we consider the algorithms used to perform type-checking in our constrained type system, based on $\mathrm{ML}_{\leq}$.

Type-checking is needed in two places. First, in the core language of Section 1, where type are inferred, we need to check that the resulting types are well-formed. As defined in Section 2.2, a type $\forall\vartheta.\,\kappa \Rightarrow \theta$ is well-formed if the constraint implication $\forall\emptyset.\,\mathit{true} \models \kappa$ holds.

Second, when type-checking multi-methods as in Section 4, we need to check that for each method implementation, the restriction of the method type to the patterns is well-formed, and that the inferred type of the implementation is below that restricted type. This second condition amounts to checking subtyping between polytypes, which is defined in Section 2.2 as a certain constraint implication.

Therefore, we only need to be able to decide constraint implication. We will present algorithms to do so in this chapter. First, we recall that constraints involving constructed monotypes can be decomposed into atomic constraints, on which the implication is decided. Then, we briefly summarize the existing techniques used to decide implication on atomic constraints in core $\mathrm{ML}_{\leq}$. This corresponds to the system presented in Section 2.2. In the last section, we present a new algorithm to decide constraint implication, in the presence of kinds, as defined in Section 10.2.

12.1 Constraint decomposition

$\mathrm{ML}_{\leq}$ is a structural type system. That is, an inequality between monotypes always follows from their having the same shape, and their sub-components being related. This is formalized in the variable elimination rule of constraint implication, which we recall here:

$$\textsc{VElim}\ \frac{t \leq \phi_V[\overline{\theta}] \in \kappa \text{ or } t \geq \phi_V[\overline{\theta}] \in \kappa \qquad \phi'_V, \overline{t} \text{ fresh}}{\forall\vartheta.\,\kappa \models \kappa \wedge t = \phi'_V[\overline{t}]}$$

A consequence of this rule, proved in [5], is that every constraint implication problem can be reduced to a simpler problem, involving only constraints on atoms: type constructors and type variables. We will therefore only consider the decision of constraint implications where atomic constraints are either of the form $\phi_V \leq \phi_V$ or of the form $t \leq t$.

12.2 Core $\mathrm{ML}_{\leq}$

Given the constraint implication $\forall\vartheta.\,\kappa \models \kappa'$, one can construct a model of κ. That is, the set of constant type constructors and type variables in κ is equipped with the partial pre-order $\leq$ induced by the constraint κ and, for the constant constructors, by the implicit type structure $\mathcal{T}$. The constraint implication then holds if there exists a substitution σ from the variables of κ' to the model, such that $\sigma(\kappa')$ is true in the model.

This problem is NP-complete, as (implicitly) shown in [43]. However, it is possible to find algorithms that are only polynomial in practice, similarly to the situation of type inference for ML. An algorithm is sketched in [5], and is the basis of the implementation made by Alexandre Frey for the language Jazz. This implementation was also used as a basis for the implementation in the Nice compiler.

The essence of the algorithm is that each variable in κ' is assigned a domain, which is a set of possible mappings of that variable into the model. The domains can be reduced, by using the inequalities in κ and κ', and their consequences by transitivity. For instance, if κ' contains the constraint $t_V \leq c_V$, then the domain of t_V can be reduces to only those values in the model that are smaller than c_V. This can in turn be used to reduce the domain of other variables that are in relation with t_V. If this reduction leaves at least one domain empty, then the implication does not hold. If all domains have size one, then a solution was found. Otherwise, it is necessary to pick one variable with a domain of cardinal at least 2, fix the mapping of that variable to each value in the domain in turn, and restart the reduction process. If that leads to a failure, it is necessary to backtrack and choose a different mapping for that variable inside its domain. If at least one attempts succeeds, then we have found a solution. If all attempts fail, the constraint implication does not hold.

12.3 Adding kinds

This algorithm can be extended to implement the system defined in Section 10.2. In this context, the implicit type structure $\mathcal{T}$ contains, besides the subtypings between constant constructors, the implementations and abstractions of kinds of these constants. Furthermore, we now deal with constraint implications where constraints are taken from the grammar $\phi_V \leq \phi_V \mid t \leq t \mid t_V : K$.

As before, given such a constraint implication $\forall \vartheta.\ \kappa \models \kappa'$, we first construct a model of κ. That is, the set of constant type constructors and type variables in κ is equipped with the partial pre-order $\leq$ induced by the constraint κ and, for the constant constructors, by the implicit type structure $\mathcal{T}$. Furthermore, each type variable t_V in κ is made to implement a kind K if and only if κ contains the constraint $t_V : K$ or the constraint $t_V : K'$ with some kind K' that extends kind K. Note that this closely matches the construction of the extended type structure $\mathcal{T}^1$ in the completeness proof of Theorem 80 (Correction and completeness of the axiomatization of kinds). The only difference in that, in the proof, t_V is made to implement K if κ implies $t_V : K$. We cannot directly apply that definition here, as this would require a way to decide constraint implication, which is precisely what we are constructing. However, our restricted definition of the kinds that t_V implements in the model is sufficient. Indeed, the only other way in which κ could imply that t_V implements another kind K' is by application of axiom EqImp. In that case, t_V would be equivalent to another element ϕ_V in the model, which implies that their domains would be the same. Any implication of t_V implementing K' would therefore also follow from ϕ_V implementing K'.

Second, we compute the domains of each variable in κ', reducing them as before by using the constraints in κ'. We can further reduce the domains by noticing that if κ' constaint $t_V : K$, then the mapping of t_V must be a constant of the model that implements K. The main issue is to be able to apply axiom Abs to deduct further constraints on the variables. For this, we need to compute views, as defined in Definition 77. However, a naive approach would have a prohibitively high complexity, since the definition involves simultaneous quantification over four type constructors. Fortunately, it is possible to significantly reduce the amount of work to be done.

First, by condition 5 of Definition 78, we know that the view of constant type constructors is independent of the possible extensions to the type structure. Therefore, it is possible to compute those views only once, in the module where the type constructor is introduced, and not in every module of the program.

Second, if $\mathsf{view}_K(c_V) = c'_V$, then it follows from Requirement 76 (Kinds) and Definition 77 that, for all c''_V below c_V, $\mathsf{view}_K(c''_V)$ is also c'_V. Furthermore, in the Abs axiom, the requirement is that the variable is below the origin of the view. If we used $\mathsf{view}_K(c''_V) = c'_V$ to apply that axiom, we could as well use $\mathsf{view}_K(c_V) = c'_V$, since the variable is also below c_V by transitivity. Therefore, we can limit the computation of views to maximal values for the origin.

Algorithm 1 defines the function compute_view, which computes views. In particular, the auxiliary

function SET_VIEW is used to implement the second optimization: if the type constructor that abstracts the kind can be the origin of the view, then we can stop the computation. Otherwise, we compute it recursively for each of its direct sub-constructors.

