On the integration of functional programming, class-based object-oriented programming, and multi-methods

François Bourdoncle

Centre de Mathématiques Appliquées,
Ecole des Mines de Paris,
60, boulevard Saint-Michel,
F-75006 Paris, France

`bourdoncle@cma.cma.fr`

Stephan Merz

Institut für Informatik,
Technische Universität München,
Arcisstraße 21,
D-80290 München, Germany

`merz@informatik.tu-muenchen.de`

October 8, 1996

Abstract

We present a new predicative and decidable type system, called $\mathrm{ML}_{\leq}$, suitable for object-oriented languages with implicit polymorphism in the tradition of ML (cf. Hindley [25] and Milner [32]). Instead of using extensible records as a foundation for object-oriented extensions of functional languages, we propose to reinterpret classical datatype declarations as abstract and concrete class declarations, and to replace pattern-matching on run-time values by dynamic dispatch on run-time types. $\mathrm{ML}_{\leq}$ is based on universally quantified polymorphic constrained types, where constraints are conjunctions of structural inequalities between monotypes built from extensible and partially ordered classes of type constructors. We show how this type system can be used to design programming languages retaining much of the ML spirit while integrating in a seamless fashion higher-order and class-based object-oriented programming, dynamic dispatch on several arguments, and parametric polymorphism. We give type-checking rules for a small, explicitly typed functional language with methods, and show that the resulting system has decidable minimal typing. We discuss type inference for this language. We then define a strict operational semantics, prove subject reduction, and show how abstraction and encapsulation can be achieved by proper use of a module system. We present a prototype implementation of this type system and discuss algorithmic and implementation issues. In particular, we give a type-checking algorithm which is exponential in the worst case but is expected to be polynomial in practice. We conclude by a comparison with other similar type systems in the literature, including ad-hoc polymorphism, dynamics, systems for performing type inference in the presence of primitive subtyping, as well as impredicative systems like $\mathrm{F}_{\leq}$.

Contents

List of Figures

Chapter 1

Introduction

Two of the most important programming paradigms today are functional programming, putting the emphasis on higher-order functions, and object-oriented programming, emphasizing data encapsulation and dynamic dispatch. Whereas the first paradigm is extremely well understood theoretically, and has led to the development of programming languages with very clean semantics, the second paradigm seems to have gained much more acceptance among practitioners. Indeed, properly written object-oriented programs are usually much easier to extend and maintain than their functional counterparts. However, the expressive power of higher-order functions can lead to very concise and clear programs, and object-oriented languages would clearly benefit from such a concept.

So far, most efforts to fill the gap between the two paradigms have been put into the so-called "objects-as-records" model or into specific object calculi [2], leading to powerful systems exhibiting most of what are considered valuable object-oriented features, in particular extensibility and data encapsulation. However, these systems often require a second-order formalism [16] or recursive types, usually fail to consider methods as first-class objects, thus preventing them to be passed as arguments to other methods or functions, and cannot deal easily with binary methods [7] without resorting to the notion of matching [6] in addition to the notion of subtyping, which can be somewhat confusing. On the other hand, recent work on the alternative "methods-as-overloaded-functions" model [14, 17] has exemplified the possibility of having powerful type systems for object-oriented languages where methods are defined outside the scope of objects and are just sets of functions dispatching on the type of all their input arguments simultaneously. However, these systems are either first-order and monomorphic [14], second-order impredicative systems dealing with explicit polymorphism [10] or extend the classical higher-order predicative model of implicit polymorphism [1, 3, 18, 19, 20, 26, 30, 39, 40], but resort to ad-hoc polymorphism or dynamics to emulate methods.

This report is an attempt to show that it is possible to design a strongly typed, higher-order object-oriented language with a polymorphic, predicative and decidable type system by only slight modifications to languages in the tradition of ML, and without resorting to "dirty" ad-hoc polymorphism (that is, overloading mechanisms allowing the same function to work non-uniformly on totally unrelated types). Our key ideas are 1) to clearly separate specification and implementation to allow extensibility and scalability, 2) to use a module system to provide data encapsulation, 3) to add primitive subtyping to allow the definition of class hierarchies, and 4) to replace pattern-matching on run-time values by dynamic dispatch on run-time types. Basically, we argue that an ML declaration of the form

$$\mathsf{datatype\ list}[\alpha] = \mathsf{nil} \mid \mathsf{cons\ of}\ \alpha * \mathsf{list}[\alpha]$$

should be split into a specification part and an implementation part, as illustrated by figure 1.1. The specification part, which takes the form of an interface, declares the existence of a *type constructor* list (i.e., an abstract parameterized class, in OO parlance), of a *data type constructor* nil (i.e., a concrete parameterized class), which is a subconstructor of list with an empty record implementation, as well as the existence of explicitly typed methods attached to objects of this type. These type declarations specify both the run-time behavior of methods and the existence of actual implementations [4].

The implementation part, which takes the form of a module, declares a data type constructor cons as a subconstructor of list. Since we are in the presence of subtyping, we must talk about the *variance* of type constructors explicitly. To this end, we define the *type constructor class* List, in a way similar to that of Gofer and Haskell [24, 27], as a set of covariant type constructors with arity one, and we explicitly declare that nil, cons, and list are members of List. Of course, the fields of data types must comply with the explicitly declared variance of their type parameters, which is the case for cons here, since α is covariant both in $1\colon \alpha$ and in $2\colon \mathsf{list}[\alpha]$. Moreover, in order to avoid dealing with implementation inheritance (which, we believe, is a syntactic notion), we impose that data type constructors have no subconstructors. Therefore, data types represent sets of records with the same "tag". This approach differs from that of ML where "tags" are run-time values which are not reflected in the type system. For instance, a ML type like list[int] denotes the set of empty and non-empty lists of integers, whereas a $\mathrm{ML}_{\leq}$ type like cons[int] denotes the set of non-empty lists of integers. We shall see that whereas ML-style pattern matching is performed on run-time *values*, $\mathrm{ML}_{\leq}$ performs dynamic dispatch on run-time *types*, which express *global properties* of run-time values. As a consequence, dispatching on the fact that a value has type list[int] v.s. the fact that the value has type list[real] is something which can be imagined, even if the current axiomatization of the system does not allow it. Moreover, having types like cons[int] allows data extractors to be total functions over their domain. For instance, the extractor associated to the second field of a cons has type

$$\forall\alpha.\ \mathsf{cons}[\alpha] \to \mathsf{list}[\alpha]$$

in $\mathrm{ML}_{\leq}$, as opposed to

$$\forall\alpha.\ \mathsf{list}[\alpha] \to \mathsf{list}[\alpha]$$

in ML. Of course, it is always possible to hide type constructors nil and cons in the implementation module and only export type constructor list and boolean-valued functions like *isNil* and *isCons*.

Note that the implementation of method *cons* in module *List* is correct w.r.t. its specification since it has type $\forall\alpha.\ \langle\alpha, \mathsf{list}[\alpha]\rangle \to \mathsf{cons}[\alpha]$, which is a subtype of the type (or specification) of *cons*. Also, note that since cons is not exported in the interface, inferring the type of the method from the type of one of its implementations does not really make sense. Finally, note that the list of methods declared in the interface is not exhaustive, as opposed to the original presentation of type classes [44] (later relaxed in System O

```
interface List is
  class List[⊕];
  type list: List;
  data nil: List;
  order nil ⊏ list;
  data nil[α] is end;
  meth cons: ∀α. ⟨α, list[α]⟩ → list[α];
  meth head: ∀α. list[α] → α;
  meth tail: ∀α. list[α] → list[α];
  meth size: ∀α. list[α] → int
end List;

module List is
  data cons: List;
  order cons ⊏ list;
  data cons[α] is 1: α; 2: list[α] end;
  meth cons(x: _, l: _) = cons x l;
  meth head(l: cons) = cons.1 l;
  meth tail(l: cons) = cons.2 l;
  meth size(l: nil) = 0;
  meth size(l: cons) = 1 + (size (cons.2 l))
end List;

module SList is
  open List;
  type slist: List;
  data scons: List;
  order slist ⊏ list;
  order nil ⊏ slist;
  order scons ⊏ slist;
  data scons[α] is 1: α; 2: slist[α]; 3: int end;
  meth cons(x: _, l: slist) = scons x l (1 + (size l));
  meth head(l: scons) = scons.1 l;
  meth tail(l: scons) = scons.2 l;
  meth size(l: scons) = scons.3 l
end SList;
```

Figure 1.1: Lists and sized lists

[39]). New methods, for instance "private" methods, can thus be defined as needed in any interface or implementation module, and, as in ML, passed as arguments to other functions.

Now, suppose that the problem at hand requires computing the size of lists very frequently, so that we would rather use explicitly sized lists, and that in order to reuse some library code, we want to consider a sized list as just another sort of list. Then one possibility, implemented in module *SList* of figure 1.1, is to define a constructor slist between nil and list, a data type constructor scons of List as a subconstructor of slist, and to refine method *cons* so that it builds sized lists whenever its second argument is a sized list (which can either be the empty list or a sized cons). This way, all lists built using the *cons* method, and in particular, lists built by library functions, will be sized lists. Of course, it is still possible to build regular conses, but this is now only possible in the *List* module, or via a function exported by *List*. Note that had data type constructor nil been abstracted in the interface, a new implementation for empty sized lists with no relationship whatsoever with nil should have been defined in module *SList*.

Primitive subtyping provides great expressive power. Indeed, we only require that the ordering between type constructors of a given class be a partial order, and we could imagine having a numeric class Num with the following "mathematical" ordering $\mathsf{neg}, \mathsf{zero}, \mathsf{pos} \sqsubset$

$\mathsf{int} \sqsubset \mathsf{real}, \mathsf{dyadic}$, where neg, zero and pos denote the sets of negative, null and positive integers, int denotes the set of all integers, real denotes the set of reals, and dyadic denotes the set of 2-adic numbers, which have recently been advocated as a foundation for the design of synchronous digital circuits [43]. However, this power ordinarily comes at a price: the loss of principal typing. Indeed, suppose function *twice* has type $\forall\alpha.\ (\alpha \to \alpha) \to (\alpha \to \alpha)$ and function f has type $\mathsf{real} \to \mathsf{int}$. Then the expression (*twice* f) has types $\mathsf{real} \to \mathsf{real}$ and $\mathsf{int} \to \mathsf{int}$, none of which is better than the other.

To solve this problem, we propose to enrich the Hindley-Milner predicative type system for implicit polymorphism with the notion of *polymorphic constrained type* of the form $\forall\vartheta\colon \kappa.\ \theta$, where ϑ is a set of variables, κ is a constraint, and θ is a monotype. Constrained polymorphic types have been proposed by some authors, in particular to accommodate overloading in various ways [19, 20, 29, 39] or to perform type inference for object-oriented languages or languages with primitive subtyping [5, 21, 22, 23, 26, 29, 33, 34, 35, 41, 42]. Our system is original in that it merges the two approaches: methods are, in a sense, "cleanly overloaded functions", and their polymorphic constrained types have constraints which are conjunctions of inequalities between monotypes. As we shall see, this modification leads to a clean notion of *functional type application* and ensures decidable minimal typing. For instance, the type of the above expression would be

$$\forall\alpha\colon \mathsf{int} \le \alpha \le \mathsf{real}.\ \alpha \to \alpha$$

that is, the expression has type $\alpha \to \alpha$ for any α between int and real. As a matter of fact, function *twice* can, as in [29], be given a better type

$$\forall\alpha,\beta\colon \beta \le \alpha.\ (\alpha \to \beta) \to (\alpha \to \beta)$$

which gives (*twice* f) the same minimal type as f.

Another advantage of polymorphic constrained types is that they naturally allow a precise typing of methods like the *move* method of figure 1.2. Intuitively, the *move* method takes an object with dynamic type α below point as input and returns an object of the same type α as output. The type of this method is thus $\forall\alpha\colon \alpha \le \mathsf{point}.\ \alpha \to \alpha$. Note that we have defined four type constructors: two type constructors point and cpoint, which play the role of abstract classes (that is, classes which only provide interfaces), and two data type constructors pt and cpt which play the role of concrete classes (that is, classes which provide implementations and can be instanciated at run-time). It is important to remark that data type constructors like cpt and pt have to be minimal (that is, cannot have subconstructors) to ensure that the two implementations of the method are well-typed. Indeed, if pt had a subconstructor c, then the first implementation of the method may accept an argument with dynamic type $\alpha = c$ and return an object with dynamic pt which is not below α. Also, remark that the absence of implementation inheritance implies that the first two fields of the two data types are unrelated, so that the code for methods operating on these two fields must be duplicated, as opposed to some systems based on extensible records [26]. This is not too much of a problem in a purely functional language, but can be problematic if data types have state. In chapter 8, we show how duplication of code can be greatly reduced by adding appropriate syntactic sugar. Note that the type of method *move* looks like a $\mathrm{F}_{\le}$ type, but this is only a

```
// Constructor class
class Point[];

// Type constructors
type point: Point;
type cpoint: Point;

// Data type constructors
data pt: Point;
data cpt: Point;

// Subtyping
order cpoint ⊏ point;
order pt ⊏ point;
order cpt ⊏ cpoint;

// Implementation of data types
data pt[] is
   1: real; 2: real
end;
data cpt[] is
   1: real; 2: real; 3: int
end;

// Method to move points functionally
meth move: ∀α: α ≤ point. α → α;
meth move(x: pt) =
        pt (inc (pt.1 x)) (inc (pt.2 x));
meth move(x: cpt) =
        cpt (inc (cpt.1 x)) (inc (cpt.2 x)) (cpt.3 x));
```

Figure 1.2: Points and colored points

formal similarity, since $\mathrm{ML}_{\leq}$ is a decidable predicative type system dealing with class hierarchies and implicit polymorphism, whereas $\mathrm{F}_{\leq}$ is an undecidable unpredicative type system dealing with explicit polymorphism. Also, note that the type of *move* could be compared to the specification that could be given in object-oriented languages allowing the like self or self_type type specifiers, namely

```
abstract class point is
   virtual method move(): like self;
end
```

However, as opposed to single-dispatch languages, polymorphic constrained types also allow for a very precise and natural typing of methods dispatching on *multiple* arguments. For instance, the subtraction operator[1] *sub* can be given the following type

$$\forall\alpha\colon \mathsf{int} \leq \alpha.\ \langle\alpha,\alpha\rangle \to \alpha$$

denoting any operation such that if there exists a numeric type α above int which is also above the dynamic types of the first two arguments of the operator, then the type of the result is below α. Consequently, reals and 2-adic numbers cannot be subtracted from one another (since there is no α above int, real, and dyadic), the subtraction of two integers is an integer, and the subtraction of two positive integers is an integer. Indeed, the minimal type of *sub* $\langle +1, +1\rangle$ is precisely

$$\forall\alpha\colon \mathsf{int} \leq \alpha \wedge \langle \mathsf{pos}, \mathsf{pos}\rangle \leq \langle\alpha,\alpha\rangle.\ \alpha$$

[1] See figure 5.3 for details.

which, as we shall see, is equivalent to

$$\forall\alpha\colon \mathsf{int} \leq \alpha \wedge \mathsf{pos} \leq \alpha \wedge \mathsf{pos} \leq \alpha.\ \alpha$$

which, since the constraint $\mathsf{pos} \leq \alpha$ is obviously redundant, is also equivalent to

$$\forall\alpha\colon \mathsf{int} \leq \alpha.\ \alpha$$

which, in turn, is equivalent to int, since int is the minimal solution of the constraint. In a way, *sub* is "constant" below $\langle\mathsf{int},\mathsf{int}\rangle$ and "polymorphic" above $\langle\mathsf{int},\mathsf{int}\rangle$. We can see that the type given to *sub* is much more precise than the type that can be given in systems without primitive subtyping where methods are just overloaded functions. Moreover, tricks as the $\mathsf{like\ self}$ type specifier do not work for methods dispatching on multiple arguments, precisely because there is no notion of self in such cases.

This report is organized as follows. In chapter 2, we fix some notations and definitions. In chapter 3, we introduce the notion of type structure and the notion of type constraint, and we prove decidability, soundness and completeness results about the universally quantified implication of existentially quantified constraints. In particular, we show that our axiomatization of constraint implication is invariant w.r.t. extensions of the type structure. In chapter 4, we introduce the syntactic notions of polymorphic constrained type and polymorphic constrained domain, and give a semantic interpretation. We introduce a decidable subtyping relation between universally quantified polymorphic constrained types, and show that functional types denote monotonic type transformers. In chapter 5, we introduce an explicitly typed applicative language with functions and methods, give typing rules for this language, and show that the resulting system has decidable minimal typing. In chapter 6, we give an operational semantics of the language and prove subject reduction. In chapter 7, we give an algorithm for constraint implication which has an exponential worst-case complexity, but is polynomial in practice. We also give indications on how to implement type-checking, and study how type inference may be performed for an untyped version of the language. In chapter 8, we show how standard object-oriented concepts may be emulated in $\mathrm{ML}_{\leq}$, we show how to design a module system for $\mathrm{ML}_{\leq}$ with modular type-checking, and we study extensions like multi-methods and type classes. Finally, we conclude in chapter 9 by comparing $\mathrm{ML}_{\leq}$ with other object-oriented type systems in the literature, including ad-hoc polymorphism, dynamics, systems for performing type inference in the presence of primitive subtyping, as well as impredicative systems like $\mathrm{F}_{\leq}$.

Chapter 2

Definitions and notations

Let us fix some notation and definitions. We assume the existence of a possibly infinite set of "names" such as Unit, Arrow, List, Bool, etc. A *type constructor class* C is a tuple $(N_C, T_C, D_C, \sqsubseteq_C, \partial_C)$ where N_C is a name, called the name of the class, T_C is a finite and non-empty set of elements t_C, called *type constructors*, $D_C \subseteq T_C$ is a set of elements d_C, called *data type constructors*, $\sqsubseteq_C$ is a partial order[1] on T_C such that every data type constructor d_C is minimal with respect to $\sqsubseteq_C$. The *variance* ∂_C of a class C is a tuple of elements of $\{\ominus, \oplus, \otimes\}$ denoting the variance of its type constructors in the context of subtyping. A class with variance ∂_C is said to be ∂_C-variant. The element $\oplus$ corresponds to covariant arguments, whereas $\ominus$ and $\otimes$ correspond to contravariant and non-variant arguments respectively. The length of ∂_C is called the *arity* of C, and by extension, the arity of type constructors $t_C \in T_C$. Type constructors with arity 0 are called *base type constructors*.