Algorithm 1 Computation of views

```
procedure COMPUTE_VIEWS
    for all K ∈ Kinds do
        for all C that abstracts K do
            SET_VIEWS(C, K)
        end for
    end for
end procedure

procedure SET_VIEWS(C, K)          ▷ Computes view_K(C'') for a minimal set of C'' below C.
Require: The type constructor C abstracts K
    min ← ⊥
    for all C' that implements K do
        if C ≤ C' then
            if min = ⊥ or C' ≤ min then
                min ← C'
            end if
        else
            if ∃C_0 such that C_0 ≤ C and C_0 ≤ C' then
                min ← ⊥
                break
            end if
        end if
    end for
    if min ≠ ⊥ then
        view_K(C) ← min
    else
        for all C'' directly below C do
            set_views(C'', K)
        end for
    end if
end procedure
```

Given that this computation is done, we can now use Algorithm 2 to complete the model of κ by repeated application of the ABS axiom.

Note that the iteration is needed because the consequence of the application of an instance of ABS can create the conditions for another instance to be applicable. This is the case in the following contrived example. Given the hierarchy of Figure 12.1, the two kinds K_1 and K_2, and that B abstracts and implements K_1 and A abstracts and implements K_2, we consider the constraint implication $\forall T.\, T : K_1, T : K_2, D \leq T \models A \leq T$.

By definition, $\mathsf{view}_{K_1}(D)$ is B and $\mathsf{view}_{K_2}(C)$ is A. It is not directly possible to conclude that $A \leq T$. However, by ABS for kind K_1, $\forall T.\, T : K_1, T : K_2, D \leq T \models B \leq T$. Since $C \leq B$, by transitivity we have $\forall T.\, T : K_1, T : K_2, D \leq T \models C \leq T$. This in turns allows to apply ABS on K_2, and to conclude that the implication holds.

Algorithm 2 Applying the ABS axiom

```
procedure SATURATE_ABS                          ▷ Finds all consequences of the ABS axiom.
    repeat
        for all K ∈ Kinds do
            for all C such that view_K(C) ≠ ⊥ do
                for all D below C do
                    for all D′ above D that implements K do
                        add view_K(C) ≤ D′ in the model
                    end for
                end for
            end for
        end for
    until nothing changed
end procedure
```

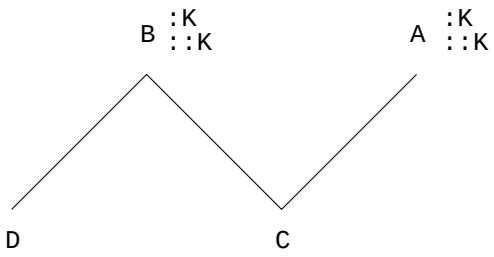


Figure 12.1: Example hierarchy

Chapter 13

The Nice language

The theoretical results of this dissertation have been used as the foundation for implementing a complete, general-purpose language called Nice. The motivation for this practical work was three-fold. First, it serves as a proof of concept by showing that the system can be implemented with a reasonable amount of work and deliver acceptable performance. Second, since the implementation is available freely[1], it allows users to write real programs in Nice when they decide its properties provide them with a benefit. This also contributes to popularizing those features that are not present in most mainstream languages, for instance multi-methods. Finally, this concrete use in turn generates feedback that is inspiring for pointing out new research themes.

13.1 Syntax

This section lists the differences between the concrete syntax used in this document and the Nice language. This should be sufficient to read the real Nice programs presented in the next chapter. A general difference is that the syntax of Nice follows the syntactical tradition of Algol (and therefore also of C, Java and other languages) by placing the type of a variable before its name, and similarly for fields and for the return type of a method.

For more details, one can refer to the online version of the user manual at `http://nice.sf.net/manual.html`.

13.1.1 Classes

The syntax for classes is close to the one presented in Chapter 3. The only difference is that field types precede their names and that field declarations are terminated by the `;` character, in the syntactic tradition of Algol [36]. Furthermore, type parameters are listed between angle brackets, like C++ [41] template parameters and Java 1.5 type parameters.

```
class C<T> {
  Type1 field1;
  Type2 field2;
}
```

13.1.2 Methods

Nice is a language with multi-methods. Therefore, methods can be declared outside classes, at the package level. It keeps the distinction between method declaration and method implementation. Method implementations dispatch on their arguments by specialization of the class of arguments. The return type does

[1] The implementation is licensed under the GNU General Public License, and can be downloaded from the website `http://nice.sf.net`, which also includes a user manual, links to mailing lists devoted to Nice and further information useful to users of the language.

not need to be written, since it is computed as the specialization of the method type for the specialized arguments. Compared to Chapter 8, the keywords `method` and `implementation` are dropped. Furthermore, types are written as in C and Java.

For implementations, a pattern can be absent, in which case it refers to the pattern "_", which matches any value.

```
ReturnType methodName(ParamType1 param1, ParamType2 param2);

methodName(param1, param2) { ... }
methodName(C1 param1, C2 param2) { ... }
```

When a method has type parameters, they are introduced in front of the declaration between angle brackets.

```
<T> T id(T);
```

While this syntax for methods would be sufficient, it looks quite unfamiliar to programmers used to traditional object-oriented syntaxes. A design choice for Nice has been made to also offer more traditional syntax when possible. The motivation of this decision is to facilitate the transition and to help programmers focus on the new features instead of struggling to learn a new syntax for the existing features. The traditional notation for declaring methods inside classes (respectively abstract interfaces) is therefore also allowed as syntactic sugar for declaring a multi-method with a first parameter named `this` belonging to the current class (respectively to a class implementing the current abstract interface). Similarly, the `alike` keyword is supported as syntactic sugar for the the type of `this`, which is implicitly quantified as a subtype of the current class (respectively a type implementing the current abstract interface, see next section). Similar syntactic sugar is also provided for implementing a method inside a class. Finally, it is possible simultaneously declare a method a provide its default implementation.

For instance, the following declarations use the syntactic sugar.

```
class C {
  C m() = this;
}

class D {
  m() = new C();
}
```

They are equivalent to the following de-sugared version.

```
class C {}
C m(C);
m(C this) = this;

class D {}
m(D this) = new C();
```

13.1.3 Kinds

Because this feature has not yet been popularized, finding a good terminology remains to be done. Nice has been modeled after Java for most of the syntax. Kinds are created with the keyword `abstract interface`. The rationale is that a kind is similar to an interface in describing some facilities that a class must possess to implement the kind. However it is "abstract" because it is not itself a type; in particular the type of a variable can not be an abstract interface.

Using most the syntax described in this chapter, one can give the concrete Nice syntax of the solution with kinds in Section 6.2.

```
abstract interface Comparable {
  boolean less(alike);
}

class String implements Comparable {
  less(String other) { ... }
}

class Date implements Comparable {
  less(Date other) { ... }
}
```

13.2 Type checking

Nice's type system is the based on the extension of $ML_{\leq}$ with modular kinds formalized in Chapter 10.2 and implemented following the algorithms presented in Chapter 12. The implementation of the constraint solver was based on Alexandre Frey's implementation for the Jazz language, extended in particular for the support of kinds.

13.2.1 Option types

The type system includes another extension for safe handling of the `null` value which is present in Java and many other languages with references. The `null` value is usually supposed to be of the special "bottom" type, meaning that it can be used in all contexts where a reference is expected. However, most operations fail at runtime when applied to the `null` value, for instance when accessing a field. We extended the type system to make it possible to prevent such failures statically. Because of time constraints, we do not formalize this extension in this dissertation but briefly present it in this section.

Our extension makes it possible to distinguish between types that include the `null` value and those that do not. Technically, we introduce two unary type constructors, ? (for "maybe") and ! (for "surely"). Therefore, instead of the single type `String` for character strings, we use `!String` that only allows real strings, and `?String` that also allows the `null` value[2]. We make ! smaller than ?, which implies that `!String` is a subtype of `?String`. We also make those two type constructors covariant. This system can therefore be implemented using the standard version $ML_{\leq}$, with ! and ? being normal type constructors. This can therefore be viewed as a layer about $ML_{\leq}$, and does not need changes in the core constraint implication solver.

To achieve type safety, we simply disallow operations on possibly null values. More precisely, we give those operations function types with domain types prefixed with the ! type constructor. For instance, given a class `C` with a field of type `!String`, the field access method has type `!C` $\rightarrow$ `!String`. We can then give the `null` constant the polymorphic type $\forall T.?T$ without breaking type safety.

To make use of values with types constructed on ?, we need to differentiate between the null and non-null case, and to be able to use the value with the corresponding type constructor on ! in the second case. It would have been possible to provide a ML-style matching operator to that effect, binding the value to a new name in the second case. However, it feels more natural to make use of the existing style in languages including `null`, which is to use tests of the forms `x` $==$ `null` and `x` $!=$ `null` to distinguish the two cases. Therefore, we incorporated in the high-level typechecker rules to recognize such tests and take them into account, which amounts to performing a static dataflow analysis on local variables. Basically, inside a branch running when `x` $!=$ `null` succeeds or when `x` $==$ `null` fails, and provided that `x` is not captured by a closure assigning a possibly non-null value to it, `x` can be assumed non-null. Such information can be merged at the points where branches join.

[2] To avoid making the syntax of types heavier, we allow the non-null case, which we consider is the most frequent by far, to be the default, so that the ! type constructor is optional. Thus `String` is a synonym for `!String`. For clarity, we keep ! explicit in this section.

Note that this type system feature is only superficially similar to ML's option type, defined with `type a option = None | Some of a`. A first difference is that ML option types can be nested, as in `string option option`. A valid value of this type is `Some(None)`. We only want to handle values that are either `null` or a normal reference, and therefore syntactically disallow consecutive occurrences of ! and ?. This allows to represent these values without any overhead, `null` being simply a specific value different from any reference. Furthermore, since `!String` is a subtype of `?String`, the user can directly use a non-null string where a possibly null one is expected. In ML, the user has to manually wrap it using the `Some` data constructor.

13.3 Code generation

The Nice compiler generates Java bytecode [31] as formalized in Chapter 11. This choice makes it possible to execute Nice programs with good performance on any computer architecture for which a Java Virtual Machine exists. Furthermore, it allows to make use of the many Java libraries in Nice programs. The compiler automatically assign Nice types to existing Java classes, fields and methods, requiring no special effort to start using such these existing libraries. Furthermore, it is possible to explicitly "retype" them by assigning them arbitrary Nice types when it is necessary, in particular when that can result in a more precise type than what is possible to express in the Java type system. Nice code is also compiled in a way that makes it easier to use from Java code. Thus, multi-methods are compiled as instance methods of the class of (the erasure of) their first argument whenever possible, that is if the class is also generated from Nice code. This makes it possible to call the method in Java with the standard `x.foo(y)` syntax, instead of `someArbitraryClass.foo(x,y)`. Furthermore, this even allows overriding a Nice multi-method in Java code, although this obviously can only be done for specializing on the first argument. All in all, this makes projects mixing Java and Nice code as simple as possible, which is important for some users who have an important existing code base in Java and want to extend it in Nice without first translating their whole project into Nice.

Chapter 14

The expression problem

The expression problem is a classical "expressiveness benchmark" for programming languages. It can be described as the following situation. Given recursively defined datatypes and operations on those datatypes, we want to be able to extend datatypes by defining new cases and operations by adding new ones. Solving this problem is very important in practice, since it is essential to enable modular and extensible programming for large programs, as identified in our third and fourth criteria in the introduction of this thesis. We base this chapter on the presentation in [38], which proposes the following requirements:

1. Possibility to define both new datatypes and new operations;
2. Strong static type safety;
3. No modification of existing code nor duplication;
4. Separate compilation;
5. Independent extensibility: it should be possible to combine independently developed extensions so that they can be used jointly.

Their review of existing solutions shows that no previous solution meets all these requirements. The functional approach makes it easy to add new operations, but impossible to add new datatypes without modifying existing code. Dually, the classic object-oriented approach makes it easy to add new datatypes, but impossible to add new operations without modifying existing code. Variants of the Visitor pattern are also available, but they either lack type safety [27, 39] or require defaults [44]. Defaults are required to be able to handle all possible future extensions, which is in practice often impossible to do in a semantically correct way, which forces the programmer to resort to runtime failure. The possibility to use multi-methods with required default implementations for external multi-methods [18] is also considered. It matches all criteria apart from this requirement for defaults.

Two solutions are proposed [38] and implemented in the Scala language, using *traits*. The first solution is based on the object-oriented approach. By leaving some types abstract until the program is closed, it allows to define new operations in extensions. Dually, the second solution is based on the functional approach. By giving an abstract type for the visitor used, which is specified when the program is closed, it allows to define new datatypes. Thus, both solutions allow extension in the direction that was previously impossible, although stays more verbose and less straightforward than the natural one in each approach.

We will now present a solution to this expression problem using our multi-methods. In particular, we will show that our single solution unifies the object-oriented and the functional approach, since both directions of extension are identically simple. We do not require default implementations. In Section 14.4, we show that our solution has the same modularity properties as either solution in [38].

In Section 14.5, we review another proposal to solve the expression problem written in the OCaml language, and we compare it with our proposal.

14.1 Base

At the core of the expression problem is the base package that defines an abstract class `Exp` for expressions, and a method `eval` that takes an expression and returns an integer. It also defines a concrete subclass `Num`.

```
package base;

abstract class Exp
{
  int eval();
}

class Num extends Exp
{
  int value;

  eval() = value;
}
```

Unlike other solutions, no special hindsight is needed to make this framework open for future extension. This package can be imported an used in a program, without any particular work to "close" it.

```
package base.test;
import base;

void main(String[] args) {
  let e = new Num(value: 7);
  println(e.eval);
}
```

14.2 Data extension

14.2.1 Linear extension

We define a simple extension of `base` by adding a new datatype for representing the addition of two expressions.

```
package plus;
import base;

class Plus extends Exp
{
  Exp left;
  Exp right;

  eval() = left.eval + right.eval;
}
```

Independently, we can define another extension adding negation.

```
package neg;
import base;

class Neg extends Exp
{
```

```
  Exp term;

  eval() = - term.eval;
}
```

14.2.2 Combining independent extensions

Those two independently developed extensions can be combined, simply by importing both.

```
package plusneg;
import plus;
import neg;
```

14.3 Operations extensions

Adding new operations is equally simple. It is sufficient to define a multi-method for the operation, and to implement it for the known datatypes.

```
package show;
import base;

String show(Exp);

show(Num e) = e.value.toString;
```

Note that `Plus` and `Neg` are not known in this package, and therefore no other implementation of `show` is required. We do not need to give `show` a default implementation either.