A *type structure* is a finite set $\mathcal{T}$ of type constructor classes such that any two distinct elements C_1, C_2 of $\mathcal{T}$ are such that N_{C_1} and N_{C_2} are distinct and T_{C_1} and T_{C_2} are disjoint. Each type constructor is thus associated to a unique class in $\mathcal{T}$ and each class has a unique name, so that we generally identify classes with their name. We assume that every type structure contains a ()-variant class Unit with at least one data type constructor unit, and a $(\ominus, \oplus)$-variant class Arrow with at least one data type constructor $\rightarrow$. In examples, we also assume a $(\oplus)$-variant class List with data type constructors nil and cons below type constructor list, and a ()-variant class Bool with data type constructors true and false below type constructor bool.

For every type structure $\mathcal{T}$, the set $\mathcal{G}_\mathcal{T}$ of *$\mathcal{T}$-ground monotypes* θ over $\mathcal{T}$ (or just *ground monotypes* θ in $\mathcal{G}$, when $\mathcal{T}$ is implicit) is the least set such that when t_C is a type constructor of a class C in $\mathcal{T}$ with arity n and $\theta_1, \ldots, \theta_n$ are ground monotypes over $\mathcal{T}$, then $t_C[\theta_1, \ldots, \theta_n]$ is a ground monotype.

A type structure can be seen as a collection of constructor classes and type constructors accessible in a given scope (e.g., an interface or a module). Since our goal is to model object-orientation, we must provide for the extension of type structures in new modules and interfaces. A type structure can be extended in two ways: by adding new classes or

[1]A partial order $\sqsubseteq$ is a reflexive ($x \sqsubseteq x$), transitive ($x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$) and antisymmetric ($x \sqsubseteq y$ and $y \sqsubseteq x$ implies $x = y$) binary relation.

C-constructors	ϕ_C	$::=$	$t_C \mid v_C$
Monotypes	θ	$::=$	$\phi_C[\Theta_C] \mid v$
Constraints	κ	$::=$	$\phi_C \sqsubseteq \phi_C \mid \theta \leq \theta \mid \kappa \wedge \kappa$
Variable sets	ϑ	$::=$	$\emptyset \mid v \mid v_C \mid \vartheta, \vartheta$
Types	τ	$::=$	$\forall \vartheta : \kappa.\, \theta$
Domains	δ	$::=$	$\exists \vartheta : \kappa.\, \theta$

Figure 2.1: Monotypes, constraints, types, and domains

by adding new type constructors to existing classes and extending the partial orderings of the extended classes. However, not all extensions are admissible. For instance, if two type constructors t_C and t'_C of a class C are not related by $\sqsubseteq_C$ in some interface, it is incorrect to add a type constructor t''_C to C such that $t_C \sqsubseteq_C t''_C$ and $t''_C \sqsubseteq_C t'_C$, since this would relate t_C and t'_C and enforce a property which is not present in the interface. We thus say that a type structure $\mathcal{T}^*$ is an *admissible extension* of a type structure $\mathcal{T}$ if for every class $C = (N_C, T_C, D_C, \sqsubseteq_C, \partial_C)$ in $\mathcal{T}$, there exists a class $C^* = (N_{C^*}, T_{C^*}, D_{C^*}, \sqsubseteq_{C^*}, \partial_{C^*})$ in $\mathcal{T}^*$ such that $N_{C^*} = N_C$, $T_{C^*} \supseteq T_C$ and $D_{C^*} \supseteq D_C$, $\partial_{C^*} = \partial_C$, and for all type constructors $t_1, t_2 \in T_C$, we have $t_1 \sqsubseteq_{C^*} t_2$ if and only if $t_1 \sqsubseteq_C t_2$. The overall requirement that data type constructors are minimal in any constructor class implies that $\mathcal{T}^*$ cannot define a subconstructor of any data type constructor defined in $\mathcal{T}$.

Assuming an implicit type structure $\mathcal{T}$, we define *monotypes*, *constraints*, *types* and *domains* by the grammar of figure 2.1. As usual, we write $\theta_1 \rightarrow \theta_2$ to denote $\rightarrow[\theta_1, \theta_2]$ and assume that the arrow is right-associative. We assume the existence of a collection of at least countable and pairwise disjoint sets of variables: a set of *type variables* v (representing ground monotypes) and, for each class C, a set of *C-constructor variables* v_C (representing type constructors of class C). However, in informal exposition, we use a universal set of variables α, β, γ, etc., and use the notation $\alpha \in C$ to denote that α is a C-constructor variable. Variables are bound by the universal and the existential quantifiers. We treat *variable sets* ϑ as sets of variables, and use the constant $\emptyset$ to denote the empty set. A *C-variable set* ϑ_C is a variable set of the form $v_1, \ldots, v_n$ where $v_1, \ldots, v_n$ are distinct type variables and n is the arity of C. A *C-monotype list* Θ_C is a list $\theta_1, \ldots, \theta_n$ where n is the arity of C. We denote by *true* the trivial constraint ($\mathsf{unit} \sqsubseteq \mathsf{unit}$), and by $\forall \vartheta.\, \theta$ the fully polymorphic type $\forall \vartheta : \mathit{true}.\, \theta$. We treat constraints as sets of conjuncts. Given a variable set ϑ and n syntactic terms X_1 and X_n, $n \geq 2$, we write $X_1 \# \ldots \# X_n \ [\vartheta]$ to denote that for any distinct i and j in $[1, n]$, every variable which is free both in X_i and in X_j belongs to ϑ. We simply write $X_1 \# \cdots \# X_n$ when ϑ is the empty set. We say that a syntactic term X is *ϑ-closed* for some variable set ϑ if the free variables of X are all in ϑ. For any syntactic terms X and Y, the meta-notation $X\{Y\}$ denotes the term X and indicates that Y is a subterm of X. We use the notation $(\theta = \theta')$ to denote the constraint $(\theta \leq \theta' \wedge \theta' \leq \theta)$. By abuse of notation, we frequently omit the brackets for base types of the form $t_C[\,]$.

We define the *standard ordering* $\leq$ as the smallest relation over $\mathcal{G}$ such that $t_C \sqsubseteq_C$

t'_C and $\Theta_C \leq_C \Theta'_C$ implies $t_C[\Theta_C] \leq t'_C[\Theta'_C]$, where the relation $\leq_C$ on C-monotype lists is defined as the componentwise ordering induced by the variance of C. In other words, assuming $\partial_C = (\partial^1_C, \ldots, \partial^n_C)$, we define $\theta_1, \ldots, \theta_n \leq_C \theta'_1, \ldots, \theta'_n$ as the constraint $\bigwedge_{i \in [1,n]} \theta_i \leq_{\partial^i_C} \theta'_i$, where $\theta \leq_\oplus \theta'$ is defined as $\theta \leq \theta'$, $\theta \leq_\ominus \theta'$ is defined as $\theta' \leq \theta$, and $\theta \leq_\otimes \theta'$ is defined as $\theta = \theta'$. Note that $\theta \leq t'_C[\Theta'_C]$ or $t'_C[\Theta'_C] \leq \theta$ implies that θ is of the form $t_C[\Theta_C]$, for some t_C and Θ_C. Moreover, $\theta \leq \theta'$ implies that the outermost type constructors of θ and θ' are of the same type constructor class.

A *ground substitution* (or $\mathcal{T}$-ground substitution, when $\mathcal{T}$ is explicit) is a total function mapping type variables to ground monotypes and C-constructor variables to type constructors in T_C. For any variable set ϑ, a *ϑ-substitution* $\sigma \in \mathcal{S}(\vartheta)$ is a total function mapping type variables to monotypes and C-constructor variables to C-constructors, such that the domain $\{x \mid \sigma(x) \neq x\}$ of σ is finite and disjoint from ϑ. In other words, σ maps variables to terms of the proper kind and is the identity for all variables in ϑ. For any variable sets ϑ, ϑ_1 and ϑ_2, a *renaming* $\sigma \in \mathcal{R}(\vartheta; \vartheta_1; \vartheta_2)$ is an injective ϑ-substitution mapping type variables to type variables and C-constructor variables to C-constructor variables, such that $\{\sigma(x) \mid x \in \vartheta_1\}$ is disjoint from ϑ_2. In other words, σ preserves variables in ϑ and renames variables in ϑ_1 with names which are not in ϑ_2. Note that $\mathcal{R}(\vartheta; \vartheta_1; \vartheta_2)$ is non-empty if and only if ϑ, ϑ_1 and ϑ_2 have an empty intersection, and that every renaming $\sigma \in \mathcal{R}(\vartheta; \vartheta_1; \vartheta_2)$ is a bijection whose inverse is a ϑ-substitution. For any substitution σ and syntactic term X, we denote by $X[\sigma]$ the syntactic term obtained by applying the substitution σ to X.

Chapter 3

Constraints

3.1 Informal introduction

As indicated in the introduction, universally quantified polymorphic constrained types (that we just call types, as opposed to monotypes) have a great expressive power. But this power comes at a price: many syntactically different types can have the same intended meaning. For instance, types like $\forall\alpha\colon \mathsf{int} \le \alpha \wedge \alpha \le \mathsf{int}.\ \alpha$ and $\forall\,\emptyset.\ \mathsf{int}$ are syntactically different but semantically identical. It is thus important that types which are intended to be the same be formally identified by the type system. We propose to perform this identification by defining a subtyping relation between types and saying that two types are semantically equivalent if they are subtypes of one another.

This approach of defining a subtyping relation between types is more in the tradition of impredicative systems like $F_{\le}$ than in the tradition of ML. Indeed, in predicative type systems, subtyping is generally implicitly defined in terms of instantiations, that is, a polytype is "below" all its ground instances. For instance, $\forall\alpha.\ \alpha \to \alpha$ is a subtype of $\mathsf{int} \to \mathsf{int}$ and $\mathsf{bool} \to \mathsf{bool}$. In other words, the intuitive denotation[1] of a polytype is its set of ground instances, and a polytype τ_1 is a subtype of another polytype τ_2 if and only if every ground instance of τ_2 is a ground instance of τ_1. As a consequence, ML types are equivalent if and only if they can be α-substituted.

We may be tempted to generalize this idea here by saying that the denotation of a type $\forall\vartheta : \kappa.\ \theta$ is its set of ground instances, that is, monotypes of the form $\theta[\sigma]$ for ground substitutions σ such that the constraint $\kappa[\sigma]$ holds w.r.t the standard ordering. This is indeed a good intuition, but it yields a definition of subtyping which is not compatible with the standard ordering on ground monotypes. As a simple example, consider the type $\tau = \forall\alpha\colon \mathsf{int} \le \alpha \wedge \alpha \le \mathsf{int}.\ \alpha$. The only ground instance that satisfies its constraint is int, and we would therefore expect τ to be equivalent to $\forall\,\emptyset.\ \mathsf{int}$ or simply int. In particular, we want τ to be a subtype of real, although real does not satisfy τ's constraint.

We argue that the intuitive denotation of a type should be defined as the *upper-closure* of its set of ground instances. The rationale is that if a type is a subtype all its ground instances, then by transitivity, it should also be a subtype all the supertypes of its ground

[1] Not to be confused with the classical ideal model of parametric polymorphism [32], where types are lower ideals, even in the absence of primitive subtyping.

instances.

Without primitive subtyping, this definition is equivalent to the standard ML notion, but in the presence of primitive subtyping, our definition ensures that types like $\forall\alpha\colon \mathsf{int} \leq \alpha.\ \alpha$ and $\forall\,\emptyset.\ \mathsf{int}$ are semantically equivalent, since they have the same denotation, namely all the supertypes of int. Moreover, defining a subtyping relation between types this way makes perfect sense in the context of object-orientation. For instance, if a function has a formal parameter with type $\mathsf{int} \to \mathsf{int}$, then actual parameters with type $\mathsf{real} \to \mathsf{int}$ or $\forall\alpha.\ \alpha \to \alpha$ are legal since both types are subtypes of $\mathsf{int} \to \mathsf{int}$. Indeed, if the formal is a function with type $\mathsf{real} \to \mathsf{int}$, then, in particular, it can be used as a function from integers to integers. Similarly, if the formal is a polymorphic function with type $\forall\alpha.\ \alpha \to \alpha$, meaning that for any α, in particular $\alpha = \mathsf{int}$, this function returns an α, then it can also be used as a function from integers to integers. Therefore, our subtyping relation matches the notion of "substitutivity" in object-oriented frameworks.

On the other hand, we require all quantifiers to occur at the outermost level; this makes polymorphic constrained types somewhat weaker than the types of an impredicative system like $\mathrm{F}_{\leq}$. In particular, $\mathrm{ML}_{\leq}$ does not provide a way to enforce that a formal parameter be a polymorphic function.

Intuitively, the universal quantifier can be interpreted as the greatest lower bound, so that, for instance, $\forall\alpha\colon \mathsf{int} \leq \alpha.\ \alpha$ can be read as "the (set of) smallest α above int", which is precisely int in this simple case.[2] With such an interpretation, it is natural to say that a type τ_1 of the form $\forall\vartheta_1\colon \kappa_1.\ \theta_1$ is a subtype of another type τ_2 of the form $\forall\vartheta_2\colon \kappa_2.\ \theta_2$ if and only if every ground instance of τ_2 is above at least one of the ground instances of τ_1. Formally, assuming an implicit type structure $\mathcal{T}$, and assuming that ϑ_1 and ϑ_2 are disjoint, we may say that τ_1 is a subtype of τ_2 if and only if for every ground substitution σ_2 such that $\kappa_2[\sigma_2]$ is a true ground constraint, there exists a ground substitution σ_1 that agrees with σ_2 on the variables of ϑ_2 such that $(\kappa_1 \wedge \theta_1 \leq \theta_2)[\sigma_1]$ is a true ground constraint. In other words, we want the following first-order implication to hold

$$\forall\,\vartheta_2.\,\kappa_2 \Longrightarrow (\exists\,\vartheta_1.\,\kappa_1 \wedge \theta_1 \leq \theta_2)$$

However, this requirement is imprecise without stating the intended universe for the bound variables. In the context of object-orientation, the "universe" of type constructors is open by essence, in order to allow the addition of new type and data type constructors in different modules. Nevertheless, a type-correct program should remain type-correct when the type structure is extended. Consequently, the notions of subtyping and constraint implication which are of interest to us must be invariant w.r.t. admissible extensions of the implicit type structure $\mathcal{T}$ in which type-checking is performed. For example, let us assume that $\mathcal{T}$ contains a ()-variant constructor class C with type constructors c_1, c_2, and c_3 partially ordered by $c_1 \sqsubseteq_C c_3$ and $c_2 \sqsubseteq_C c_3$. The problem is then whether a type τ like

$$\forall\alpha\colon c_1 \leq \alpha \wedge c_2 \leq \alpha.\ \alpha$$

[2]In general, types denote sets of ground monotypes, because we do not require type constructor classes or the set of ground monotypes to be lattices. In effect, we use sets to obtain a sup-semi lattice structure for types. This will be made more precise in chapter 4. As a matter of fact, we conjecture that the semantic denotation of a type can still be defined as the intersection of the ideal denotation of the elements of its intuitive, upper-closed denotation.

[*Approx*]	$\forall\vartheta.\ \kappa\{\kappa'\} \models \kappa'$	
[*CRef*]	$\forall\vartheta.\ \kappa \models \kappa \wedge \phi_C \sqsubseteq \phi_C$	
[*CTrans*]	$\forall\vartheta.\ \kappa\{\phi_C \sqsubseteq \phi'_C \sqsubseteq \phi''_C\} \models \kappa \wedge \phi_C \sqsubseteq \phi''_C$	
[*CTriv*]	$\forall\vartheta.\ \kappa \models \kappa \wedge t_C \sqsubseteq t'_C$	$(t_C \sqsubseteq_C t'_C)$
[*CMin*]	$\forall\vartheta.\ \kappa\{\phi_C \sqsubseteq d_C\} \models \kappa \wedge d_C \sqsubseteq \phi_C$	
[*VIntro*]	$\forall\vartheta.\ \kappa[\sigma] \models \kappa$	$(\sigma \in \mathcal{S}(\vartheta))$
[*VElim*]	$\forall\vartheta.\ \kappa\{v \simeq \phi_C[\Theta_C]\} \models \kappa \wedge v = v'_C[\vartheta'_C]$	$(v'_C \# \vartheta'_C \# (\vartheta, \kappa))$
[*MRef*]	$\forall\vartheta.\ \kappa \models \kappa \wedge \theta \leq \theta$	
[*MTrans*]	$\forall\vartheta.\ \kappa\{\theta \leq \theta' \leq \theta''\} \models \kappa \wedge \theta \leq \theta''$	
[*MIntro*]	$\forall\vartheta.\ \kappa\{\Theta_C \leq_C \Theta'_C \wedge \phi_C \sqsubseteq \phi'_C\} \models \kappa \wedge \phi_C[\Theta_C] \leq \phi'_C[\Theta'_C]$	
[*MElim*]	$\forall\vartheta.\ \kappa\{\phi_C[\Theta_C] \leq \phi'_C[\Theta'_C]\} \models \kappa \wedge \phi_C \sqsubseteq \phi'_C \wedge \Theta_C \leq_C \Theta'_C$	

Figure 3.1: Constraint implication

is equivalent to $\forall\,\emptyset.\ c_3$ or is just a strict subtype (since the smallest α above c_1 and c_2 in any admissible extension of $\mathcal{T}$ is necessarily below c_3, which is an upper bound of c_1 and c_2). So suppose that a new constructor c_4 is now added to C with the ordering $c_1 \sqsubseteq_C c_4$, $c_2 \sqsubseteq_C c_4$ and $c_4 \sqsubseteq_C c_3$. The addition of c_4 is admissible since it does not change the ordering between existing constructors. In this new context, the greatest lower bound of c_1 and c_2 is c_4, not c_3, hence τ is not equivalent to $\forall\,\emptyset.\ c_3$. Formally, for τ and $\forall\,\emptyset.\ c_3$ to be equivalent, both constraint implications

$$\forall\,\emptyset.\ \mathit{true} \Longrightarrow \exists\,\alpha.\ (c_1 \leq \alpha \wedge c_2 \leq \alpha \wedge \alpha \leq c_3)$$

and

$$\forall\,\alpha.\ (c_1 \leq \alpha \wedge c_2 \leq \alpha) \Longrightarrow (c_3 \leq \alpha)$$

would have to hold in every admissible extension $\mathcal{T}^*$ of $\mathcal{T}$. This is indeed the case for the former implication, but not for the latter, which does not hold for $\alpha = c_4$. Consequently, $\forall\alpha\colon c_1 \leq \alpha \wedge c_2 \leq \alpha.\ \alpha$ is a strict subtype of $\forall\,\emptyset.\ c_3$.

3.2 Constraint implication

We must therefore try to axiomatize this notion of constraint implication in a decidable way. To this end, we introduce the judgement $\forall\vartheta.\ \kappa \models \kappa'$ which reads "constraint κ implies κ' for all ϑ" axiomatized by the rules of figure 3.1 plus the transitivity rule

$$\frac{\forall\vartheta.\ \kappa_1 \models \kappa_2 \qquad \forall\vartheta.\ \kappa_2 \models \kappa_3}{\forall\vartheta.\ \kappa_1 \models \kappa_3}\ [\mathit{Trans}]$$

In these rules, we write $\theta_1 \simeq \theta_2$ to denote either $\theta_1 \leq \theta_2$ or $\theta_2 \leq \theta_1$. If ϑ is the empty set, we simply say that κ implies κ'. A constraint κ is said to be *well-formed* if *true* implies κ. Two constraints are said to be ϑ-equivalent if they imply each other for all ϑ. The following lemmas are used to modify the set ϑ of universally quantified variables of an implication: it is always legal to reduce ϑ, but only variables that do not occur on the right-hand side can be added to ϑ.

Lemma 1 (Quantification1)

$$\frac{\forall \vartheta, \vartheta'.\ \kappa \models \kappa'}{\forall \vartheta.\ \kappa \models \kappa'}$$

Lemma 2 (Quantification2)

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

The following lemma and corollaries are used to derive conjunctions $\kappa_1 \wedge \kappa_2$ on the right-hand side of implications. Basically, such derivations are legal only when every variable which is free both in κ_1 and κ_2 belongs to the set ϑ of universally quantified variables.

Lemma 3 (Conjunction1)

$$\frac{\forall \vartheta.\ \kappa_1 \models \kappa_1' \qquad \forall \vartheta.\ \kappa_2 \models \kappa_2' \qquad \kappa_1' \# \kappa_2'\ [\vartheta]}{\forall \vartheta.\ \kappa_1 \wedge \kappa_2 \models \kappa_1' \wedge \kappa_2'}$$

Corollary 4 (Conjunction2)

$$\frac{\forall \vartheta.\ \kappa \models \kappa_1 \qquad \forall \vartheta.\ \kappa \models \kappa_2 \qquad \kappa_1 \# \kappa_2\ [\vartheta]}{\forall \vartheta.\ \kappa \models \kappa_1 \wedge \kappa_2}$$

Corollary 5 (Conjunction3)

$$\frac{\forall \vartheta.\ \kappa\{\kappa'\} \models \kappa'' \qquad \kappa'\ \textit{is}\ \vartheta\textit{-closed}}{\forall \vartheta.\ \kappa \models \kappa' \wedge \kappa''}$$

3.3 Constraint normalization

In this section, we show that well-formed constraints can always be reduced, for any given variable set ϑ, to a simple ϑ-equivalent form, called a ϑ-prenormal form, which is essentially the conjunction of a "substitution" κ_s for some type variable in ϑ, a constraint κ_t on the type variables not substituted by κ_s, and a constraint κ_C on C-constructor variables for every class C. The κ_t and κ_C constraints do not contain complex monotypes of the form $\phi_C[\Theta_C]$. This prenormal form is used to decide implication separately on type variables and on C-constructor variables.

Definition 6 (Base form) *A constraint κ_b is said to be in base form if it is of the form $\kappa_t \wedge \bigwedge_{C \in \mathcal{T}} \kappa_C$, where κ_t and κ_C are of the following form*

$$\begin{array}{lcl} \kappa_t & ::= & v \leq v \mid \kappa_t \wedge \kappa_t \\ \kappa_C & ::= & \phi_C \sqsubseteq \phi_C \mid \kappa_C \wedge \kappa_C \end{array}$$

Definition 7 (Prenormal form) *Let ϑ be any variable set. A constraint is said to be in ϑ-prenormal form if it is of the form $\kappa_s \wedge \kappa_b$, where (1) κ_b is a constraint in base form, (2) κ_s is of the following form*

$$\kappa_s \quad ::= \quad \mathit{true} \mid v = \theta \mid \kappa_s \wedge \kappa_s$$

and (3) for every type variable v such that $v = \theta$ is a conjunct of κ_s, v belongs to ϑ and v does not occur in θ or in any other conjunct of κ_s.

3.3.1 Well-kinded constraints

In order to prove that every well-formed constraint has a ϑ-prenormal form, we first show that every well-formed constrained is *well-kinded*, that is, it is unifiable once C-constructors are identified with their class C. As always, we assume an implicit type structure $\mathcal{T}$, which, in particular, defines a language (constants and variables) to write constraints. We define an auxiliary algebra $\lceil\mathcal{T}\rceil$ that contains a function symbol C of arity n for every n-ary type constructor class C of $\mathcal{T}$. We inductively define a function $\lceil\_\rceil$ that takes monotypes to terms over $\lceil\mathcal{T}\rceil$ as follows:

$$\begin{array}{rcl} \lceil v \rceil & = & v \quad \text{if } v \text{ is a type variable} \\ \lceil \phi_C[\theta_1, \ldots, \theta_n] \rceil & = & C[\lceil\theta_1\rceil, \ldots, \lceil\theta_n\rceil] \end{array}$$

We say that a constraint κ is *well-kinded* if $\lceil\kappa\rceil$ is unifiable, where the set of equations $\lceil\kappa\rceil$ over $\lceil\mathcal{T}\rceil$ is inductively defined as follows

$$\begin{array}{rcl} \lceil \kappa \wedge \kappa' \rceil & = & \lceil\kappa\rceil \cup \lceil\kappa'\rceil \\ \lceil \phi_C \sqsubseteq \phi'_{C'} \rceil & = & \{C = C'\} \\ \lceil \theta_1 \leq \theta_2 \rceil & = & \{\lceil\theta_1\rceil = \lceil\theta_2\rceil\} \end{array}$$

For example, if κ is the constraint

$$\mathsf{list}[\alpha] \leq \mathsf{nil}[\langle \mathsf{int}, \mathsf{real} \rangle]$$

then κ is well-kinded, since

$$\{\ \mathsf{List}[\alpha] = \mathsf{List}[\mathsf{Pair}[\mathsf{Num}[\,], \mathsf{Num}[\,]]]\ \}$$

is unifiable in $\lceil\mathcal{T}\rceil$. Note, however, that κ is not well-formed, because $\mathsf{list} \sqsubseteq \mathsf{nil}$ does not hold.

Lemma 8 *Assume that κ_1 is a constraint such that $\lceil\kappa_1\rceil$ is unifiable, and that κ_1 implies κ_2 for all ϑ. Then $\lceil\kappa_2\rceil$ is unifiable.*

Corollary 9 *Every well-formed constraint κ is well-kinded.*

$$
\begin{array}{llll}
(N_1) & S,\ (\kappa \wedge \phi_C[\Theta] \leq \phi'_C[\Theta']) & \longrightarrow_\vartheta & S,\ (\kappa \wedge \phi_C \sqsubseteq \phi'_C \wedge \Theta \leq_C \Theta') \\
(N_2) & S,\ \kappa\{v \simeq \phi_C[\Theta]\} & \longrightarrow_\vartheta & (S \wedge v = v_C[\vartheta_C]),\ \kappa[v_C[\vartheta_C]/v] \\
 & & & \text{if } v \in \vartheta,\ \text{where } v_C \# \vartheta_C \# (v, \vartheta, S, \kappa) \\
(N_3) & S,\ \kappa\{v \simeq \phi_C[\Theta]\} & \longrightarrow_\vartheta & S[v_C[\vartheta_C]/v],\ \kappa[v_C[\vartheta_C]/v] \\
 & & & \text{if } v \notin \vartheta,\ \text{where } v_C \# \vartheta_C \# (\vartheta, S, \kappa)
\end{array}
$$

Figure 3.2: A rewrite system for prenormal form

3.3.2 Normalization by rewriting

Consider the rewriting system $\mathcal{N}$ that consists of the rewrite rules (N_1), (N_2), and (N_3) shown in figure 3.2, where ϑ is some variable set. Rewriting occurs on pairs of constraints (S, κ) where S consists of equations of the form $v = \theta$. We denote by $\Rightarrow_\vartheta$ the rewriting relation induced by these rules, and by $\Rightarrow^*_\vartheta$ the reflexive and transitive closure of $\Rightarrow_\vartheta$. In this section, we prove the following facts, assuming that κ is well-kinded:

- if $(S, \kappa) \Rightarrow_\vartheta (S', \kappa')$, then $S \wedge \kappa$ and $S' \wedge \kappa'$ are ϑ-equivalent constraints;
- if $(\mathit{true}, \kappa) \Rightarrow^*_\vartheta (S', \kappa')$, and (S', κ') cannot be rewritten via $\Rightarrow_\vartheta$, then $S' \wedge \kappa'$ is a constraint in ϑ-prenormal form;
- if κ is well-kinded, then (true, κ) does not allow an infinite sequence of rewrite steps.

At first, we state an auxiliary lemma, which intuitively says that the constraints $\kappa \wedge (v = \theta)$ and $\kappa[\theta/v] \wedge (v = \theta)$ are equivalent, where $\kappa[\theta/v]$ denotes the constraint obtained from κ by substituting θ for all occurrences of v.

Lemma 10 *If $S \wedge \kappa$ is well-kinded, then both $\forall\vartheta.\ S \wedge \kappa \wedge v = \theta \models S \wedge \kappa[\theta/v] \wedge v = \theta$ and $\forall\vartheta.\ S \wedge \kappa[\theta/v] \wedge v = \theta \models S \wedge \kappa \wedge v = \theta$ hold for any variable set ϑ.*

Lemma 11 *If $S \wedge \kappa$ is well-kinded, and $(S, \kappa) \Rightarrow_\vartheta (S', \kappa')$, then $S \wedge \kappa$ and $S' \wedge \kappa'$ are ϑ-equivalent, and κ' is well-kinded.*

The next lemma says that pairs (S, κ) that are irreducible w.r.t. $\Rightarrow_\vartheta$ represent constraints in ϑ-prenormal form.

Lemma 12 *If κ is a well-kinded constraint, and $(\mathit{true}, \kappa) \Rightarrow^*_\vartheta (S', \kappa')$ is such that (S', κ') cannot be rewritten via $\Rightarrow_\vartheta$, then $S' \wedge \kappa'$ is a constraint in ϑ-prenormal form.*

It remains to prove that no well-kinded constraint admits an infinite rewrite via $\Rightarrow_\vartheta$.

Lemma 13 *If κ is well-kinded, then there is no infinite sequence $(S_i, \kappa_i)_{i \geq 0}$ such that $(S_0, \kappa_0) = (\mathit{true}, \kappa)$ and for all $i \geq 0$, $(S_i, \kappa_i) \Rightarrow_\vartheta (S_{i+1}, \kappa_{i+1})$.*

Theorem 14 (Prenormal form) *Let ϑ be any variable set. Then every well-formed constraint is ϑ-equivalent to a constraint in ϑ-prenormal form.*

3.4 Interpretation of constraint implication

In this section, we show that for any two well-formed constraints κ_1 and κ_2, κ_1 implies κ_2 for all ϑ if and only if

$$\forall \vartheta. (\exists \vartheta_1. \kappa_1) \Longrightarrow (\exists \vartheta_2. \kappa_2)$$

holds for every admissible extension of $\mathcal{T}$ (the formal interpretation of the above first-order formula is stated in theorem 16 below), where ϑ_1 (resp. ϑ_2) is the set of free variables of κ_1 (resp. κ_2) which are not in ϑ. For the completeness proof, we make use of the prenormal form for constraints of theorem 14. We state a preliminary lemma for future reference.

Lemma 15 *For any base type constructors t_C and t'_C, and any ground monotypes θ and θ' of $\mathcal{T}$, if $\forall\emptyset.\ true \models t_C \sqsubseteq t'_C$ or $\forall\emptyset.\ true \models \theta \leq \theta'$, then $t_C \sqsubseteq_C t'_C$ or $\theta \leq \theta'$ holds in $\mathcal{T}$.*

The soundness of the axiom system of figure 3.1 can be shown by an obvious inductive proof along derivations, and is formalized by the following lemma.

Lemma 16 (Soundness of implication) *Let κ_1 and κ_2 be constraints and ϑ be a variable set such that $\forall\vartheta.\ \kappa_1 \models \kappa_2$ is derivable. Then for every admissible extension $\mathcal{T}^*$ of $\mathcal{T}$ and every $\mathcal{T}^*$-ground substitution σ_1 such that $\kappa_1[\sigma]$ is satisfied in $\mathcal{T}^*$, there exists a $\mathcal{T}^*$-ground substitution σ_2 that agrees with σ_1 for the variables in ϑ such that $\kappa_2[\sigma_2]$ is a ground constraint satisfied in $\mathcal{T}^*$.*

The remainder of this section is devoted to proving the completeness of our axiomatization of the implication, that is, the "if" part of the above statement (formalized in theorem 22). By theorem 14, we may assume that κ_1 is a constraint in ϑ-prenormal form. Moreover, we may assume that the only variables that appear both in κ_1 and κ_2 are in ϑ. With this assumption, lemma 1 and lemma 2 imply that $\forall\vartheta.\ \kappa_1 \models \kappa_2$ holds if and only if $\forall\vartheta, \vartheta_1.\ \kappa_1 \models \kappa_2$ holds where ϑ_1 is the variable set that contains all variables that appear in κ_1, but not in ϑ or κ_2. For notational simplicity, we therefore assume that κ_1 is ϑ-closed.

We first consider constraints κ_1 in base form. We show that we can define an admissible extension of $\mathcal{T}$ from κ_1 in a "syntactic" fashion. To this end, we define the set C^ϑ as the set that consists of all ground type constructors t_C of class C in $\mathcal{T}$ and all constructor variables v_C of class C in ϑ. We denote by T^ϑ the set of type variables in ϑ. We define binary relations $\simeq_C$ on C^ϑ (for every type constructor class C in $\mathcal{T}$) and $\simeq_T$ on T^ϑ as follows

$$\begin{aligned} \phi_C \simeq_C \phi'_C \quad &\text{iff} \quad \forall\vartheta.\ \kappa_1 \models \phi_C = \phi'_C \\ v \simeq_T v' \quad &\text{iff} \quad \forall\vartheta.\ \kappa_1 \models v = v' \end{aligned}$$

Rules *CRef*, *CTrans*, *MRef*, and *MTrans* ensure that all relations $\simeq_C$ and $\simeq_T$ are equivalence relations on their respective domains. We denote the equivalence class of ϕ_C with respect to $\simeq_C$ by $[\phi_C]$ (and similarly for type variables), and define relations $\sqsubseteq^1_C$ and $\leq^1$ on these equivalence classes by

$$\begin{aligned} [\phi_C] \sqsubseteq^1_C [\phi'_C] \quad &\text{iff} \quad \forall\vartheta.\ \kappa_1 \models \phi_C \sqsubseteq \phi'_C \\ [v] \leq^1 [v'] \quad &\text{iff} \quad \forall\vartheta.\ \kappa_1 \models v \leq v' \end{aligned}$$

The following lemma shows that these relations are well-defined partial orderings and that, moreover, the collection $\mathcal{T}^\vartheta$ that contains the type constructor classes $C^\vartheta / \cong_C$, as well as an additional type constructor class $T^\vartheta / \cong_T$ whose orderings are given by $\sqsubseteq^1_C$ and $\leq^1$ is (an isomorphic copy of) an admissible extension of $\mathcal{T}$ if κ_1 is well-formed.

Lemma 17 *Let ϑ be a variable set, and κ_1 be a well-formed ϑ-closed constraint in base form. Then $\mathcal{T}^\vartheta$ is (up to isomorphism) an admissible extension of $\mathcal{T}$, where $\mathcal{T}$ consists of the type constructor classes $(N_C, C^\vartheta / \cong_C, D_C / \cong_C, \sqsubseteq^1_C, \partial_C)$ for every class C of $\mathcal{T}$, and the additional type constructor class $T = (N_T, T^\vartheta / \cong_T, \emptyset, \leq^1, ())$, where N_T is different from any N_C, if ϑ contains some type variable.*

The following lemma shows that any ground constraint κ (over the alphabet of $\mathcal{T}^\vartheta$) holds in $\mathcal{T}^\vartheta$ iff κ is implied by κ_1 for all ϑ.

Lemma 18 *Let ϑ be a variable set, κ_1 be a well-formed ϑ-closed constraint in base form, and κ be a ϑ-closed constraint. Then κ holds in $\mathcal{T}^\vartheta$ if and only if κ_1 implies κ for all ϑ.*

We can now assert theorem 22 for the case where κ_1 is a constraint in base form.