14.3.1 Linear extensions

We can adapt independently developed extensions of `base` so that they support the `show` operation. To this end, we simply import the corresponding packages, and add the required implementations for method `show`.

```
package showplusneg;
import show;
import plusneg;

show(Plus plus) = plus.left.show + "+" + plus.right.show;
show(Neg neg) = "-(" + neg.term.show + ")";
```

Note that if we omitted these implementations, the compilation of package `showplusneg` would have resulted in a compile-time error because of the coverage test for method `show`.

We can use this extended version in a program to uses both `show`, `Plus` and `Neg`.

```
package showplusneg.test;
import showplusneg;

void main(String[] args) {
  let e = new neg.Neg(term: new plus.Plus(left:  new base.Num(value: 7),
                                          right: new base.Num(value: 6)));
  println(e.show + " = " + e.eval);
}
```

14.3.2 Tree transformer extensions

It is equally easy to add new operations that return an expression. For instance, we define a method `double` that return a number similar to the argument except that all `Num` leafs have their value doubled.

```
package doubleplusneg;
import plusneg;

Exp double(Exp);

double(Num num) = new Num(value: num.value * 2);
double(Plus p) = new Plus(left: p.left.double, right: p.right.double);
double(Neg neg) = new Neg(term: neg.term.double);
```

In [38], this case requires defining abstract factory methods to create the new objects to be returned, and to instantiate those factory methods in the main program. This comes from the fact that they need to create new versions of the type `Exp` in each extending package. In our model, there is only one type `Exp`, and object creation does not pose any problem. An advantage of their solution is that it allows to refer to "the version of type `Exp` that only supports `plus` and `neg` but not `show`" even in a program that uses `show` in other parts. However, it is not clear how useful this is in practice. The downside of this distinction is that it becomes possible to get errors when trying to mix different versions of type `Exp`, which is likely to be confusing for the programmer.

A program using method `double` can be written directly:

```
package doubleplusneg.test;
import doubleplusneg;

void main(String[] args) {
  let e = new Plus(left: new Neg(term: new Plus(left:  new Num(value: 1),
                                                right: new Num(value: 2))),
                   right: new Num(value: 3));
  println(e.double.eval);
}
```

14.3.3 Combining independent extensions

We now put it all together by combining all previous extensions. Again, this is a simple question of importing the right packages. There is no need to explicitly plug the pieces together.

```
package doubleplusneg.test;
import doubleplusneg;
import showplusneg;

void main(String[] args) {
  let e = new Plus(left: new Neg(term: new Plus(left:  new Num(value: 1),
                                                right: new Num(value: 2))),
                   right: new Num(value: 3));
  println(e.double.show + " = " + e.double.eval);
}
```

14.3.4 Binary methods

Binary methods are methods whose implementation depends on the type of more than one of their arguments. They are hard to implement in a class-based language, whose methods are asymmetric between their first

argument and the other. Unsurprisingly since we use multi-methods which are by design symmetrical in all their arguments, handling binary methods can be done very naturally.

For instance, we define a method for testing the structural equality of two expressions.

```
package equals;
import base;

boolean equal(Exp e1, Exp e2);

// Default implementation
equal(e1, e2) = false;

equal(Num e1, Num e2) = e1.value == e2.value;
```

Note that we give a default implementation of `equal` returning false. This is not required and we could also not provide it, and instead provide implementations for all combinations of parameters. However, this would require a large number of implementations. Indeed, it is clear that most expressions are not structurally equal. It is therefore more practical to define the default as `false` and handle the few interesting cases explicitly.

We can now consider implementing `equal` in the context where `Plus` and `Neg` are defined.

```
package equalsplusneg;
import plus;
import neg;
import equals;

equal(Plus e1, Plus e2) =
  equal(e1.left, e2.left) && equal(e1.right, e2.right);

equal(Neg e1, Neg e2) =
  equal(e1.term, e2.term);
```

We can use all the features together in a test program:

```
package equalsshowplusneg;
import equalsplusneg;
import showplusneg;

void main(String[] args) {
  let term1 = new Plus(new Num(value: 1), new Num(value: 2));
  let term2 = new Plus(new Num(value: 1), new Num(value: 2));
  let term3 = new Neg(new Num(value: 2));

  print(term1.show + "=" + term2.show + "? ");
  println(term1.equal(term2));

  print(term1.show + "=" + term3.show + "? ");
  println(term1.equal(term3));
}
```

Note that although `equal` is a method defined at toplevel, it can be used as well using the "dot" notation which is usual in object-oriented languages.

14.4 Discussion

A crucial point to ascertain is whether our solution is satisfactory from a modularity point of view. This question is covered by two of the five requirements of the problem. First, it must be possible to separately typecheck and compile the packages containing the independent extensions. Second, it must be possible to combine those extensions to use them jointly. We now argue that both points are satisfied by our module system in general. It is therefore in particular the case of our solution to the expression problem.

As formalized in Chapter 8, we can typecheck modules independently based on the interfaces of their imported modules. Furthermore, we have shown that the typechecking of method implementations done in their module does not need to be duplicated, since it implies validity in the context of the whole program. This is true even in the presence of polymorphic types thanks to the fact that method implementations are checked with an open-world assumption. It is indeed possible that the typechecking of a module importing several other modules fails solely because of the content of those modules, but only because some method implementation is missing. This situation could arise in package `showplusneg` of Section 14.3. Package `showplusneg` imports package `show`, which defines a multi-method `show` whose domain contains the abstract class `Exp` without implementing `show` for `Exp`. That is, the `show` method does not have a default implementation in package `show`. This is valid in the context of package `show` since `Exp` is abstract. In parallel, package `plusneg` defines concrete subclasses `Plus` and `Neg` of `Exp` without knowledge of method `show`. When importing both packages, an implementation of `show` for `Plus` and `Neg` is required. This requirement is not a technicality required because of the use of multi-methods. It is fundamentally expressing that after independently developing two extensions that are not orthogonal, one needs to explicitly specify how they interact. In the solution using traits of [38], this requirement is exactly the same: trait `ShowPlusNeg` must define how the new classes `Plus` and `Neg` implement the show operation. A similar situation occurs when two modules `M1` and `M2` contain ambiguous implementations. In that case, a more precise implementation in `M` is sufficient to resolve the ambiguity[1].

In general, it is always possible in our system to combine valid independently developed modules by providing the adequate method implementations. Indeed, method coverage fails if some method implementations are missing, in which case the missing cases can be added, or if some implementations are ambiguous, in which case more precise implementations can be added. We argue that this possible requirement of additional method implementations is acceptable, and even desired. This requirement can only be avoided by restricting expressivity, which can also have the paradoxical effect of reducing actual static safety. For instance, as is argued in [34], when multi-methods are required to have a default implementation, it will sometimes happen that no sensible implementation exists, as with a method `area` on the abstract class `Shape`. Therefore, the default implementation can hardly do something else than fail, for instance by throwing a runtime exception. This introduces the risk of the program failing at runtime. In contrast, when independently importing the method `area` and a concrete class extending `Shape`, our system will report the need to implement `area` for that imported class, which should not be surprising. Similarly, in singly-dispatched languages, it is typical to simulate multiple dispatch using instance tests or some version of the visitor pattern. However, since the compiler has no knowledge about these techniques, there is no static guarantee that all cases are covered.

14.5 Comparison with polymorphic variants

Another solution to the expression problem is presented in [24]. That solution indeed meets the five criteria set at the beginning of this chapter. This result is quite remarkable, given the relatively early date of its publication and its compatibility with a major general purpose programming language. The solution is based on the used of polymorphic variants, which are in particular present in the OCaml language [29]. Essentially, types can consist of lists of variants open for the extension to other variants. This allows for functions that accept an open set of data cases. Extension of operations can be achieved by creating a new function that

[1] To make this solution possible when `M1` and `M2` contain *identical* implementations, we can refine the notion of method implementation ordering so that for identical implementations, the one defined in a module `M` is considered more precise than the one defined in an any module that `M` imports. This allows `M` to contain a disambiguating implementation of `m`.

calls the existing one for the existing cases. This can be made to work even if the function needs to be recursive, the old function recursively calling the new one. However, this has to be handled explicitly by adding an additional parameter to all recursive functions, and by closing the recursion explicitly where the set of data cases completely known.

The solution with polymorphic variants is less general in one aspect: it does not allow the creation of sub-cases, which is naturally achieved in object-oriented languages (and in particular in our solution) by the creation of subclasses. In other words, variants allow data extension in width only, not in depth.

Polymorphic variants use structural subtyping, while our solution uses nominal subtyping. Both forms of subtyping have their strengths and weaknesses. Nominal subtyping is more natural when types are declared as part of the design of the program, while structural subtyping is more suitable to type inference. With structural subtyping, types can become large, which can be problem when writing types to the programmer. On the other hand, it allows for polymorphism over an *ad-hoc* set of variants that can have been designed separately, simply by listing those variants in extension. While this is typically not possible with nominal subtyping, we gain a similar functionality thanks to ours kinds. Indeed, it is possible to declare that several existing classes or interfaces implements a new kind, effectively allowing the declaration of a method that accepts any type implementing the kind.

The solution of the expression problem using polymorphic variants requires the programmer to manually call the existing functions from the new ones. This corresponds in our solution to the multiple dispatch code that is generated automatically by the compiler. An advantage of writing it by hand is that it allows the existing functions to exist independently of the new ones, and to write several independent extensions of the same base function. On the other hand, our model is flat, in that new method implementations are unconditionally extending the base method. If several different behaviors are needed in the same program, it is required to add an additional parameter whose value can direct which one is desired. Our flat model makes is unnecessary to handle recursion explicitly, and makes it possible to let the compiler generate the dispatch code. The program can therefore be much more concise. This also opens the possibility for the compiler to generate more efficient dispatch code.

Chapter 15

Related work

15.1 Core

The $\lambda\&$ calculus [12] proposed an extension of the lambda-calculus with functions that dispatch on their arguments runtime types. They argued that considering methods as first-class elements rather than object components significantly simplifies the theory over record based models that require recursive types. However, this presentation was not aimed at modeling programming languages directly: they did not allow programmer's definable base types; the use of the & operator to build methods does not fit directly the incremental definition of method implementations.

The $\text{ML}_{\leq}$ type system and language [5, 6] is a remarkable attempt at unifying languages with multi-methods and ML-style type systems. It provided the main foundation on which we built our research. For comparison's sake, the instantiation of our algebraic type system with the type language of $\text{ML}_{\leq}$ in Section 2.2 together with the multi-method extension in Chapter 4 produces a system similar to the whole of $\text{ML}_{\leq}$. These presentations used a highly non-standard operational semantics and soundness proofs through the use of a typed abstract machine, which made the presentation and the proof of the system tedious and hard to understand. Our work uses a more conventional approach, which consists in describing an untyped language and untyped small-step semantics, and proving soundness through subject-reduction and progress lemmas. Another major achievement of this work is to separate the object-oriented part from the core language, both in presentation and in the proofs. Since the latter part is already well understood, this makes the study of the theoretical aspects of multi-methods much easier.

We formalize the type-checking of multi-methods, while [5] tackled formally only methods with one argument, and described how to extend the theory to methods with multiple arguments only informally. Similarly, open multi-methods and the modularity aspects were not formalized.

We also remove the need to annotate lambda-expressions with their domain, and show that their type can be inferred. Thus, only top-level declarations of multi-methods require type annotations.We believe that these top-level annotations are acceptable in a programming language, and that they cannot be avoided if modular type-checking is needed. Indeed, implementations of methods might not be known when type-checking of code making use of these methods is performed.

$\text{ML}_{\leq}$ had no notion of $\#C$ pattern. This feature is especially useful with polymorphic multi-methods: it makes it possible to implement methods of type $\forall t.\ t \to t$ or $\forall t.\, t \leq C \Rightarrow\ t \to t$ that are not the identity function but that actually return a newly constructed object. Useful concrete examples include the `clone` method, the `union` method on a hierarchy of collections with the precisetype $\forall t.\, t \leq \texttt{Collection} \Rightarrow\ (t, t) \to t$, etc.

Frey's doctoral dissertation [22] is an elegant algebraic approach to the typing of an ML-like language with objects, subtyping and multi-methods. Like our work, it builds on the earlier works on $\text{ML}_{\leq}$ [5]. His algebraic type system is built on arbitrary monotypes. Polymorphism is handled in extension, which make is possible to factor a bigger part of the soundness proof than in our system. However, this makes the framework less general than ours, where potential polymorphism is handled in concrete instances. Frey only formalized

a closed-world form of type-checking. In the discussion part, typechecking under the open-world assumption is proposed as typechecking in all possible extensions. No practical definition is included, since that could only be done for specific instances of the system. Frey's language includes local multi-methods, defined inside expressions, as in [5]. This is more expressive than our language, where multi-methods only appear at toplevel. On the other hand, this forces multi-methods to be treated as part of the core language, while we handle them separately as user-defined operators. It is not clear how local multi-methods interact with modular programming and the open-world assumption. In particular, is it possible to add implementations to the local methods of an imported module? If so, would such an implementation be able to access the variables defined in the scope of the local method? If implementations cannot be added to imported local methods, what specific method coverage rules can guarantee that the method will still be covered in the whole program? This last problem is similar to the question of guaranteeing that imported (toplevel) methods are always covered, which is discussed in Section 15.2.

15.2 Modular multi-methods

Multi-methods first appeared in dynamically typed languages – CLOS and later Cecil [14]. While this tainted multi-methods as "powerful but unsafe", it also opened up a line of research for devising static type systems for them. In particular, a full proposal for a statically typed language with multi-methods and modules can be found in [16], with the motivation of adding optional static typechecking to Cecil. However, the need to perform coverage and non-ambiguity checks for multimethods based on knowledge for the whole program has often been seen as a failure to achieve modular typechecking. In this view, modularity is defined as the guarantee that when modules that have been checked independently — but possibly never imported together in a single module, even in compiled form — are linked together in the same program, this whole program is guaranteed to be type-safe. This is indeed impossible to achieve with unrestricted multimethods as found in [16]. Several trade-offs are possible between fully modular typechecking and full expressiveness of multimethods [33, 35]. This approach has been implemented with MultiJava [18], a practical language that extends Java with a restricted form of multimethods that guarantees modular typechecking. The first restriction is that external methods cannot be declared abstract (that is, lack a default implementation). The second restriction is that multimethod implementations must either be written in the same module as the method's declaration, or it must be declared inside the class declaration of its first argument. As a compromise to overcome those limitations, Relaxed MultiJava [34] is an extension of MultiJava in which those restrictions are transformed into compile-time warnings instead of errors. During class loading, immediately prior to execution, the additional coverage and non-ambiguity tests are performed to detect whether errors can occur.