Lemma 19 *Let ϑ be a variable set, κ_1 be a well-formed ϑ-closed constraint in base form, and κ_2 be any constraint. If for every admissible extension $\mathcal{T}^*$ of $\mathcal{T}$, and every $\mathcal{T}^*$-ground substitution σ_1 such that $\kappa_1[\sigma_1]$ is satisfied in $\mathcal{T}^*$, there exists a $\mathcal{T}^*$-ground substitution σ_2 that agrees with σ_1 on the variables of ϑ such that $\kappa_2[\sigma_2]$ is a ground constraint satisfied in $\mathcal{T}^*$, then κ_1 implies κ_2 for all ϑ.*

It now remains to generalize lemma 19 to constraints in ϑ-prenormal form.

Lemma 20 *Let ϑ be a variable set, θ be a ϑ-closed monotype, κ_1 be a ϑ-closed constraint, κ_2 be an arbitrary constraint, and v be a variable in ϑ which does not occur in κ_1, κ_2, or θ. Then $\forall\vartheta.\, v = \theta \wedge \kappa_1 \models \kappa_2$ if and only if $\forall\vartheta.\, \kappa_1 \models \kappa_2$.*

Lemma 21 *Lemma 19 remains valid if κ_1 is a constraint in ϑ-prenormal form.*

Theorem 22 (Interpretation) *Let ϑ be a variable set, and κ_1 and κ_2 be two well-formed constraints. Then $\forall\vartheta.\, \kappa_1 \models \kappa_2$ holds if and only if for every admissible extension $\mathcal{T}^*$ of $\mathcal{T}$ and every $\mathcal{T}^*$-ground substitution σ_1 such that $\kappa_1[\sigma_1]$ is satisfied in $\mathcal{T}^*$, there exists a $\mathcal{T}^*$-ground substitution σ_2 that agrees with σ_1 on the variables of ϑ such that $\kappa_2[\sigma_2]$ is a ground constraint satisfied in $\mathcal{T}^*$.*

Corollary 23 (Satisfiability) *A constraint κ is well-formed if and only if it is has a ground solution in $\mathcal{T}$.*

3.5 Decidability of constraint implication

In this section, we sketch an extremely inefficient algorithm to decide whether $\forall\vartheta.\ \kappa_1 \models \kappa_2$ holds. We make use of the prenormal form for constraints established in theorem 14 and of several lemmas proved for theorem 22.

Let us call a derivation for $\forall\vartheta.\ \kappa_1 \models \kappa_2$ *normal* if it ends with an application of rule *VIntro* followed by an application of rule *Approx*, but does not apply *VIntro* before.

Lemma 24 *If $\forall\vartheta.\ \kappa_1 \models \kappa_2$, then there is a normal derivation of $\forall\vartheta.\ \kappa_1 \models \kappa_2$.*

We say that a constraint κ is *simple* if it does not contain any subconstraint of the form $v \simeq \phi_C[\Theta_C]$, and if for every subconstraint $\phi_C[\Theta] \leq \phi'_C[\Theta'_C]$, $\Theta_C \leq_C \Theta'_C$ is a simple constraint. In particular, a constraint in base form is simple. For simple well-kinded constraints, we define a syntactical operation *atomize* inductively as follows:

$$\begin{aligned}
\mathit{atomize}(\phi_C \sqsubseteq \phi'_C) &= \phi_C \sqsubseteq \phi'_C \\
\mathit{atomize}(v \leq v') &= v \leq v' \\
\mathit{atomize}(\phi_C[\Theta] \leq \phi'_C[\Theta']) &= \phi_C \sqsubseteq \phi'_C \wedge \mathit{atomize}(\Theta \leq_C \Theta') \\
\mathit{atomize}(\kappa_1 \wedge \kappa_2) &= \mathit{atomize}(\kappa_1) \wedge \mathit{atomize}(\kappa_2)
\end{aligned}$$

Clearly, if κ is simple and well-kinded, then so is $\mathit{atomize}(\kappa)$.

Lemma 25 *Assume that κ_1 is a well-kinded and ϑ-closed constraint in base form. If κ_2 is a ϑ-closed constraint in base form, then $\forall\vartheta.\ \kappa_1 \models \kappa_2$ holds iff $\forall\vartheta.\ \kappa_1 \models \kappa_2$ can be derived using only the rules Approx, CRef, CTrans, CTriv, CMin, MRef, and MTrans.*

Lemma 26 *Assume that κ_1 and κ_2 are ϑ-closed and in base form and that κ_1 is well-kinded. Then it is decidable whether $\forall\vartheta.\ \kappa_1 \models \kappa_2$ holds.*

Reusing lemmas from 3.4, we can lift lemma 26 to arbitrary constraints κ_2 in base form.

Lemma 27 *Assume that κ_1 and κ_2 are in base form, and that κ_1 is ϑ-closed and well-kinded. Then it is decidable whether $\forall\vartheta.\ \kappa_1 \models \kappa_2$ holds.*

Finally, we establish the decidability of constraint implication.

Theorem 28 (Decidability) *If κ_1 is a well-formed constraint, and κ_2 is an arbitrary constraint, then it is decidable whether $\forall\vartheta.\ \kappa_1 \models \kappa_2$ holds. In particular, well-formedness of constraints is decidable.*

3.6 Constraint contexts

In order to deal with nested polymorphic functions, we define the notion of *constraint context* (or context, for short) as a pair Δ, written $(\vartheta : \kappa)$, where ϑ is a variable set and κ is a constraint. We say that Δ is *well-formed* if κ is both ϑ-closed, and well-formed.

Intuitively, a constraint context is one of the components of typing contexts (cf. chapter 5) and denotes the assumption "there exists variables ϑ such that κ holds". Let Δ and Δ' denote the two contexts $(\vartheta : \kappa)$ and $(\vartheta' : \kappa')$. We define the conjunction $\Delta \wedge \Delta'$ of Δ and Δ' as $(\vartheta, \vartheta' : \kappa \wedge \kappa')$. If Δ is well-formed, we say that Δ' is well-formed w.r.t. Δ if ϑ and ϑ' are disjoint, κ' is (ϑ, ϑ')-closed, and κ implies κ' for all ϑ, in which case we define the *update* $\Delta[\Delta']$ of Δ by Δ' as the conjunction of Δ and Δ', and corollary 5 shows that $\Delta[\Delta']$ is well-formed. We define the *trivial context True* as $(\emptyset : \mathit{true})$. An alternative definition would have had Δ' well-formed w.r.t. Δ iff $\Delta \wedge \Delta'$ is well-formed. For instance, if Δ is of the form $(\alpha : \mathit{true})$ and Δ' is of the form $(\beta : \beta \leq \alpha \wedge \alpha \leq \mathsf{int})$, then Δ' is well-formed w.r.t. Δ for the latter definition, but is not well-formed for the former definition, since it is not true that there exists a β such that $\beta \leq \alpha$ and $\alpha \leq \mathsf{int}$ for every α satisfying *true*. Unfortunately, this weaker definition is not very useful for the intended use of constraint contexts: if Δ is an existentially quantified constraint, and Δ' is a "nested" constraint, e.g., the constraint defining the domain of a nested function, then Δ' should not enforce properties on the variables in ϑ that are not already present in κ, or else, there is a risk of inconsistency between nested constraints.

Chapter 4

Type system

This chapter formally introduces the notion of universally quantified polymorphic constrained type $\forall\vartheta\colon \kappa.\ \theta$. We define a syntactic preorder on the set of types and show that this set is a sup-semi-lattice modulo equivalence. We show that functional types of the form $\forall\vartheta\colon \kappa.\ \theta \to \theta''$ denote monotonic type transformers over their domain $\exists\vartheta\colon \kappa.\ \theta$. We define a syntactic preorder on the set of domains and show that this set is an inf-semi-lattice modulo equivalence. We show that types and domains are related by a membership relation such that if a type belongs to a domain, then so does all its subtypes, which shows that a type τ_1 is a subtype of another type τ_2 if and only if every object with type τ_1 can be safely used everywhere an object of type τ_2 can be safely used. Differently stated, domains denote downward-closed sets of types. This property corresponds to the substitutivity property associated to the subclasses of a class in class-based object-oriented languages.

In this chapter, we assume given an implicit type structure $\mathcal{T}$ and and implicit well-formed context Δ of the form $(\widehat{\vartheta}\colon \widehat{\kappa})$.

4.1 Types

In this section, we assume given two variable sets ϑ_1 and ϑ_2 and two monotypes θ_1 and θ_2. For $i \in [1,2]$, we define the type τ_i as $\forall\vartheta_i\colon \kappa_i.\ \theta_i$. We say that τ_i is *closed* if κ_i and θ_i are ϑ_i-closed. We define the *closure* $\tau_i[\Delta]$ of τ_i w.r.t. Δ as $\forall\widehat{\vartheta},\vartheta_i\colon \widehat{\kappa} \wedge \kappa_i.\ \theta_i$. We say that τ_i is *well-formed* w.r.t. Δ, written $\Delta \vdash \tau_i$, if $(\vartheta_i\colon \kappa_i)$ is well-formed w.r.t. Δ and θ_i is $(\widehat{\vartheta},\vartheta_i)$-closed. If τ_1 and τ_2 are both well-formed, we say that τ_1 is a *subtype* of τ_2 w.r.t. Δ if $\Delta \vdash \tau_1 \le \tau_2$ can be derived from rule *Type* of figure 4.1. We say that τ_1 and τ_2 are *equivalent* w.r.t. Δ, written $\Delta \vdash \tau_1 \doteq \tau_2$, if τ_1 is a subtype of τ_2 and τ_2 is a subtype of τ_1. When Δ is trivial or implicit, we simply write $\tau_1 \le \tau_2$ and $\tau_1 \doteq \tau_2$. For instance, type

$$\forall\alpha,\beta\colon (\mathsf{nil}[\alpha] \le \beta \wedge \mathsf{cons}[\alpha] \le \beta).\ \beta$$

is a subtype of $\forall\gamma.\ \mathsf{list}[\gamma]$ since

$$\forall\gamma.\ \mathit{true} \models (\mathsf{nil}[\alpha] \le \beta \wedge \mathsf{cons}[\alpha] \le \beta \wedge \beta \le \mathsf{list}[\gamma])$$

$$\frac{\sigma \in \mathcal{R}(\vartheta; \vartheta_1; \vartheta_2) \qquad \forall \vartheta, \vartheta_2.\ \kappa \wedge \kappa_2 \models \kappa_1[\sigma] \wedge \theta_1[\sigma] \leq \theta_2}{\vartheta \colon \kappa \vdash (\forall \vartheta_1 \colon \kappa_1.\ \theta_1) \leq (\forall \vartheta_2 \colon \kappa_2.\ \theta_2)}\ [\mathit{Type}]$$

$$\frac{\sigma \in \mathcal{R}(\vartheta; \vartheta_2; \vartheta_1) \qquad \forall \vartheta, \vartheta_1.\ \kappa \wedge \kappa_1 \models \kappa_2[\sigma] \wedge \theta_1 \leq \theta_2[\sigma]}{\vartheta \colon \kappa \vdash (\exists \vartheta_1 \colon \kappa_1.\ \theta_1) \leq (\exists \vartheta_2 \colon \kappa_2.\ \theta_2)}\ [\mathit{Domain}]$$

$$\frac{\sigma \in \mathcal{R}(\vartheta; \vartheta_1; \vartheta_2) \qquad \forall \vartheta.\ \kappa \models \kappa_1[\sigma] \wedge \kappa_2 \wedge \theta_1[\sigma] \leq \theta_2}{\vartheta \colon \kappa \vdash (\forall \vartheta_1 \colon \kappa_1.\ \theta_1) \in (\exists \vartheta_2 \colon \kappa_2.\ \theta_2)}\ [\mathit{Member}]$$

Figure 4.1: Type and domain orderings

holds (choose $\alpha = \gamma$ and $\beta = \mathsf{list}[\gamma]$), but they are not equivalent since

$$\forall \alpha, \beta.\ (\mathsf{nil}[\alpha] \leq \beta \wedge \mathsf{cons}[\alpha] \leq \beta \wedge) \models (\mathsf{list}[\gamma] \leq \beta)$$

does not hold. This is easy to understand in an "open world", since it may be the case that in some other module of the program, constructors nil and cons have an upper bound which is a strict subconstructor of list (cf. figure 1.1). The following lemma gives an equivalent definition of subtyping which is less intuitive but can be handy in some proofs, since this definition does not impose the renaming of the universally quantified variables of one of the types.

Lemma 29 (Type ordering) *Let τ_i, $i \in [1, 2]$, be two well-formed types $\forall \vartheta_i \colon \kappa_i.\ \theta_i$ w.r.t. Δ, and v be a fresh type variable not in $(\widehat{\vartheta}, \vartheta_1, \vartheta_2)$. Then τ_1 is a subtype of τ_2 w.r.t. Δ if and only if*

$$\forall \widehat{\vartheta}, v.\ \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq v \models \kappa_1 \wedge \theta_1 \leq v$$

The next lemma shows that the universally quantified variables of a well-formed type can be freely renamed as needed without altering its semantics.

Lemma 30 (Type renaming) *Let $\tau = \forall \vartheta \colon \kappa.\ \theta$ be a well-formed type w.r.t. Δ, and ϑ' be a variable set disjoint from $\widehat{\vartheta}$. Then τ is equivalent to $\forall \vartheta[\sigma] \colon \kappa[\sigma].\ \theta[\sigma]$ for any renaming $\sigma \in \mathcal{R}(\widehat{\vartheta}; \vartheta; \vartheta')$.*

The following lemmas shows that the ordering on types is a consistent and decidable extension of the standard ordering on ground monotypes.

Lemma 31 *The type ordering relation is decidable.*

Lemma 32 (Type extension) *Let θ_1 and θ_2 be two ground monotypes such that $\theta_1 \leq \theta_2$. Then $\forall \emptyset.\ \theta_1$ is a subtype of $\forall \emptyset.\ \theta_2$ w.r.t. Δ.*

We say that a well-formed type τ' of the form $\forall \vartheta' \colon \kappa'.\ \theta'$ is an *instance* of a well-formed type τ of the form $\forall \vartheta \colon \kappa.\ \theta$ if there exists a $\widehat{\vartheta}$-substitution σ such that $\kappa' = \kappa[\sigma]$ and

$\theta' = \theta[\sigma]$. For instance, type $\forall\emptyset\colon \mathsf{int} \leq \mathsf{real}.\ \mathsf{real} \to \mathsf{int}$, which is equivalent to $\mathsf{real} \to \mathsf{int}$, is an instance of type $\forall\alpha, \beta\colon \beta \leq \alpha.\ \alpha \to \beta$ but $\forall\emptyset\colon \mathsf{real} \leq \mathsf{int}.\ \mathsf{int} \to \mathsf{real}$ is not, since the constraint $\mathsf{real} \leq \mathsf{int}$ is ill-formed. As in ML, we can show that every polymorphic type is a subtype of all its instances.

Theorem 33 (Type instances) *A well-formed type is a subtype of all its instances.*

In particular, a type is a subtype of all its ground instances. Consequently, the universal quantifier can be intuitively interpreted as the greatest lower bound, so that, for instance, $\forall\alpha\colon \mathsf{int} \leq \alpha.\ \alpha$ can be read as "the (set of) smallest α above int", which is precisely int in this simple case. However, note that this interpretation is rather informal in that, assuming a ()-variant class C with type constructors c_1, c_2, and c_3 partially ordered by $c_1 \sqsubseteq_C c_3$ and $c_2 \sqsubseteq_C c_3$, it is not the case that $\forall\alpha\colon c_1 \leq \alpha \wedge c_2 \leq \alpha.\ \alpha$ and $\forall\,\emptyset.\ c_3$ are equivalent (cf. beginning of chapter 3). This greatest lower bound interpretation is thus only valid only if the greatest lower bound is a solution of the constraint.

Formally, defining the *denotation* of a well-formed type w.r.t. the trivial context and type structure $\mathcal{T}$ as the following upper-ideal (i.e., upward-closed set of ground monotypes w.r.t. the standard ordering, sometimes called *filter*)

$$[\![\forall\vartheta\colon \kappa.\ \theta]\!]_{\mathcal{T}} \quad = \quad \{\theta' \in \mathcal{G}_{\mathcal{T}} \mid \exists_{\mathcal{T}}\,\vartheta.\ \kappa \wedge \theta \leq \theta'\}$$

we have the following theorem.

Theorem 34 (Type denotation) *Let τ and τ' be two well-formed types w.r.t. the trivial context and type structure $\mathcal{T}$. Then τ is a subtype of τ' if and only if $[\![\tau']\!]_{\mathcal{T}^*}$ is a subset of $[\![\tau]\!]_{\mathcal{T}^*}$ for every admissible extension $\mathcal{T}^*$ of $\mathcal{T}$.*

The following theorem shows that the set of types is a sup-semi-lattice, that is, two types with an upper bound have a least upper bound. This result, which holds without any hypothesis on the partial orderings between type constructors, is particularly important since the existence of a least upper bound is essential to ensure minimal typing of expressions like conditional expressions.