Our system supports the general form of multimethods without such restrictions[1]. In Section 14.4, we argue that our system is indeed modular. The key is that the main module of the program has knowledge of the *compiled interface* of all modules composing the program, and that this information is sufficient to perform all necessary checks. Furthermore, when method coverage fails, it is always possible to provide additional method implementations that will make it succeed. Since it is never necessary to modify imported modules, the modularity requirement is met. This argument applies to the case where the program can be compiled together before its start. A different situation arises when modules — known in this context as plugins — can be linked with a running program. It is then obviously too late for a programmer to add missing method implementations. In this context, we think that the technique presented in [34] of compile-time warnings combined with runtime checking is very promising, especially as the linking of plugins can in any case fail for other reasons. Alternatively, if more static guarantees are desired, some form of restricting expressivity could be used. It would in particular be interesting to investigate whether no restrictions at all can be imposed on the main program, since it can be supposed to be known by all plugins.

[1] Performing coverage checking in each module and enforcing the precocity rule, as defined in Section 8.3, can be seen as a form of restriction compared to solely performing coverage checking on the whole program. However, those rules are only useful to help detect errors earlier and to enforce what we believe is a good organization of the program, from a software engineering point of view. They are optional, and the type safety and modularity results do not depend on their presence.

A similar problem arises if one wants to be able to declare methods private to a certain module. In an unrestricted setting, the declaration of new subclasses might render the implementations of a method declared in an imported module either incomplete or ambiguous. Although, as said above, this situation could be handled by adding additional implementations of the method, this is most likely unacceptable when the method was private to the imported module, since its existence should in that case not be relevant to client modules. Therefore, in this situation as well, we think it would be beneficial to combine our system with a form of restriction proposed in [33] for non-public methods.

15.3 Kinds

F-bounded polymorphism [8] has been introduced to extend the record-based structural approach to typing object-oriented languages. It allows to type binary methods, at the cost of preventing subclasses to be subtypes. This makes it difficult to compare with our proposal which both guarantees subclasses to be subtypes and at the same time accepts, for instance, the `plus` method, which is more complex than a binary method since it is partially polymorphic. Using F-bounded quantification in conjunction with multi-methods has been proposed in [32], but it is still an open area of research, in particular with respect to soundness and decidability.

If we do not consider programming in such record-based language, but focus on the types that can be expressed with F-bounded polymorphism, we believe that it is possible to encode kinding constraints using F-bounded quantification, translating a kind K to a parameterized class K<T> and a kinding constraint $T : K$ into $T \leq K$<T>. This is similar to the *framework* syntactic sugar proposed in [32]. Type-checking in System F-bounded is also known to be undecidable [3]. This does not preclude of the decidability of F-bounded quantification in nominal type systems, as found in Pizza [37] or Generic Java [7], but none of these systems have been proved decidable yet. Furthermore, their type systems cannot handle partially polymorphic methods using the above encoding, since they prevent a class to implement the same interface twice with different type parameters. Our proposal does not require recursive constraints; complex constraints can always be deconstructed into atomic constraints, which simplifies decidability and efficient type-checking. This also makes quantification over type constructors straightforward, which is crucial for parameterized types. For instance, given the kind `Collection<T>`, one can give `map` the type $\forall$`C: Collection, T, U.` (`C<T>`, `T` $\rightarrow$ `U`) $\rightarrow$ `C<U>`, which allows `C` to range over type constructors of kind `Collection`.

The Abel language [9] has a type system based on [10] that can model object-orientation using kinds and polymorphic recursive types. These kinds are defined by $K ::= \mathsf{Type} \mid K \Rightarrow K \mid \mathsf{POWER}[T]$, where T is a type. Type T_1 has kind $\mathsf{POWER}[T_2]$ in fact means that T_1 is a subtype of T_2. In Abel, one can therefore simulate bounded polymorphism by kinding the type variable with a POWER kind. Together with recursive types, this allows for the same solutions as in System F-bounded for the situations presented in this paper, but with the same problems. Our kinds are very different, since they are generative names, and do not enforce transitivity. It is essential for our solution that $T : K$ and $T' \leq T$ does not imply $T' : K$. It might be possible to extend power kinds to relax transitivity, but to our knowledge it has not been done yet.

Type-classes [42] address the issue of homogeneous functions by defining predicates on types. For instance, the following Haskell-like code

```
class Eq t where
  == :: t -> t -> Bool

class Num t where
  + : t -> t -> t

instance Num Int where
  == x y = intEq x y
  + x y  = intAdd x y
```

can be expressed with kinds in the following way:

```
kind Eq
== :: <t:Eq> t -> t -> Bool

kind Num extends Eq
+ :: <t:Num> t -> t -> t

class Int implements Num
== @Int @Int = intEq
+  @Int @Int = intAdd
```

An important difference is that our kinds are open: it is not required to define operations syntactically together with the kind they operate on. This allows for modular definition of orthogonal operations, and operations that operate on more than one kind. Additionally, type-classes are not mixed with subtype polymorphism. Therefore they do no raise the question of the interactions between subtyping and kinding as found in partially polymorphic functions. On the other hand, the possibility to define type-classes inductively does not have an equivalent in our proposal. This feature is certainly useful, and should be considered in an extension of our system.

Chapter 16

Conclusion

We have given a modular presentation of a complete language and type system. This structure made it possible to consider independently extensions of the language and of the type system. This possibility is useful when presenting a complex system, since one does not need to consider and prove the whole system at once. Furthermore, it helps researching extensions of the system without redoing the whole work.

In particular, we make a presentation of object-orientation with multi-methods as an extension of a core, the traditional lambda-calculus with constants. The proof of type-safety for this extension could be made by only proving subject-reduction and progress lemmas for multi-methods themselves.

We formalized a module system for our language with multi-methods. We showed how typechecking can be performed modularly, without limiting the expressiveness of (public) multi-methods.

In parallel, we showed how two existing type system, Hindley-Milner and $ML_{\leq}$, fit into our framework. We also extended the $ML_{\leq}$ type system, by motivating and formalizing the introduction of kinds and kinding constraints for typing homogeneous or partially polymorphic methods.

Finally, we formalized some aspects of the translation of our system into practice, in particular the compilation of our high-level language into a monomorphically-typed bytecode language, and the algorithms needed to implement constraint implication in the presence of kinds. We also presented how we chose to design the Nice language to implement these ideas.

Experience using Nice has brought up new challenges that can motivate extensions of our work. Some of them have been sketched in this dissertation, like the type-safe handling of null values in Section 13.2.1. In practice, a difficulty arises from the fact that imported Java methods do not have these nullness type information. The user can explicitly retype such methods to add that information, provided it can be inferred from the documentation of the method. In the absence of such retyping, the compiler has to resort to either a safe bet (possibly null return types, but non-null method arguments), which is likely too restrictive for practical use because it results in many false typing errors, or a more lenient default that does not guarantee nullness safety across imported method calls. It would be interesting to investigate how much of this nullness information could be inferred by a static analysis of imported method code.

Another challenge is the support of static method overloading, in particular while preserving type inference. This feature allows several unrelated methods to have the same name and the same number of arguments[1]. Note that the disambiguation is performed at compile-time based on the static types of the arguments, which makes this feature orthogonal to runtime dispatch. For instance, suppose there are two methods `foo` of types `int` → `void` and `String` → `void`, and consider an expression of the form λ`x`....`foo`(`x`).... If the part of the expression preceding the call `foo`(`x`) does not further constraint `x`, it is not possible to decide at that point what is the expected type of `x` and which method `foo` is called. A simple solution is to report an ambiguity error, which can be solved by adding a type annotation to the declaration of `x`. However, it would be interesting to consider other solutions that can resolve the ambiguity when it is possible based

[1] This is of practical important for Nice, since Java does have static method overloading, and therefore the compiler has to deal with this situation at least for imported methods. For this reason, it was also natural to allow static overloading for Nice multi-methods.

on the rest of the expression and the context in which the whole expression is used, and to see how these solutions can be fit into our presentation.

A fundamental property of the $ML_{\leq}$ type system is that related type constructors have the same variance. In practice, this entails that subclasses must have the same number of type parameters as their parents[2]. This turns out to be restrictive in practice. For instance, one could want to assert that the type `Integer` is a subtype of `List` $<$ `Boolean` $>$ by considering integers as a list of the bits in their binary representation. Conversely, it could also be useful to introduce in a subclass a new type parameter. We believe it might be possible to extend $ML_{\leq}$ to support such situations by keeping track of the conditions on type parameters for type constructor orderings to hold. These conditions would become additional premises in the the MIntro rule. Additionally, constructed monotypes should in general contain open lists of type parameters to accommodate for variable type constructors, as introduced in rule VElim.

Closer to our original topic, our modular typechecking system for multi-methods does not apply directly to non-public and "plugin" multi-methods. In those situation, it should be investigated how to integrate other works on modular multi-methods with stricter modularity criteria, as discussed in Section 15.2. This aspect is also of practical importance for the Nice language.

[2] Thankfully, this does not rule parametrized classes out since there is no need for a root ancestor common to all classes.

Conclusion

Nous avons effectué une présentation modulaire d'un langage complet et de son système de types. Cette organisation nous a permis de considérer indépendemment les extensions du langage et du système de types. Cette possibilité est utile pour la présentation d'un système complexe, puisque cela permet de ne avoir à faire la présentation et la preuve de correction du système en un seul bloc, les rendant plus digestes. De plus, cela premet de rechercher des extensions du système sans refaire tout le travail.

En particulier, nous avons présenté l'orientation objet avec multi-méthodes comme une extension d'un noyau bien connu, le lambda-calcul avec constantes. La preuve de correction de cette extesion a pu être faire en prouvant simplement l'auto-réduction et le lemme de progrès pour les multi-méthodes elles-mêmes.

Nous avons formalisé un système de modules pour notre langage à multi-méthodes, et nous avons montré comment le typage peut être effectué modulairement, sans limiter l'expressivité des multi-méthodes (tout du moins des multi-méthodes publiques).

Parallelement, nous avons montré comment deux systèmes de types existants, Hindley-Milner et $ML_{\leq}$, peuvent être intégrés à notre système. Nous avons aussi étendu $ML_{\leq}$ en motivant et formalisant l'introduction de kinds et de contraintes de kinding pour typer les méthodes polymorphes homogènes et le méthodes partiellement polymorphes.

Enfin, nous avons formalisé certains aspects de l'implémentation de notre système, en particulier la compilation de notre langage de haut niveau vers un langage bytecode typé monomorphe, et les algorithmes nécessaires à l'implémentation des kinds. Nous avons aussi présenté les choix de conception du langage Nice.

L'utilisation de Nice a permis de découvrir de nouveaux défis qui peuvent motiver des extensions de notre travail. Certains ont été esquissés dans cette dissertation, comme le traitement statiquement sûr des valeurs nulles dans la section 13.2.1. En pratique, une difficuté supplémentaire résulte du fait que les méthodes Java importées ne fournissent pas d'information sur leur traitement des valeurs nulles. L'utilisateur de Nice peut explicitement "retyper" ces méthodes pour ajouter cette information quand elle peut être devinée grâce aux commentaires de documentation. En l'absence de tels retypages, le compilateur doit soit supposer le cas le plus restrictif (type de retour possiblement nul, mais arguments non-nuls) pour découvrir toutes les erreurs potentielles, mais aussi signaler de nombreuses fausses alertes au point d'être inutilisable, soit être plus accomodant, perdant alors la guarantie de sûreté des valeurs nulles lors des appels de méthodes importées non-retypées. Une alternative interessante à explorer serait d'inférer cette information à partir d'une analyse statique du code des méthodes importées.

Un autre défi consiste à permetre la surcharge statique des méthode, en particulier tout en conservant l'inférence de types. Cette fonctionalité permet à des méthodes sans rapports d'avoir le même nom et nombre d'arguments[3]. Remarquons que la désambiguation est effectuée à la compilation sur la base des types statiques des arguments, ce qui rend cette fonctionalité orthogonale au dispatch dynamique. Par example, supposons avoir deux méthodes `foo` des types `int` → `void` et `String` → `void`, et considérons une expression de la forme λ`x`....`foo`(`x`).... Si la partie de l'expression précédant l'appel `foo`(`x`) ne contraint pas `x`, il n'est pas possible de décider à cet endroit du type de `x` et de quelle méthode `foo` est appelée. Une solution simple est de signaler une erreur d'ambiguité, qui peut être résolue par le programmeur en ajoutant une annotation de type à la déclaration de `x`. Toutefois, il serait intéressant de considérer d'autres solutions

[3] Ceci est important en pratique pour Nice, puisque Java inclus la surcharge statique. Le compilateur Nice doit donc traiter ce cas, au moins pour les méthodes importées. Il était donc naturel de permettre la surcharge statique pour les méthodes Nice aussi.

qui pourraient résoudre l'ambiguité quand c'est possible en utilisant le reste de l'expression et le contexte dans lequel l'expression entière est utilisé, et de voir comment ces solutions peuvent être intégrées à notre système.

Une propriété fondamentale du système de types $ML_{\leq}$ est que les constructeurs de types d'une même hierarchie ont la même variance. En pratique, cela implique que les sous-classes doivent avoir le même nombre de paramètres de types que leurs parents[4]. Cela est limitant en pratique. Par example, on peut vouloir indiquer que le type `Integer` est un sous-type de `List` $<$ `Boolean` $>$ en considérant la représentation binaire des entiers. Inversement, il peut aussi être utile d'introduire une nouveau paramètre de type dans une sous-classe. Nous pensons qu'il est possible d'étendre $ML_{\leq}$ pour permettre ces situations en accumulant les conditions sur les paramètres de types sous lesquelles un constructeur de types est plus petit qu'un autre. Ces conditions deviendraient des prémisses suplémentaires dans la règle MIntro. De plus, les monotypes construits devraient contenir des listes ouverts de paramètres de types pour gérer les variables de constructeurs de types introduits par la règle VElim.