Theorem 35 (Sup-semi-lattice of types) *Well-formed types modulo equivalence form a sup-semi-lattice with least element $\forall\alpha.\ \alpha$. Two well-formed types τ_1 and τ_2 are said to be compatible if they have a common supertype. If τ_1 of the form $\forall\vartheta_1\colon \kappa_1.\ \theta_1$ and τ_2 of the form $\forall\vartheta_2\colon \kappa_2.\ \theta_2$ are compatible, $(\vartheta_1 \not\# \vartheta_2)$, and v is a fresh type variable, then the least upper bound $(\tau_1 \vee \tau_2)$ of τ_1 and τ_2 is equivalent to*

$$\forall v, \vartheta_1, \vartheta_2\colon (\kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq v \wedge \theta_2 \leq v).\ v$$

4.2 Functional types and domains

A well-formed type is said to be *functional* if its monotype is of the form $\theta \to \theta'$. We intend functional types to be the types of functions and methods, and to denote monotonic type transformers, abstracting the input-output relation of the function. However, as in

ML, typing rules must be able to deal with cases where the function has the empty type $\forall\alpha.\ \alpha$, as in

$$\mathsf{letrec}\ loop() = loop()\ \mathsf{in}\ (loop())\ 1\ \mathsf{end}$$

for instance, where $(loop\ ())$ has the empty type, so that $((loop())\ 1)$ also has the empty type. In order to turn empty types into functional types, we define the *functional cast* operator fun_Δ as follows

$$fun_\Delta(\forall\vartheta : \kappa.\ \theta \to \theta') \quad = \quad \forall v, v', \vartheta : \kappa \wedge \theta \leq (v \to v').\ (v \to v')$$

where v and v' are distinct type variables not in $(\widehat{\vartheta}, \vartheta)$, and we say that τ is *prefunctional* if $fun_\Delta(\tau)$ is well-formed w.r.t. Δ. For instance, the minimum type $\forall\alpha.\ \alpha$ is prefunctional and is mapped to $\forall\alpha,\beta.\ \alpha \to \beta$ by the functional cast operator. The following lemma shows that fun_Δ is covariant (that is, monotonic w.r.t. the subtyping relation, which is a preorder). In particular, fun_Δ preserves equivalence. Also, note that functional types are mapped to themselves (up to equivalence) by the functional cast operator and that every prefunctional type is a subtype of its functional cast, so that fun_Δ is an upper-closure over the set of prefunctional types modulo equivalence.

Theorem 36 (Fun covariance) *The functional cast operator is an upper-closure over the set of prefunctional types, and if τ_1 is a subtype of τ_2 w.r.t. Δ, and τ_2 is prefunctional, then $fun_\Delta(\tau_1)$ is a subtype of $fun_\Delta(\tau_2)$ w.r.t. Δ.*

As every functional type denotes a type transformer, we may like to define the *domain* of a functional type as the set of types over which it is defined. To this end, we define the domain operator as follows

$$dom(\forall\vartheta : \kappa.\ \theta \to \theta') \quad = \quad \exists\vartheta : \kappa.\ \theta$$

and for any prefunctional type τ, we define $dom_\Delta(\tau)$ as $dom(fun_\Delta(\tau))$. Intuitively, the domain of a functional type is its set of valid input types, and if types denote the *upper*-closure of their set of ground instances, domains denote the *lower*-closure of their set of ground instances. The idea is that function domains must be downward closed for the "substitutivity property" associated to subtyping in object-oriented programming to hold. For instance, a functional type τ_1 like $\forall\alpha : \alpha \leq \mathsf{real}.\ \langle\alpha, \alpha\rangle \to \alpha$, which may be the type of the addition operator over reals, has domain $\exists\alpha : \alpha \leq \mathsf{real}.\ \langle\alpha, \alpha\rangle$, which, intuitively, contains all the subtypes of $\langle\mathsf{real}, \mathsf{real}\rangle$, e.g. $\langle\mathsf{int}, \mathsf{real}\rangle$. Note that the domain operator is not injective. For instance, a functional type τ_2 like $\forall\,\emptyset.\ \langle\mathsf{real}, \mathsf{real}\rangle \to \mathsf{real}$, which is a strict supertype of τ_1, has the same domain as τ, namely $\exists\,\emptyset.\ \langle\mathsf{real}, \mathsf{real}\rangle$. In other words, τ_1 and τ_2 denote two distinct type transformers over the domain of all pairs of reals.

The theory of domains is very similar to that of types. For $i \in [1, 2]$, we define the domain δ_i as $\exists\vartheta_i : \kappa_i.\ \theta_i$. We say that δ_i is *closed* of κ_i and θ_i are ϑ_i-closed. We define the *closure* $\delta_i[\Delta]$ of δ_i w.r.t. Δ as $\exists\widehat{\vartheta}, \vartheta_i : \widehat{\kappa} \wedge \kappa_i.\ \theta_i$. We say that δ_i is *well-formed* w.r.t. Δ, written $\Delta \vdash \delta_i$, if $(\vartheta_i : \kappa_i)$ is well-formed w.r.t. Δ and θ_i is $(\widehat{\vartheta}, \vartheta_i)$-closed. If δ_1 and δ_2 are both well-formed, we say that δ_1 is a *subdomain* of δ_2 w.r.t. Δ if $\Delta \vdash \delta_1 \leq \delta_2$ can be derived from rule *Domain* of figure 4.1. We say that δ_1 and δ_2 are *equivalent* w.r.t. Δ, written $\Delta \vdash \delta_1 \doteq \delta_2$, if δ_1 is a subdomain of δ_2 and δ_2 is a subdomain of δ_1. When Δ is trivial or implicit, we simply write $\delta \leq \delta'$ and $\delta \doteq \delta'$. As for types, the universally quantified variables of a domain can be freely renamed, the ordering on domains is a consistent and decidable extension of the standard ordering on ground monotypes, and can be given an equivalent definition, namely, δ_1 is a subdomain of δ_2 w.r.t. Δ if and only if

$$\forall \widehat{\vartheta}, v.\ \widehat{\kappa} \wedge \kappa_1 \wedge v \leq \theta_1 \models \kappa_2 \wedge v \leq \theta_2$$

holds for some fresh variable v.

Now, defining domain instances like type instances, it can be shown that every well-formed domain is a *superdomain* of all its instances. In particular, a well-formed domain is a *superdomain* of all its ground instances. Consequently, the existential quantifier can be intuitively interpreted as the *least upper bound*, so that, for instance, $\exists \alpha\colon \alpha \leq \mathsf{int}.\ \alpha$ can be read as "the (set of) largest α below int", which is precisely int in this simple case. As for types, though, note that this is just an intuitive interpretation. Finally, it can be shown that the set of domains modulo equivalence is an *inf-semi-lattice* with greatest element $\exists \alpha.\ \alpha$, and that the greatest lower bound $(\delta_1 \wedge \delta_2)$ of two compatible domains δ_1 and δ_2 is equivalent to

$$\exists v, \vartheta_1, \vartheta_2\colon (\kappa_1 \wedge \kappa_2 \wedge v \leq \theta_1 \wedge v \leq \theta_2).\ v$$

where v is a fresh variable.

We said earlier that domains denote sets of valid input *types* of functional types seen as type transformers, but so far, we've only interpreted domains as downward-closed sets of *monotypes*. In order to interpret domains as sets of *types*, we define a membership relation between types and domains and interpret every domain as the set of types which belong to it. Formally, we say that a well-formed type τ *belongs* to a well-formed domain δ if $\Delta \vdash \tau \in \delta$ can be derived from rule *Member* of figure 4.1. In other words, τ belong to δ if and only if there exists an instance of τ which is below some instance of δ. For example, the minimum type $\forall \alpha.\ \alpha$ belong to every domain, e.g., $\exists \emptyset.\ \mathsf{unit}$, since

$$\forall \widehat{\vartheta}.\ \widehat{\kappa} \models \alpha \leq \mathsf{unit}$$

trivially holds (e.g., choose $\alpha = \mathsf{unit}$). Note that τ belongs to δ if and only if any renaming of τ belongs to any renaming of δ. The following lemma show that type membership is a consistent and decidable extension of the standard ordering on ground monotypes.

Lemma 37 *Type membership is decidable.*

Lemma 38 (Membership extension) *Let θ_1 and θ_2 be two ground monotypes such that $\theta_1 \leq \theta_2$. Then $\forall \emptyset.\ \theta_1$ belongs to $\exists \emptyset.\ \theta_2$ w.r.t. Δ.*

The following theorem shows that domains are downward closed w.r.t. type membership, that is, if a type belongs to a domain, then so does all its subtypes. In particular, if a type τ belongs to a domain δ, than any type in the equivalence class of τ belongs to δ and τ belongs to any domain in the equivalence class of δ. This "transitivity" property shows that, as expected, domains denote downward-closed sets of types, which is of course essential from the perspective of object-oriented programming.

Theorem 39 (Membership transitivity) *The type membership relation is transitive in the following sense*

$$\frac{\Delta \vdash \tau_1 \leq \tau_2 \qquad \Delta \vdash \tau_2 \in \delta_3}{\Delta \vdash \tau_1 \in \delta_3} \qquad\qquad \frac{\Delta \vdash \tau_1 \in \delta_2 \qquad \Delta \vdash \delta_2 \leq \delta_3}{\Delta \vdash \tau_1 \in \delta_3}$$

Note that this theorem also suggests that domains may be considered as existential types by considering type membership as the partial ordering between types and domains. Also, note that it is *not* the case that the least upper bound of two types in the same domain δ belong to δ. For instance, let δ be the domain

$$\exists\alpha\colon \mathsf{int} \leq \alpha \wedge \alpha \leq \mathsf{real}.\ \alpha \rightarrow \alpha$$

τ_1 be the type $\forall\emptyset.\ \mathsf{int} \rightarrow \mathsf{int}$, and τ_2 be the type $\forall\emptyset.\ \mathsf{real} \rightarrow \mathsf{real}$. Then τ_1 and τ_2 obviously belong to δ, but $\tau_1 \vee \tau_2$, which is a well-formed typed defined as

$$\forall\beta\colon \mathsf{int} \rightarrow \mathsf{int} \leq \beta \wedge \mathsf{real} \rightarrow \mathsf{real} \leq \beta.\ \beta$$

does not belong to δ, since the constraint

$$\mathsf{int} \leq \alpha \wedge \alpha \leq \mathsf{real} \wedge \mathsf{int} \rightarrow \mathsf{int} \leq \beta \wedge \mathsf{real} \rightarrow \mathsf{real} \leq \beta \wedge \beta \leq \alpha \rightarrow \alpha$$

is not satisfiable. This unfortunate property of type membership comes from the fact that the denotations of well-formed types and domains are not necessarily principal ideals. Note that, symmetrically, a type which belongs to two domains does not necessarily belong to the greatest lower bound of these domains.

The following lemma shows that dom_Δ is contravariant (that is, anti-monotonic) and thus maps equivalent types to equivalent domains. This property corresponds to our intuition that it is necessary that $dom_\Delta(\tau_2)$ be a subdomain of $dom_\Delta(\tau_1)$ for a function with type τ_1 to be used safely where a function with type τ_2 is expected.

Theorem 40 (Dom contravariance) *The domain operator dom_Δ is contravariant, that is, if τ_1 is a subtype of τ_2 w.r.t. Δ, and τ_2 is prefunctional, then $dom_\Delta(\tau_2)$ is a subdomain of $dom_\Delta(\tau_1)$ w.r.t. Δ.*

Now that we've studied the domains of functional types, we are going to see how a functional type τ_0 of the form $\forall\vartheta_0\colon \kappa_0.\ \theta_0 \rightarrow \theta_0'$ can be identified with a monotonic type transformer. To this end, we define the application $app(\tau_0, \tau_1)$ of τ_0 to a well-formed type τ_1 of the form $\forall\vartheta_1\colon \kappa_1.\ \theta_1$ as

$$app(\tau_0, \tau_1) \quad = \quad \forall\vartheta_0, \vartheta_1[\sigma_1]\colon \kappa_0 \wedge \kappa_1[\sigma_1] \wedge \theta_1[\sigma_1] \leq \theta_0.\ \theta_0'$$

where σ_1 is any renaming in $\mathcal{R}(\widehat{\vartheta}; \vartheta_1; \vartheta_0)$. Note that $app(\tau_0, \tau_1)$ is well-formed whenever τ_1 belongs to the domain of τ_0. For any prefunctional type τ_0 and any type τ_1 in the domain of τ_0, we define $app_\Delta(\tau_0, \tau_1)$ as $app(fun_\Delta(\tau_0), \tau_1)$. In order to deal with curried functions with several arguments, we inductively define $app_\Delta(\tau_0, \tau_1, \ldots, \tau_n)$ as

$$app_\Delta(app_\Delta(\tau_0, \tau_1, \ldots, \tau_{n-1}), \tau_n)$$

for $n \geq 2$. The following theorem shows that app_Δ is *covariant* in both arguments. As a consequence, app_Δ is a function on equivalence classes of types modulo equivalence, and the function $\lambda\tau_1.\ app_\Delta(\tau_0, \tau_1)$ is a monotonic type transformer which is total over the domain of τ_0. Together with theorem 40, this property shows that prefunctional types can be identified with monotonic type transformers which are total over their domain, even though the arrow type constructor on monotypes is contravariant in its first argument, which is yet another evidence that the never-ending covariance/contravariance controversy is unfounded [13].

Theorem 41 (App covariance) *The type application operator app_Δ is covariant in both arguments, that is*

$$\frac{\Delta \vdash \tau_1 \leq \tau_2 \qquad \Delta \vdash \tau_1' \leq \tau_2' \qquad \Delta \vdash \tau_2' \in dom_\Delta(\tau_2)}{\Delta \vdash app_\Delta(\tau_1, \tau_1') \leq app_\Delta(\tau_2, \tau_2')}$$

4.3 Data types

As stated in the introduction, the only run-time entities in $\mathrm{ML}_\leq$ are functions, methods and tagged records. The dynamic type of each of these run-time entities is of the form $\forall\vartheta : \kappa.\ d_C[\Theta_C]$. Such types are called *run-time types.* In particular, there is no run-time entity with the empty type $\forall\alpha.\ \alpha$. The declaration of a tagged record, which can only occur at toplevel, is of the form

$$\mathsf{data}\ d_C[\vartheta_C]\ \mathsf{is}\ 1\colon d_C^1\langle\vartheta_C\rangle; \ldots;\ n\colon d_C^n\langle\vartheta_C\rangle\ \mathsf{end}$$

where the tag d_C is a data type constructor of C, and for each field i, $d_C^i\langle\vartheta_C\rangle$ is a ϑ_C-closed monotype. The only effect of this declaration is to define $n+1$ operators: the *data constructor*[1] d_C, used to build records tagged by d_C, and the *data extractors* $d_C.1$, ..., $d_C.n$, used to access the fields of such records. These toplevel operators have the following types

$$\begin{cases} \dot{d}_C &=& \forall\vartheta_C.\ d_C^1\langle\vartheta_C\rangle \to \cdots \to d_C^n\langle\vartheta_C\rangle \to d_C[\vartheta_C] \\ \dot{d}_C^1 &=& \forall\vartheta_C.\ d_C[\vartheta_C] \to d_C^1\langle\vartheta_C\rangle \\ & \vdots & \\ \dot{d}_C^n &=& \forall\vartheta_C.\ d_C[\vartheta_C] \to d_C^n\langle\vartheta_C\rangle \end{cases}$$

except for $n = 0$, where the data constructor has type

$$\dot{d}_C \;=\; \forall\vartheta_C.\ \mathsf{unit} \to d_C[\vartheta_C]$$

For instance, in figure 1.1, cons-cells are declared as follows

$$\mathsf{data\ cons}[\alpha]\ \mathsf{is}\ 1\colon \alpha;\ 2\colon \mathsf{list}[\alpha]\ \mathsf{end};$$

so that the data constructor for cons-cells has type

$$\forall\alpha.\ \alpha \to \mathsf{list}[\alpha] \to \mathsf{cons}[\alpha]$$

[1] We assume that type constructor names and function names belong to different name spaces.

and the extractors associated to the head and the tail of cons-cells have the following types

$$\begin{cases} \forall\alpha.\ \mathsf{cons}[\alpha] \to \alpha \\ \forall\alpha.\ \mathsf{cons}[\alpha] \to \mathsf{list}[\alpha] \end{cases}$$

We assume that data type constructors have no subconstructors, so that we can think of data types as sets of records with the same tag. There are several advantages in considering that nil, cons, and scons are type constructors, instead of tags ignored by the type system. In particular, data extractors are total functions over their domain (whereas they are partial functions in ML), and pattern matching can be replaced by dynamic dispatch, since dispatching on cons is equivalent to doing pattern-matching in ML. For instance, the extractor associated to the tail of a cons-cell has type

$$\forall\alpha.\ \mathsf{cons}[\alpha] \to \mathsf{list}[\alpha]$$

in $\mathrm{ML}_{\leq}$, as opposed to

$$\forall\alpha.\ \mathsf{list}[\alpha] \to \mathsf{list}[\alpha]$$

in ML, but a method with the latter type can of course be defined in $\mathrm{ML}_{\leq}$. However, note that replacing pattern matching by dynamic dispatch cannot be done without the notion of subconstructors, which are not present in ML and complicate the type system somewhat, since polymorphic constrained types are needed in this context to maintain minimal typing (cf. theorem 48). We shall see in section 5.3 that the restriction that data type constructors have no subconstructors is also essential to ensure that a subtraction operation with type

$$\forall\alpha\colon \alpha \leq \mathsf{int}.\ \alpha \to \alpha \to \alpha$$

can actually be implemented as a method.

Note that records are not first-class citizens in $\mathrm{ML}_{\leq}$, that is, it is not possible to use "untagged" or "anonymous" records with arbitrary fields, and the type system does not depend on data types being implemented as records. For instance, data types like float need not be implemented as records. However, arbitrary records can always be defined by using dummy constructor classes with a unique data type constructor used as a dummy tag.

In order to ensure that the variance ∂_C of a class C, which is part of the specification of C, is an invariant of its implementations, we require that the following implication hold for every data type constructor d_C of C defined as above and for any disjoint ϑ_C and ϑ'_C

$$\forall\vartheta_C, \vartheta'_C.\ \vartheta_C \leq_C \vartheta'_C \models \bigwedge_{i\in[1,n]} d^i_C\langle\vartheta_C\rangle \leq d^i_C\langle\vartheta'_C\rangle$$

This condition ensures that the variance of each of the fields of d_C is compatible with the variance of C, but is also rather liberal in that it allows, for instance, a $(\otimes)$-variant class C to have an implementation like

$$\mathsf{data}\ d_C[\alpha]\ \mathsf{is}\ 1\colon \alpha\ \mathsf{end}$$

with a covariant field. As shown by the following theorem, this condition also ensures that, hopefully, the type of the field of a tagged record is a supertype of the type of the data stored into this fiel.

Theorem 42 (Data types) *Let Δ be an implicit well-formed context, and $\tau_1, \ldots, \tau_n$ be $n \geq 1$ well-formed types such that $app_\Delta(\dot{d}_C, \tau_1, \ldots, \tau_n)$ is well-formed w.r.t. Δ. Then for every $i \in [1, n]$*

$$\Delta \vdash \tau_i \leq app_\Delta(\dot{d}_C^i, app_\Delta(\dot{d}_C, \tau_1, \ldots, \tau_n))$$

Note that the resulting type can be a strict subtype. For instance, if we implement the data constructor d_C of a $(\otimes)$-variant class C as follows

$$\mathsf{data}\ d_C[\alpha]\ \mathsf{is}\ 1\colon \alpha \to \alpha;\ 2\colon \alpha\ \mathsf{end}$$

and suppose that e_1 has type $\tau_1 = \forall\,\emptyset.\ \mathsf{real} \to \mathsf{int}$ and e_2 has type $\tau_2 = \forall\,\emptyset.\ \mathsf{real}$, then, by the typing rules of figure 5.2, $d_C.1\,(d_C\ e_1\ e_2)$ has type

$$app_\Delta(\dot{d}_C^1, app_\Delta(\dot{d}_C, \tau_1, \tau_2))$$

that is

$$\forall \alpha, \beta\colon \mathsf{real} \to \mathsf{int} \leq \beta \to \beta \wedge \mathsf{real} \leq \beta \wedge d_C[\beta] \leq d_C[\alpha].\ \alpha \to \alpha$$

or else

$$\forall \alpha, \beta\colon \beta = \mathsf{real} \wedge \beta = \alpha.\ \alpha \to \alpha$$

which is equivalent to $\forall\,\emptyset.\ \mathsf{real} \to \mathsf{real}$, which, in turn, is a strict supertype of $\forall\,\emptyset.\ \mathsf{real} \to \mathsf{int}$. In a way, we could say that this type system gives every record a type which is a *predicative approximation* of its impredicative type

$$\langle\!\langle d_C; 1\colon \forall\,\emptyset.\ \mathsf{real} \to \mathsf{int}, 2\colon \forall\,\emptyset.\ \mathsf{real} \rangle\!\rangle$$

as a record.

Chapter 5

Type-checking

We now define an explicitly typed higher-order functional language using the type system of chapter 4, and define type-checking rules for this language. We start by considering the problem of type-checking a simple expression w.r.t. a fixed type structure $\mathcal{T}$. We show how methods can be defined and how to type-check them, and we show that this language has decidable minimal typing.

Extensions of this simple language supporting modules as well as the implementation of methods in several modules are studied in section 8.1. Type inference for an untyped version of this language, which can be seen as an extension of ML, is studied in section 7.2.

5.1 Programs

A program consists in two parts. The first part is a possibly empty, semicolon-separated list of mutually recursive declarations (figure 5.1) defining a type structure $\mathcal{T}$ together with the implementation of each data type of $\mathcal{T}$ as a tagged record. The class declaration declares a new ∂_C-variant constructor class C. We assume that class Arrow and class Unit are predefined. Type constructors t_C of a class C (resp. data type constructors d_C) are added to T_C (resp. D_C) using the type declaration (resp. data declaration).

The partial ordering of T_C is defined by declarations of the form $\mathsf{order}\ t_C \sqsubset t'_C$ which assert that t_C and t'_C are two distinct type constructors of C such that $t_C \sqsubseteq_C t'_C$. We assume that these declarations define a partial order (that is, we assume that there is no sequence $t^1_C, \ldots, t^n_C$, $n \geq 2$, such that $t^1_C = t^n_C$ and $t^1_C \sqsubset \ldots \sqsubset t^n_C$), and that data type constructors are minimal. As in section 4.3, data types are implemented as tagged records using declarations of the form

$$\mathsf{data}\ d_C[\vartheta_C]\ \mathsf{is}\ 1\colon d^1_C\langle\vartheta_C\rangle; \ldots;\ n\colon d^n_C\langle\vartheta_C\rangle\ \mathsf{end}$$

where we impose that the type $d^i_C\langle\vartheta_C\rangle$ of each field i be a ϑ_C-closed monotype compatible with the variance ∂_C of C.

The second part of a program is an expression e which must type-check w.r.t. $\mathcal{T}$. The syntax of expressions is given figure 5.1. An *expression variable* x is a non-empty sequence

Declarations		::=	$\mathsf{class}\ C[\partial_C]$	$(C \in \mathcal{T})$
		|	$\mathsf{type}\ t_C : C$	$(t_C \in T_C)$
		|	$\mathsf{data}\ d_C : C$	$(d_C \in D_C)$
		|	$\mathsf{order}\ t_C \sqsubset t'_C$	$(t_C \sqsubseteq_C t'_C)$
		|	$\mathsf{data}\ d_C[\vartheta_C]\ \mathsf{is}\ 1 : \theta; \ldots ; n : \theta\ \mathsf{end}$	
Patterns	π	::=	$\exists v.\ v \mid \exists \vartheta_C.\ t_C[\vartheta_C]$	
Abstractions	f	::=	$\mathsf{fun}\ \{\vartheta \mid \kappa\}\,(x : \theta) \Rightarrow e$	(*function*)
		|	$\mathsf{meth}\ \{\vartheta \mid \kappa\}\,(x : \theta) : \theta \Rightarrow [\pi \Rightarrow e; \ldots ; \pi \Rightarrow e]$	(*method*)
Expressions	e	::=	$x \mid f \mid e\ e$	
		|	$d_C.i$	(*i-th extractor*)
		|	$\rho(d_C; \vartheta_C; x, \ldots, x)$	(*record*)
		|	$\mathsf{let}\ x = e\ \mathsf{in}\ e\ \mathsf{end}$	(*simple let*)
		|	$\mathsf{letrec}\ x : \tau = f; \ldots ; x : \tau = f\ \mathsf{in}\ e\ \mathsf{end}$	(*recursive let*)

Figure 5.1: Programs, type declarations, and expressions

of alphanumeric characters starting by a letter, and is bound in lambda-expressions, let-expressions and recursive let-expressions. The definition of mutually recursive abstractions (that is, functions and methods) is achieved by recursive let-expressions, and explicit typing is required. In order to increase the readability of programs, we often split letrec-bindings of the form $x : \tau = f$ into a type declaration $x : \tau$ and a simple definition $x = f$. Explicit typing is not required in let-expressions.

An expression of the form $\mathsf{fun}\ \{\vartheta \mid \kappa\}\,(x : \theta) \Rightarrow e$ denotes a function taking a formal argument x with dynamic type in the domain $\exists\vartheta : \kappa.\ \theta$ of the function and returning e. Note that only the domain of functions is defined, not the type of their result, which is derived from e. The type variables ϑ are bound exactly as x, and are accessible in the body e of the function.

In what follows, we use the notation $\mathsf{fun}\ \{\vartheta\}\,(x : \theta) \Rightarrow e$ to denote the fully polymorphic function $\mathsf{fun}\ \{\vartheta \mid \mathit{true}\}\,(x : \theta) \Rightarrow e$ (parametric polymorphism) and $\mathsf{fun}\ (x : \theta) \Rightarrow e$ to denote the monomorphic function $\mathsf{fun}\ \{\,\emptyset \mid \mathit{true}\}\,(x : \theta) \Rightarrow e$.

As in section 4.3, we assume that the declaration of the implementation of d_C as a tagged record defines n built-in data extractors $d_C.1, \ldots,\ d_C.n$. However, we do not assume that the data constructor is a primitive operator. Rather, we assume that the data constructor is implemented in a global implicit recursive let-expression as follows

$$d_C : \forall\vartheta_C.\ d_C^1\langle\vartheta_C\rangle \to \cdots \to d_C^n\langle\vartheta_C\rangle \to d_C[\vartheta_C];$$
$$d_C = \mathsf{fun}\ \{\vartheta_C\}\,(x_1 : d_C^1\langle\vartheta_C\rangle) \Rightarrow \ldots \Rightarrow \mathsf{fun}\ (x_n : d_C^n\langle\vartheta_C\rangle) \Rightarrow \rho(d_C; \vartheta_C; x_1, \ldots, x_n);$$

when $n > 0$ and as

$$d_C : \forall\vartheta_C.\ \mathsf{unit} \to d_C[\vartheta_C];$$
$$d_C = \mathsf{fun}\ \{\vartheta_C\}\,(x : \mathsf{unit}) \Rightarrow \rho(d_C; \vartheta_C);$$

when $n = 0$. In these expressions, $\rho(d_C; \vartheta_C; x_1, \ldots, x_n)$ is a *record expression*. Record expressions can only appear, as above, in the implementation of data constructors. The only exception is $\rho(\mathsf{unit}; \emptyset)$, also written $()$, which implements the unit constant and can appear anywhere. This definition of the data constructor in terms of an atomic record constructor will simplify the operational semantics later on.

A method m is an expression of the form

$$\mathsf{meth}\ \{\vartheta \mid \kappa\}\,(x\colon \theta)\colon \theta' \Rightarrow [\pi_1 \Rightarrow e_1; \ldots; \pi_n \Rightarrow e_n]$$

where each π_i is a special kind of domain, called a *pattern*, defined figure 5.1. A pattern is well-formed if it is well-formed as a domain w.r.t. the trivial context. The existentially quantified variables of the i-th pattern are bound by the existential quantifier and are accessible in the scope e_i of the alternative.

Note that $\mathrm{ML}_{\leq}$ patterns do not coincide with ML patterns in that ML patterns are used both for pattern-matching and to bind variables to record components. Moreover, ML patterns can be nested, that is, the fields of tagged records can be recursively matched. In contrast, $\mathrm{ML}_{\leq}$ patterns are only used to perform dynamic dispatch on the *outermost* type constructor of types, and access to the fields of tagged records is performed by means of extractors. It is essential to understand that $\mathrm{ML}_{\leq}$ dispatches on the *run-time type* of values, whereas ML "dispatches" on *values* themselves. As a consequence, a ML pattern can look like this

$$\mathsf{cons}\,\{1 = \mathsf{pair}\,\{1 = x_1,\ 2 = x_2\},\ 2 = \mathsf{nil}\,\{\}\}$$

whereas a generalization of $\mathrm{ML}_{\leq}$ patterns allowing dynamic dispatch on nested type constructors will at best have the form $\exists \alpha_1, \alpha_2.\ \mathsf{cons}[\mathsf{pair}[\alpha_1, \alpha_2]]$ so that the fact that the tail of the list is empty cannot be directly expressed as a pattern.

Method m takes a parameter x with dynamic type τ in the domain δ of the method, defined as $\exists \vartheta\colon \kappa.\ \theta$, and returns e_i for the *most specific* pattern π_i such that τ belongs to π_i. We call this mechanism *dynamic dispatch*. Note that $\mathrm{ML}_{\leq}$ methods take a single argument. Section 8.2 addresses the problem of multi-methods.

The type of method m is $\forall \vartheta\colon \kappa.\ \theta \to \theta'$, and this specification must be matched by each alternative $\pi_i \Rightarrow e_i$ of m. For instance, assuming the type structure of figure 1.1, we could define a recursive method *dup* to duplicate lists as follows

$$
\begin{array}{l}
\mathit{dup}\colon \forall \alpha.\ \mathsf{list}[\alpha] \to \mathsf{list}[\alpha];\\
\mathit{dup} = \mathsf{meth}\ \{\alpha\}\,(x\colon \mathsf{list}[\alpha])\colon \mathsf{list}[\alpha] \Rightarrow [\\
\quad \exists\,\beta.\ \beta \Rightarrow x;\\
\quad \exists\,\beta.\ \mathsf{cons}[\beta] \Rightarrow \mathsf{cons}\ (\mathsf{cons}.1\ x)(\mathit{dup}\ (\mathsf{cons}.2\ x));\\
\quad \exists\,\beta.\ \mathsf{scons}[\beta] \Rightarrow \mathsf{scons}\ (\mathsf{scons}.1\ x)(\mathit{dup}\ (\mathsf{scons}.2\ x))(\mathsf{scons}.3\ x)\\
]
\end{array}
$$

Note that we often use "_" to denote the pattern $\exists v.\ v$ for some fresh type variable v, and "t_C" to denote the pattern $\exists \vartheta_C.\ t_C[\vartheta_C]$ for some fresh C-variable set ϑ_C. These abbreviations are very handy when the existentially quantified variables of the pattern π_i

of the i-th alternative of m do not occur in the body e_i of the alternative. For instance, we generally write the declaration of method *dup* as follows

$$\begin{array}{l} dup = \mathsf{meth}\ \{\alpha\}\ (x\colon \mathsf{list}[\alpha])\colon \mathsf{list}[\alpha] \Rightarrow [\\ \quad \_ \Rightarrow x; \\ \quad \mathsf{cons} \Rightarrow \mathsf{cons}\ (\mathsf{cons}.1\ x)(dup\ (\mathsf{cons}.2\ x)); \\ \quad \mathsf{scons} \Rightarrow \mathsf{scons}\ (\mathsf{scons}.1\ x)(dup\ (\mathsf{scons}.2\ x))(\mathsf{scons}.3\ x) \\ ] \end{array}$$

It is important to emphasize the fact that *the order in which the alternatives of a method are listed is not important.* This is in contrast with ML, where patterns are matched in sequence, starting from the first pattern. For instance, assuming a datatype declaration of the form

$$\mathsf{datatype}\ \mathsf{list}[\alpha] = \mathsf{nil} \mid \mathsf{cons}\ \mathsf{of}\ \alpha * \mathsf{list}[\alpha] \mid \mathsf{scons}\ \mathsf{of}\ \alpha * \mathsf{list}[\alpha] * \mathsf{int}$$

(where slist is not defined for obvious reasons) the above example could be written as follows in ML

$$\begin{array}{l} \mathsf{fun}\ dup\ (\mathsf{cons}\,(h,t)) = \mathsf{cons}\,(h, dup\ t) \\ \quad \mid\ dup\ (\mathsf{scons}\,(h,t,s)) = \mathsf{scons}\,(h, dup\ t, s) \\ \quad \mid\ dup\ x = x \end{array}$$

but swapping the first and the last alternatives would define *dup* as the identity function on lists. It is easy to see that sequential matching is not appropriate for an object-oriented language where methods can be implemented in several modules, and are thus naturally unordered. For the sake of simplicity, we choose, in this chapter, to present type-checking w.r.t. to a fixed type structure and to have "closed" methods, as in ML, but section 8.1 addresses the problem of defining a module system allowing "open" methods to be implemented in different modules while retaining a modular type-checking.

A method m with type τ is "correct" if it satisfies two conditions. First, each alternative of m must be type-correct, that is, the domain π_i of each alternative must be compatible with the domain δ of the method w.r.t Δ, and each alternative must have a type τ_i which, as a type transformer, must be below the restriction of τ to π_i. Second, the set $\pi_1, \ldots, \pi_n$ must be such that for every run-time type[1] τ in δ, there exists a minimum pattern π_i, $i \in [1, n]$, such that τ belongs to $\delta \wedge \pi_i$. This condition ensures the absence of "method not understood" or "match not exhaustive" run-time errors (completeness), as well as a "best match" algorithm for dynamic dispatch (non-ambiguity). Note that in the presence of modules, the second condition can only be checked at link-time, when all the modules are known, whereas the first condition can be checked separately for each module once and for all.

In order to formally state the first condition, we define the *restriction* $res(\tau, \pi')$ of a functional type τ of the form $\forall\vartheta\colon \kappa.\ \theta \to \theta''$ to a pattern π' of the form $\exists\vartheta'.\ \theta'$ compatible with the domain of τ as follows

$$res(\tau, \pi') \quad = \quad \forall v, \vartheta, \vartheta'\colon \kappa \wedge v \leq \theta \wedge v \leq \theta'.\ v \to \theta''$$

[1] C.f. section 4.3.

where v is a fresh variable, and ϑ and ϑ' are assumed to be disjoint. It is obvious that the domain of the restriction of τ to π' is the greatest lower bound of π' and of the domain of τ. Moreover, as shown by the following theorem, the restriction of a functional type τ, viewed as a type transformer, has the same behavior as τ on *run-time types* (but not on arbitrary types). As a consequence, a method m satisfying the first condition can be seen as a collection of functions f_i defined as follows

$$\mathsf{fun}\ \{\vartheta, \vartheta_i, v \mid \kappa \wedge \kappa_i \wedge v \leq \theta \wedge v \leq \theta_i\}\ (x\colon v) \Rightarrow e_i \qquad (v\ \textit{fresh})$$

such that the minimum type of each f_i is below the restriction of the type of m to π_i.

Theorem 43 (Restriction) *Let Δ be an implicit well-formed context, τ be a functional type, δ be the domain of τ, π' be a pattern compatible with δ, and τ'' be a run-time type both in δ and π'. Then τ'' belongs to $\delta \wedge \pi'$ and*

$$app(res(\tau, \pi'), \tau'') \ \doteq\ app(\tau, \tau'')$$

Note that this theorem would *not* hold if τ'' was not a run-time type, or if arbitrary domains were used as patterns. For instance, let τ be the functional type $\forall\alpha.\ \mathsf{list}[\alpha] \to \alpha$, δ be the domain of τ, δ' be the domain $\exists\,\emptyset.\ \mathsf{list}[\mathsf{unit}]$, which is compatible with δ, and τ'' be the run-time type $\forall\beta.\ \mathsf{nil}[\beta]$. Then τ'' belongs both to δ and δ', $app(\tau, \tau'')$ is equivalent to $\forall\alpha.\ \alpha$, but

$$\begin{aligned} app(res(\tau, \pi'), \tau'') &\doteq \forall\alpha, \beta\colon \mathsf{nil}[\beta] \leq \mathsf{list}[\alpha] \wedge \mathsf{nil}[\beta] \leq \mathsf{list}[\mathsf{unit}].\ \alpha \\ &\doteq \forall\alpha, \beta\colon \beta \leq \alpha \wedge \beta \leq \mathsf{unit}.\ \alpha \\ &\doteq \forall\alpha, \beta\colon \beta \leq \alpha \wedge \mathsf{unit} \leq \beta.\ \alpha \\ &\doteq \forall\,\emptyset.\ \mathsf{unit} \end{aligned}$$

so that allowing dynamic dispatch *inside* type constructors, as for the following method

$$\mathsf{meth}\ \{\alpha\}\ (x\colon \mathsf{list}[\alpha])\colon \alpha \Rightarrow [\exists\,\emptyset.\ \mathsf{list}[\mathsf{unit}] \Rightarrow ()]$$

would be unsafe in $\mathrm{ML}_{\leq}$ without further hypotheses on the partial ordering of constructor classes (e.g., lattice structure) and/or a stronger axiomatization of the implication. In particular, we believe that it is necessary to introduce an empty type $\perp$ or an empty constructor $\perp_C$ for each class C. In doing so, the empty list would have type $\mathsf{nil}[\perp]$, and the above counter-example would fail.