Plus près de notre sujet initial, notre système de types modulaire pour les multi-méthodes ne s'applique pas directement au méthodes non publiques ou présentes dans des "plugins". Pour ces situations, il serait utile d'intégrer d'autres travaux sur les multi-méthodes modulaires ayant des critères de modularité plus stricts, comme nous l'avons argumenté dans la section 15.2. Cet aspect a une importance pratique pour le langage Nice.

[4] Heureusement, cela n'empêche pas les classes paramètrées d'exister puisqu'il n'y pas pas besoin d'y avoir une classe racine ancêtre de toutes les autres.

Bibliography

[1] M. Abadi, L. Cardelli, B. C. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995. Also appeared as SRC Research Report 120. Preliminary version appeared in the Proceedings of the ACM SigPlan Workshop on ML and its Applications, June 1992.

[2] R. Agrawal, L. G. DeMichiel, and B. G. Lindsay. Static Type Checking of Multi-Methods. In *Proceedings of the OOPSLA '91 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 113–128, 1991.

[3] P. Baldan, G. Ghelli, and A. Raffaeta. Basic theory of F-bounded quantification. *Information and Computation*, 153(1):173–237, 1999.

[4] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common lisp object system specification x3j13. *SIGPLAN Notices*, 23(Special Issue), September 1998.

[5] F. Bourdoncle and S. Merz. On the integration of functional programming, class-based object-oriented programming, and multi-methods. Research Report 26, Centre de Mathématiques Appliquées, Ecole des Mines de Paris, Paris, Mar. 1996.

[6] F. Bourdoncle and S. Merz. Type-checking higher-order polymorphic multi-methods. In *Conference Record of the 24th Annual ACM Symposium on Principles of Programming Languages*, pages 302–315, Paris, Jan. 1997. ACM.

[7] G. Bracha, M. Odersky, D. Soutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of OOPSLA*, 1998.

[8] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In ACM, editor, *Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[9] P. Canning, W. Hill, and W. Olthoff. A kernel language for object-oriented programming. Technical Report STL-88-21, Hewlett-Packard Labs, 1988.

[10] L. Cardelli. Structural subtyping and the notion of power type. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 70–79, San Diego, California, 1988.

[11] G. Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.

[12] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. In *ACM Conference on LISP and Functional Programming*, pages 182–192, 1992. Extended and revised version in *Information and Computation* 117(1):115-135, 1995.

[13] C. Chambers. Object-oriented multi-methods in Cecil. In O. L. Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 33–56, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.

[14] C. Chambers. The Cecil language: Specification and rationale, version 2.1. Technical report, Department of Computer Science and Engineering University of Washington, Box 352350, Seattle, Washington 98195-2350 USA, March 1997.

[15] C. Chambers and W. Chen. Efficient multiple and predicate dispatching. In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA '99)*, volume 34(10) of *ACM SIGPLAN Notices*, pages 238–255, Denver, CO, November 1999. ACM.

[16] C. Chambers and G. T. Leavens. Typechecking and modules for multi-methods. In *ACM Transactions on Programming Languages (TOPLAS)*, volume 17(9). ACM, November 1995.

[17] W. Chen, V. Turau, and W. Klas. Efficient dynamic look-up strategy for multi-methods. In M. Tokoro and R. Pareschi, editors, *ECOOP '94, European Conference on Object-Oriented Programming, Bologna, Italy*, volume 821 of *Lecture Notes in Computer Science*, pages 408–431, New York, N.Y., July 1994. Springer-Verlag.

[18] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10), pages 130–145, 2000.

[19] W. R. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag, New York, N.Y., 1991.

[20] C. Dutchyn, P. Lu, D. Szafron, S. Bromling, and W. Holst. Multi-dispatch in the Java virtual machine: Design and implementation. In *Proceedings of 6th Usenix Conference on Object-Oriented Technologies and Systems (COOTS'2001)*, San Antonio, USA, January 2001.

[21] Eastern Research Apple Computer and Technology. Dylan, an object-oriented dynamic language. Apple Computer, Inc., April 1992.

[22] A. Frey. *Approche algébrique du typage d'un langage à la ML avec objets, sous-typage et multi-méthodes.* PhD thesis, École des Mines de Paris, 2005.

[23] R. P. Gabriel, J. L. White, and D. G. Bobrow. CLOS: Integrating object-oriented and functional programming. *Communications of the ACM*, 34(9):28–38, September 1991.

[24] J. Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, Sasaguri, Japan, November 2000.

[25] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Sunsoft Java Series. Addison Wesley Developers Press, second edition, 2000.

[26] U. Hölzle. Integrating independently-developed components in object-oriented languages. In *ECOOP '93*, pages 36–56, 1993.

[27] S. Krishnamurthi, M. Felleisen, and D. P. Friedman. Synthesizing object-oriented and functional design to promote re-use. *Lecture Notes in Computer Science*, 1445:91–??, 1998.

[28] C. Lécluse and P. Richard. The O2 database programming language. In P. M. G. Apers and G. Wiederhold, editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases, Amsterdam, The Netherlands*, pages 411–422. Morgan Kaufmann, August 22-25 1989.

[29] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system, documentation and user's manual - release 3.05. Technical report, INRIA, July 2002. Documentation distributed with the Objective Caml system.

[30] X. Leroy and M. Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

[31] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, second edition.* Addison-Wesley, 1999.

[32] V. Litvinov. Constraint-based polymorphism in Cecil. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Aplications*, volume 33(10), pages 388–411, Vancouver, Canada, October 1998.

[33] T. Millstein and C. Chambers. Modular statically typed multimethods. In *Proceedings of the Thirteenth European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 279–303, Lisbon, Portugal, June 1999. Springer Verlag.

[34] T. Millstein, M. Reay, and C. Chambers. Relaxed MultiJava: balancing extensibility and modular typechecking. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 224–240. ACM Press, 2003.

[35] T. D. Millstein. *Reconciling software extensibility with modular program reasoning.* PhD thesis, Department of Computer Science and Engineering, University of Washington, October 2003.

[36] P. Naur, J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language algol 60. *Commun. ACM*, 3(5):299–314, 1960.

[37] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of the 24th Annual ACM Symposium on Principles of Programming Languages*, pages 146–159, Paris, Jan. 1997. ACM.

[38] M. Odersky and M. Zenger. Independently extensible solutions to the expression problem. In *Proc. FOOL 12*, Jan. 2005. `http://homepages.inf.ed.ac.uk/wadler/fool`.

[39] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC*, pages 9–15, 19–21 1998.

[40] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[41] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, 1991.

[42] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Principles of Programming Languages*, Jan 89.

[43] M. Wand and P. O'Keefe. On the complexity of type inference with coercion. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 293–298. ACM Press, 1989.

[44] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 241–252. ACM Press, 2001.