In order to enforce the second condition of correctness, we define the notion of *partition* as follows, and we impose that the set of patterns $\pi_1, \ldots, \pi_n$ of a method be a partition of the domain δ. The next theorem shows that this restriction on methods is sufficient to define a complete and non-ambiguous dynamic dispatch algorithm. Note that the notion of partition is decidable for a given type structure $\mathcal{T}$, but is *not* invariant w.r.t. admissible extensions of $\mathcal{T}$, so that this notion only really makes sense w.r.t. the "closed" link-time type structure.

Definition 44 (Partition) *Let Δ be a well-formed context, δ be a well-formed domain w.r.t. Δ, and $\pi_1, \ldots, \pi_n$, $n \geq 1$, be n well-formed patterns w.r.t. Δ. We say that $\pi_1, \ldots, \pi_n$ is a partition of δ w.r.t. Δ, written $\Delta \vdash \delta \bowtie \pi_1, \ldots, \pi_n$, if (assuming we identify equivalent patterns)*

(1) for every $i \in [1, n]$, π_i is compatible with δ w.r.t. Δ;

(2) for every distinct $i, j \in [1, n]$, π_i is distinct from π_j;

(3) for every data type constructor d_C in $\mathcal{T}$ such that $\delta[\Delta]$ and the pattern π defined as $\exists\vartheta_C.\ d_C[\vartheta_C]$ are compatible, the set $\{\pi_i \mid \exists i \in [1, n].\ \pi \leq \pi_i\}$ has a minimum element.

This definition shows that $\exists\alpha.\ \alpha$ is a partition of every well-formed domain. Note that the requirement that $\delta[\Delta]$ and $(\exists\vartheta_C.\ d_C[\vartheta_C])$ be compatible in condition (3) is more restrictive that the more intuitive requirement that δ and $(\exists\vartheta_C.\ d_C[\vartheta_C])$ be compatible w.r.t. Δ. For instance, assuming a constraint context Δ of the form $(\alpha\colon \mathit{true})$ and a domain of the form $\exists\,\emptyset.\ \alpha$, there is no data type d_C satisfying the latter condition, since

$$\forall\alpha.\ \mathit{true} \models \mathit{true} \wedge \mathit{true} \wedge d_C[\vartheta_C] \leq \alpha$$

is not derivable, whereas every d_C satisfies the former condition, since

$$\forall\emptyset.\ \mathit{true} \models \mathit{true} \wedge \mathit{true} \wedge d_C[\vartheta_C] \leq \alpha$$

is trivially derivable. However, it is easy to understand that the latter condition is desirable, since if δ is the domain of a "nested method", and α is the type of the formal of the enclosing function, then the method can potentially be called with any α.

Theorem 45 (Dynamic dispatch) *Let Δ be a well-formed context, δ be a well-formed domain w.r.t. Δ, τ be a closed run-time type in $\delta[\Delta]$, and $\pi_1, \ldots, \pi_n$, $n \geq 1$, be a partition of δ w.r.t. Δ. Then there exists a unique index $i \in [1, n]$, written $\mathit{disp}_\Delta(\tau; \delta; \pi_1, \ldots, \pi_n)$, such that (1) τ belongs to $\delta[\Delta] \wedge \pi_i$, and (2) π_i is a subdomain of π_j for every $j \in [1, n]$ such that τ belongs to $\delta[\Delta] \wedge \pi_j$.*

5.2 Minimal typing

A *typing context* is a pair $(\Delta; \Gamma)$, where Γ is a possibly empty, comma-separated list of bindings of the form $x\colon \tau$. Typing contexts are the environments in which expressions are type-checked. Intuitively, a typing context with a constraint context Δ of the form $(\vartheta\colon \kappa)$ assumes the existence of type and constructor variables ϑ such that κ holds. A typing context $(\Delta; \Gamma)$ is said to be *well-formed* if Δ is well-formed, every binding $x\colon \tau$ of x to τ in Γ is such that τ is well-formed w.r.t. Δ, and no expression variable x is bound more than once in Γ. The set of expression variables bound in Γ is called the domain of Γ.

For any well-formed typing context $(\Delta; \Gamma)$, distinct expression variables $x_1, \ldots, x_n$ and types $\tau_1, \ldots, \tau_n$ such that $x_1, \ldots, x_n$ are not bound in Γ and $\tau_1, \ldots, \tau_n$ are well-formed w.r.t. Δ, we denote by $\Delta; \Gamma[x_1\colon \tau, \ldots, x_n\colon \tau_n]$ the well-formed typing context $(\Delta; \Gamma, x_1\colon \tau_1, \ldots, x_n\colon \tau_n)$. Finally, a *subcontext* of a well-formed typing context $(\Delta; \Gamma)$ is a well-formed typing context $(\Delta; \Gamma')$ such that x has type τ in Γ if and only if x has type τ' in Γ' and τ' is a subtype of τ w.r.t. Δ.

An expression e is said to be well-formed w.r.t. a well-formed typing context $(\vartheta\colon \kappa; \Gamma)$ if (1) its free type and constructor variables belong to ϑ, (2) its free expression variables

$$\Delta; \Gamma\{x\colon \tau\} \vdash x\colon \tau \quad [Var]$$

$$\Delta; \Gamma \vdash \rho(d_C; \vartheta_C; x_1, \ldots, x_n)\colon (\forall \emptyset.\ d_C[\vartheta_C]) \quad [Rec]$$

$$\Delta; \Gamma \vdash d_C.i\colon (\forall \vartheta_C.\ d_C[\vartheta_C] \to d_C^i \langle \vartheta_C \rangle) \quad [Prj]$$

$$\frac{\Delta; \Gamma \vdash e\colon \tau \qquad \Delta \vdash \tau \leq \tau'}{\Delta; \Gamma \vdash e\colon \tau'}\ [Sub]$$

$$\frac{\Delta; \Gamma \vdash e_1\colon \tau_1 \qquad \Delta; \Gamma[x_1\colon \tau_1] \vdash e_0\colon \tau_0}{\Delta; \Gamma \vdash (\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ e_0\ \mathsf{end})\colon \tau_0}\ [Let]$$

$$\frac{\Delta; \Gamma[x_1\colon \tau_1, \ldots, x_n\colon \tau_n] \vdash e_i\colon \tau_i \qquad (0 \leq i \leq n)}{\Delta; \Gamma \vdash (\mathsf{letrec}\ x_1\colon \tau_1 = e_1; \ldots; x_n\colon \tau_n = e_n\ \mathsf{in}\ e_0\ \mathsf{end})\colon \tau_0}\ [LetRec]$$

$$\frac{\Delta; \Gamma \vdash e\colon \tau \qquad \Delta; \Gamma \vdash e'\colon \tau' \qquad \Delta \vdash \tau' \in dom_\Delta(\tau)}{\Delta; \Gamma \vdash (e\ e')\colon app_\Delta(\tau, \tau')}\ [App]$$

$$\frac{\Delta[v, \vartheta\colon \kappa \wedge v \leq \theta]; \Gamma[x\colon \forall \emptyset.\ v] \vdash e\colon (\forall \vartheta'\colon \kappa'.\ \theta') \quad (v\ fresh)}{\Delta; \Gamma \vdash (\mathsf{fun}\ \{\vartheta \mid \kappa\}\ (x\colon \theta) \Rightarrow e)\colon (\forall \vartheta, \vartheta', v\colon \kappa \wedge \kappa' \wedge v \leq \theta.\ v \to \theta')}\ [Fun]$$

$$\frac{\begin{array}{c}\delta = \exists \vartheta\colon \kappa.\ \theta \qquad \pi_i = \exists \vartheta_i.\ \theta_i \qquad \Delta \vdash \delta \bowtie \pi_1, \ldots, \pi_n \\ \Delta[v, \vartheta, \vartheta_i\colon \kappa \wedge v \leq \theta \wedge v \leq \theta_i]; \Gamma[x\colon \forall \emptyset.\ v] \vdash e_i\colon (\forall \emptyset.\ \theta') \quad (1 \leq i \leq n,\ v\ fresh)\end{array}}{\Delta; \Gamma \vdash (\mathsf{meth}\ \{\vartheta \mid \kappa\}\ (x\colon \theta)\colon \theta' \Rightarrow [\pi_1 \Rightarrow e_1; \ldots; \pi_n \Rightarrow e_n])\colon (\forall \vartheta\colon \kappa.\ \theta \to \theta')}\ [Meth]$$

Figure 5.2: Typing rules

belong to the domain of Γ, and (3) none of its variables is bound more than once. A well-formed expression e is said to be well-typed and to have type τ w.r.t. typing context $(\Delta; \Gamma)$ if $\Delta; \Gamma \vdash e\colon \tau$ can be derived from the rules of figure 5.2. Note that we assume that each d_C is associated to a fresh C-variable set ϑ_C so that the type associated to $d_C.i$ by rule *Prj* is always well-formed w.r.t. Δ. A well-formed expression e is said to be well-typed and to have minimal type τ in typing context $(\Delta; \Gamma)$ if e is well-typed, e has type τ, and τ is a subtype of all the types of e.

Rule *Sub* is the subsumption rule, similar to the one found in $F_\leq$. This rule is usually not present in ML-like type systems, where typing judgements have the form $\Gamma \vdash e\colon \theta$, and subtyping is hidden is a non-deterministic instanciation rule of the form $\Gamma\{x\colon \tau\} \vdash x\colon \theta$ whenever θ is an instance of τ. Similarly, the application rule *App* is closer in spirit from $F_\leq$ than from ML-like type systems, since it makes use of the application of a polytype to another, instead of making use of the application of a monotype to a monotype and resorting to the non-deterministic instanciation of polytypes. Intuitively, this rule trivially implies minimal typing for function application since dom_Δ is contravariant (theorem 40), and app_Δ is covariant in both arguments (theorem 41). However, note that type application is approximated in $\mathrm{ML}_{\leq}$ (cf. theorem 42) whereas it is supposedly exact in $\mathrm{F}_{\leq}$. The *Let* and *LetRec* rules are standard. Rule *Fun* is more interesting. Let f be the abstraction $\mathsf{fun}\ \{\vartheta \mid \kappa\}\ (x\colon \theta) \Rightarrow e$. The body e of f is type-checked in a context where x is known to have type $\forall\,\emptyset.\ \theta$ for at least one instanciation of ϑ such that κ holds. Expression e is thus well-typed if it type-checks using the property enforced by κ, but no more. Now, if e has type $\forall\vartheta'\colon \kappa'.\ \theta'$ in context $\Delta[\vartheta\colon \kappa]; \Gamma[x\colon \forall\,\emptyset.\ \theta]$, then f has type $\forall\vartheta, \vartheta'\colon \kappa \wedge \kappa'.\ \theta \to \theta'$ in context $(\Delta; \Gamma)$. Note that since $\forall\vartheta'\colon \kappa'.\ \theta'$ is necessarily well-formed w.r.t. $\Delta[\vartheta\colon \kappa]$, the domain of this type is (hopefully) equivalent to the explicitly declared domain $\exists\vartheta\colon \kappa.\ \theta$ of f. As an example, let f be the following function, assumed to be type-checked w.r.t. the trivial context

$$\mathsf{fun}\ \{\alpha \mid \alpha \leq \mathsf{real}\}\ (x\colon \alpha) \Rightarrow \mathsf{fun}\ (x'\colon \alpha) \Rightarrow \langle x, x'\rangle$$

Then e has type $\forall\,\emptyset.\ \alpha \to \langle \alpha, \alpha\rangle$ w.r.t. to typing context $(\alpha\colon \alpha \leq \mathsf{real}; x\colon \forall\,\emptyset.\ \alpha)$, meaning that given any α below real such that x is bound to an object with type α, e can be regarded as a monomorphic function from α to $\langle \alpha, \alpha\rangle$. Now, assuming f is called with integer 7 as actual argument, the dynamic type of e, that is, the polymorphic type by which its closure is tagged when created at run-time, is $\forall\alpha\colon (\mathsf{int} \leq \alpha \leq \mathsf{real}).\ \alpha \to \langle \alpha, \alpha\rangle$ w.r.t. the trivial context, so that this particular instance of e can be regarded as the following *toplevel*-defined function

$$\mathsf{fun}\ \{\alpha \mid \mathsf{int} \leq \alpha \leq \mathsf{real}\}\ (x'\colon \alpha) \Rightarrow \langle 7, x'\rangle$$

which is obviously well-typed since if the body of e type-checks w.r.t. the static typing context, it also type-checks w.r.t. the dynamic typing context $(\alpha\colon \mathsf{int} \leq \alpha \leq \mathsf{real}; x\colon \forall\,\emptyset.\ \alpha)$ which enforces stronger properties on α (cf. lemma 47). This fact will be at the heart of the subject reduction theorem.

Note that, abstractly speaking, we could have done without functions, since a function $\mathsf{fun}\ \{\vartheta \mid \kappa\}\ (x\colon \theta) \Rightarrow e$ whose type is known to be $\forall\vartheta, \vartheta'\colon \kappa \wedge \kappa'.\ \theta \to \theta'$ can always be replaced by the following method

$$\mathsf{meth}\ \{\vartheta, \vartheta' \mid \kappa \wedge \kappa'\}\ (x\colon \theta)\colon \theta' \Rightarrow [\exists v.\ v \Rightarrow e]$$

which has both the same type and the same run-time behavior. However, the advantage of functions is that their type does not need to be completely specified.

As mentioned in section 5.1, rule *Meth* imposes two conditions on well-typed methods. First, the set of patterns must be a partition of the domain of the method, to ensure that dynamic dispatch is both complete and non-ambiguous. Second, in order to ensure subject reduction, the body e_i of each alternative must have type $\forall\,\emptyset.\ \theta'$ in the context where the formal x is known to have both type θ and type θ_i, which is achieved by saying that x has type $\forall\,\emptyset.\ v$ and adding the conjuncts $v \leq \theta$ and $v \leq \theta_i$ to the constraint of the typing context.

The following lemmas and theorem show that $\mathrm{ML}_{\leq}$ has decidable minimal typing.

Lemma 46 (Typing context) *Let $(\Delta; \Gamma)$ be a well-formed typing context and Δ' be a context such that Δ' is well-formed w.r.t. Δ. Then $\Delta[\Delta']; \Gamma$ is a well-formed typing context, and for every subcontext $(\Delta; \Gamma')$ of $(\Delta; \Gamma)$, $\Delta[\Delta']; \Gamma'$ is a subcontext of $\Delta[\Delta']; \Gamma$.*

Lemma 47 (Strengthening) *Let* $(\Delta;\Gamma)$ *be a well-formed typing context, e be a well-typed expression with type* τ *w.r.t.* $(\Delta;\Gamma)$*, and* Δ' *be a context of the form* $\vartheta'\colon\kappa'$ *(not necessarily well-formed) such that* ϑ' *is fresh and* $\Delta\wedge\Delta'$ *is well-formed. Then e is well-typed and has type* τ *w.r.t.* $(\Delta\wedge\Delta';\Gamma)$*.*

Theorem 48 (Minimal typing) *Let* $(\Delta;\Gamma)$ *be an implicit well-formed typing context and e be a well-formed expression w.r.t.* $(\Delta;\Gamma)$*. The existence of a type* τ *such that e is well-typed and has type* τ *is decidable. Let e be well-typed and have type* τ*. Then* τ *is well-formed, e has a minimal type and the determination of this minimal type is decidable. Moreover, e is well-typed w.r.t. any subcontext of* $(\Delta;\Gamma)$ *and its minimal type w.r.t. such a context is a subtype of* τ *w.r.t.* Δ*.*

5.3 Examples

We now illustrate some of the subtleties involved in the type-checking of methods. First, we remark that since the type structure of figure 1.1 only defines three data type constructors nil, cons, and scons compatible with the domain $\exists\alpha.\ \mathsf{list}[\alpha]$ of method *dup* defined in the previous section, we may have chosen to implement the first alternative of *dup* as follows

$$\mathsf{nil} \Rightarrow \mathsf{nil}\,()$$

which is well-typed since

$$(\alpha,\beta,\gamma\colon \gamma\leq \mathsf{list}[\alpha], \mathsf{nil}[\beta]) \vdash (\forall\varepsilon.\ \mathsf{nil}[\varepsilon]) \leq (\forall\,\emptyset.\ \mathsf{list}[\alpha])$$

obviously holds. Similarly, since the type of *dup* only requires that its output be a list of the same sort as its input, which is always the case of the empty list, we may have chosen to implement the same alternative as follows

$$\_ \Rightarrow \mathsf{nil}\,()$$

Also, the "natural" type of *dup*, namely $\forall\alpha.\ \mathsf{list}[\alpha]\rightarrow\mathsf{list}[\alpha]$, is rather imprecise, since it does not reflect the fact that the duplication of the empty list is the empty list, that the duplication of a cons is a cons, and that the duplication of a sized cons is a sized cons. In other words, it would be nice to be able to impose that $(dup\ e)$ has type β whenever e has type β and $\beta\leq\mathsf{list}[\alpha]$ for some α. This is possible in $\mathrm{ML}_{\leq}$, and *dup* can also be defined as follows

$$\begin{array}{l}
dup\colon \forall\alpha,\beta\colon \beta\leq\mathsf{list}[\alpha].\ \beta\rightarrow\beta\\
dup = \mathsf{meth}\ \{\alpha,\beta\,|\,\beta\leq\mathsf{list}[\alpha]\}\,(x\colon\beta)\colon\beta\Rightarrow [\\
\quad \mathsf{nil}\Rightarrow\mathsf{nil}\ ();\\
\quad \mathsf{cons}\Rightarrow\mathsf{cons}\ (\mathsf{cons}.1\ x)\ (dup\ (\mathsf{cons}.2\ x));\\
\quad \mathsf{scons}\Rightarrow\mathsf{scons}\ (\mathsf{scons}.1\ x)\ (dup\ (\mathsf{scons}.2\ x))\ (\mathsf{scons}.3\ x)\\
]
\end{array}$$

with a very precise type which is in fact a strict subtype of the natural type of *dup*. This new specification is also an example of why the minimality of data type constructors is essential. Indeed, let us show that the first alternative of the new implementation of *dup* is well-typed. This amounts to showing that

$$(\alpha, \beta, \gamma, \varepsilon \colon \varepsilon \leq \beta \wedge \varepsilon \leq \mathsf{nil}[\gamma] \wedge \beta \leq \mathsf{list}[\alpha]) \vdash (\forall \rho.\ \mathsf{nil}[\rho]) \leq (\forall \emptyset.\ \beta)$$

that is to say

$$\forall \alpha, \beta, \gamma, \varepsilon.\ \varepsilon \leq \beta \wedge \varepsilon \leq \mathsf{nil}[\gamma] \wedge \beta \leq \mathsf{list}[\alpha] \models \mathsf{nil}[\rho] \leq \beta$$

which is derivable, since by rules *VElim* and *CMin*, the left-hand side of the implication implies the existence of η such that $\varepsilon = \mathsf{nil}[\eta]$, and consequently, the existence of η such that $\mathsf{nil}[\eta] \leq \beta$, which implies the right-hand side thanks to rule *VIntro*. In other words, the first alternative of *dup* is well-typed only because we know that every input run-time object in the domain $\exists \gamma.\ \mathsf{nil}[\gamma]$ is necessarily the empty list, so that the new empty list which is built in the body of the alternative is known to have the very same type as the input object. On a similar track, note that this stronger specification does not allow the first alternative to be written as follows

$$\_ \Rightarrow \mathsf{nil}\,()$$

since it may be the case that the input is not the empty list (in some extension of the type structure) whereas the output is the empty list. Formally, the following does not hold

$$(\alpha, \beta, \gamma, \varepsilon \colon \varepsilon \leq \beta \wedge \varepsilon \leq \gamma \wedge \beta \leq \mathsf{list}[\alpha]) \vdash (\forall \rho.\ \mathsf{nil}[\rho]) \leq (\forall \emptyset.\ \beta)$$

Another interesting example is the method *conc* to concatenate two lists defined as follows (assuming that the only data type constructors are nil and cons)

$$
\begin{aligned}
&\mathit{conc} \colon \forall \alpha, \beta \colon \alpha \leq \mathsf{list}[\beta].\ \alpha \to \alpha \to \alpha;\\
&\mathit{conc} = \mathsf{meth}\ \{\alpha, \beta \mid \alpha \leq \mathsf{list}[\beta]\}\ (x \colon \alpha) \colon \alpha \to \alpha \Rightarrow [\\
&\quad \mathsf{nil} \Rightarrow \mathsf{fun}\ (x' \colon \alpha) \Rightarrow x';\\
&\quad \mathsf{cons} \Rightarrow \mathsf{fun}\ (x' \colon \alpha) \Rightarrow \mathsf{cons}\ (\mathsf{cons}.1\ x)\ (\mathit{conc}\ (\mathsf{cons}.2\ x)\ x')\\
&]
\end{aligned}
$$

with a type showing, in particular, that the concatenation of two empty lists is an empty list. Of course, *conc* could also be given its standard ML type $\forall \alpha.\ \mathsf{list}[\alpha] \to \mathsf{list}[\alpha] \to \mathsf{list}[\alpha]$, giving more freedom for its implementation, but also restricting the contexts in which it can be used.

A more real-life example is given figure 5.3. The type structure of this example is a numerical hierarchy with two uncomparable maximum types real and dyadic corresponding to real and 2-adic numbers. The data types are floating points (e.g., IEEE floating point numbers), binary numbers (e.g., periodic 2-adic numbers), zero, positive and negative numbers. The signature of the subtraction operator is

$$\forall \alpha \colon \mathsf{int} \leq \alpha.\ \alpha \to \alpha \to \alpha$$

that is, reals and 2-adics cannot be added together, and the type of a subtraction is always at least an integer.

```
class Num[];

type int: Num;
type real: Num;
type dyadic: Num;

data neg: Num;
data pos: Num;
data zero: Num;
data float: Num;
data bin: Num;

order neg ⊏ int;
order zero ⊏ int;
order zero ⊏ dyadic;
order pos ⊏ int;
order int ⊏ real;
order int ⊏ dyadic;
order float ⊏ real;
order bin ⊏ dyadic;

data neg[] is ··· end;
data pos[] is ··· end;
data zero[] is ··· end;
data float[] is ··· end;
data bin[] is ··· end;

letrec
   // Extern functions
   subInt: int → int → int;
   subFloat: float → float → float;
   subBin: bin → bin → bin;

   neg: ∀α: int ≤ α. α → α;
   neg = meth {α | int ≤ α} (x: α): α ⇒ [
      bin ⇒ bin (···);
      float ⇒ float (···);
      neg ⇒ pos (···);
      pos ⇒ neg (···);
      zero ⇒ zero (···)
   ];

   toFloat: ∀α: float ≤ α. α → float;
   toFloat = meth {α | float ≤ α} (x: α): float ⇒ [
      float ⇒ x;
      int ⇒ float (···)
   ];

   toBin: ∀α: bin ≤ α. α → bin;
   toBin = meth {α | bin ≤ α} (x: α): bin ⇒ [
      bin ⇒ x;
      int ⇒ bin (···)
   ];

   sub: ∀α: int ≤ α. α → α → α;
   sub = meth {α | int ≤ α} (x: α): α → α ⇒ [
      float ⇒ fun (x′: α) ⇒ subFloat x (toFloat x′);
      bin ⇒ fun (x′: α) ⇒ subBin x (toBin x′);
      int ⇒ meth (x′: α): α ⇒ [
         _ ⇒ sub (neg x′) (neg x);
         int ⇒ subInt x x′
      ]
   ];
in
   sub (neg(···)) (float(···))
```

Figure 5.3: Numerical operators

// *Constructor class*
class $C[\,]$;

// *Type constructor*
type $c\colon C$;

// *Data type constructors*
data $c_1 : C$;
data $c_2 : C$;

// *Implementations*
data c_1 is end;
data c_2 is end;

// *Subtyping*
order $c_1 \sqsubset c$;
order $c_2 \sqsubset c$;

letrec
$\quad m\colon \forall\alpha\colon \alpha \leq c.\ \alpha \to \alpha \to \alpha;$
$\quad m = \mathsf{meth}\ \{\alpha \mid \alpha \leq c\}\ (x\colon \alpha)\colon \alpha \to \alpha \Rightarrow [$
$\quad\quad c_1 \Rightarrow \mathsf{meth}\ (x'\colon \alpha)\colon \alpha \Rightarrow [$
$\quad\quad\quad c_1 \Rightarrow c_1\ ();$ // *Ok*
$\quad\quad\quad c_2 \Rightarrow c_1\ ()$ // *Wrong*
$\quad\quad];$
$\quad\quad c_2 \Rightarrow \mathsf{meth}\ \{\alpha' \mid \alpha' \leq c \wedge \alpha \leq \alpha'\}\ (x'\colon \alpha')\colon \alpha' \Rightarrow [$
$\quad\quad\quad c_1 \Rightarrow c_2\ ();$ // *Ok*
$\quad\quad\quad c_2 \Rightarrow c_2\ ()$ // *Ok*
$\quad\quad]$
$\quad]$
in
$\quad m\ (c_1\ ())\ (c_2\ ())$

Figure 5.4: Curried methods

5.4 Curried methods

It is important to point out that a type like $\forall\alpha\colon \alpha \leq \mathsf{point}.\ \alpha \to \alpha \to \mathsf{bool}$, which is in fact equivalent to $\mathsf{point} \to \mathsf{point} \to \mathsf{bool}$, is definitely *not* the type of functions which first take a point as argument, and then take a second argument whose dynamic type is below the dynamic type of the first argument! As a matter of fact, this remark is strongly related to the form of curried methods. We argue that the "most general" implementation of a curried method m with a type τ of the form

$$\forall\vartheta : \kappa.\ \theta_1 \to \theta_2 \to \theta_3$$

is of the following form

$$\mathsf{meth}\ \{\vartheta \mid \kappa\}\ (x_1 : \theta_1)\colon \theta_2 \to \theta_3 \Rightarrow [\cdots \Rightarrow \mathsf{meth}\ \{\vartheta' \mid \kappa' \wedge \theta_1 \leq \theta_1'\}\ (x_2 : \theta_2')\colon \theta_3' \Rightarrow [\cdots]\,]$$

where $\tau' = \forall\vartheta'\colon \kappa'.\ \theta_1' \to \theta_2' \to \theta_3'$ is a fresh renaming of τ. To understand why, note that the context in which the inner method is type-checked assumes the existence of variables ϑ such that κ holds and x_1 has type $\forall\,\emptyset.\ \theta_1$. In other words, the inner method is in some sense equivalent to the closure of the application of m to an argument with type $\forall\,\emptyset.\ \theta_1$ (cf. section 5.2), so that its type should intuitively be the application of τ' to $\forall\,\emptyset.\ \theta_1$, that is to say

$$\forall\vartheta'\colon \kappa' \wedge \theta_1 \leq \theta_1'.\ \theta_2' \to \theta_3'$$

which is obviously a subtype of $\forall\,\emptyset.\ \theta_2 \to \theta_3$ w.r.t. context $\Delta = (\vartheta\colon \kappa)$. Note that the domain of the inner method w.r.t. Δ is thus $\exists\vartheta'\colon \kappa' \wedge \theta_1 \leq \theta_1'.\ \theta_2'$, which is a superdomain

$$
\begin{array}{l}
\mathit{conc}\colon \forall\alpha,\beta\colon \alpha \le \mathsf{list}[\beta].\ \alpha \to \alpha \to \alpha;\\
\mathit{conc} = \mathsf{meth}\ \{\alpha_1,\beta_1 \mid \alpha_1 \le \mathsf{list}[\beta_1]\}\,(x_1\colon \alpha_1)\colon \alpha_1 \to \alpha_1 \Rightarrow [\\
\quad \mathsf{nil} \Rightarrow \mathsf{fun}\ (x_2\colon \alpha_1) \Rightarrow x_2;\\
\quad \mathsf{cons} \Rightarrow \mathsf{meth}\ \{\alpha_2,\beta_2 \mid \alpha_2 \le \mathsf{list}[\beta_2] \wedge \alpha_1 \le \alpha_2\}\,(x_2\colon \alpha_2)\colon \alpha_2 \Rightarrow [\\
\quad\quad \mathsf{nil} \Rightarrow x_1;\\
\quad\quad \_ \Rightarrow \mathsf{cons}\ (\mathsf{cons}.1\ x_1)\ (\mathit{conc}\ (\mathsf{cons}.2\ x_1)\ x_2)\\
\quad];\\
\quad \mathsf{scons} \Rightarrow \mathsf{meth}\ \{\alpha_2,\beta_2 \mid \alpha_2 \le \mathsf{list}[\beta_2] \wedge \alpha_1 \le \alpha_2\}\,(x_2\colon \alpha_2)\colon \alpha_2 \Rightarrow [\\
\quad\quad \mathsf{nil} \Rightarrow x_1;\\
\quad\quad \mathsf{cons} \Rightarrow \mathsf{cons}\ (\mathsf{scons}.1\ x_1)\ (\mathit{conc}\ (\mathsf{scons}.2\ x_1)\ x_2);\\
\quad\quad \mathsf{scons} \Rightarrow \mathsf{let}\ x_3 = \mathit{conc}\ (\mathsf{scons}.2\ x_1)\ x_2\ \mathsf{in}\ \mathsf{scons}\ (\mathsf{scons}.1\ x_1)\ x_3\ (\mathit{inc}\ (\mathit{size}\ x_3))\\
\quad]\\
]
\end{array}
$$

Figure 5.5: Concatenation of sized lists

of $\exists\,\emptyset.\ \theta_2$, so that the most general form of m makes as few assumptions as possible on the arguments of the inner method. Of course, in some cases, it is possible to write m using the following form, which is less general, but is also simpler and more intuitive

$$\mathsf{meth}\ \{\vartheta \mid \kappa\}\,(x_1\colon \theta_1)\colon \theta_2 \to \theta_3 \Rightarrow [\cdots \Rightarrow \mathsf{meth}\ (x_2\colon \theta_2)\colon \theta_3 \Rightarrow [\cdots]\,]$$

This is the case, in particular, for method sub of figure 5.3 which is of the form

$$\mathsf{meth}\ \{\alpha \mid \mathsf{int} \le \alpha\}\,(x\colon \alpha)\colon \alpha \to \alpha \Rightarrow [\,\mathsf{int} \Rightarrow \mathsf{meth}\ (x'\colon \alpha)\colon \alpha \Rightarrow [\,\mathsf{int} \Rightarrow \mathit{subInt}\ x\ x']]$$

instead of the more systematic form

$$
\begin{array}{l}
\mathsf{meth}\ \{\alpha \mid \mathsf{int} \le \alpha\}\,(x\colon \alpha)\colon \alpha \to \alpha \Rightarrow [\\
\quad \mathsf{int} \Rightarrow \mathsf{meth}\ \{\alpha' \mid \mathsf{int} \le \alpha' \wedge \alpha \le \alpha'\}\,(x'\colon \alpha')\colon \alpha' \Rightarrow [\,\mathsf{int} \Rightarrow \mathit{subInt}\ x\ x']\\
]
\end{array}
$$

However, this simple form could not have been used had the inner method dispatched on two different patterns, instead of dispatching on int twice. As a counter-example, consider the first alternative of method m of figure 5.4. The body e of this alternative, which is itself a method, type-checks w.r.t. the following typing environment $(\Delta; \Gamma)$

$$(\alpha,\beta\colon \alpha \le c \wedge \beta \le \alpha \wedge \beta \le c_1);\ (x\colon \forall\,\emptyset.\ \beta)$$

such that Δ is well-formed w.r.t. the trivial context True. However, the second alternative of e is ill-typed, since the context $(\beta'\colon \beta' \le \alpha \wedge \beta' \le c_2)$ in which it would be type-checked is not well-formed w.r.t. Δ, as shown by the following implication, which is not derivable

$$\forall\alpha,\beta.\ \alpha \le c \wedge \beta \le \alpha \wedge \beta \le c_1 \models \beta' \le \alpha \wedge \beta' \le c_2$$

(remember that $\beta \leq c_1$ implies $\beta = c_1$, since c_1 is a data type constructor, and is thus minimal). Intuitively, the outer method assumes the existence of some α between c_1 and c, whereas the inner method assumes that α is above c_2, a property which is not true for every α between c_1 and c (e.g., for $\alpha = c_1$). In contrast, the second alternative of the body of the second alternative of m is well-typed since the context in which the body of this alternative is type-checked, namely

$$\alpha', \beta' \colon \alpha' \leq c \wedge \alpha \leq \alpha' \wedge \beta' \leq \alpha \wedge \beta' \leq c_2$$

is well-formed w.r.t. Δ. Intuitively, the inner method now assumes the existence of some α' such that α' is above both α and c_2, and is also below c, which trivially holds for every α between c_1 and c (e.g., for $\alpha' = c$).

As a last example, figure 5.5 shows how the *conc* method of section 5.3 can be implemented in the presence of sized lists while maintaining a very precise specification (i.e., type) enforcing that the concatenation of two empty lists be an empty list, that the concatenation of two sized lists be a sized list, and that the concatenation of two regular lists be a regular list.

Finally, note that in the absence of constraints and primitive subtyping (as in ML), the most general implementation of a curried method m with a type τ of the form $\forall\vartheta \colon \kappa.\ \theta_1 \to \theta_2 \to \theta_3$ has the following form

$$\mathsf{meth}\ \{\vartheta\}\,(x_1 \colon \theta_1) \colon \theta_2 \to \theta_3 \Rightarrow [\cdots \Rightarrow \mathsf{meth}\ \{\vartheta' \mid \theta_1 = \theta_1'\}\,(x_2 \colon \theta_2') \colon \theta_3' \Rightarrow [\cdots]]$$

where $\tau' = \forall\vartheta'.\ \theta_1' \to \theta_2' \to \theta_3'$ is a fresh renaming of τ. That is to say, m can be implemented as follows

$$\mathsf{meth}\ \{\vartheta\}\,(x_1 \colon \theta_1) \colon \theta_2 \to \theta_3 \Rightarrow [\cdots \Rightarrow \mathsf{meth}\ (x_2 \colon \theta_2) \colon \theta_3 \Rightarrow [\cdots]]$$

in pretty much the same way curried methods are implemented in explicitly typed versions of ML. Unfortunately, this simple and intuitive form does not carry over to languages with primitive subtyping.

Chapter 6

Operational semantics

We now define the operational semantics of the language. For the sake of simplicity, we choose a strict semantics instead of a lazy semantics. This is because strictness is more natural in a language with dynamic dispatch where arguments have to be evaluated before dynamic dispatch can be performed on their dynamic type. Of course, a mixture of strictness and laziness could also be used. In particular, we assume the existence of a lazy conditional with type $\forall\alpha.\ \mathsf{bool} \to \alpha \to \alpha \to \alpha$. The operational semantics of $\mathrm{ML}_{\leq}$ departs from the standard approach in that it uses the static type system to tag run-time values by their minimal type. Some rules thus "call" the type-checker to determine this minimal type w.r.t. to the run-time environment. The decidability of minimal typing is thus essential for the operational semantics to make sense. However, the semantics does not actually *use* minimal types, since dynamic dispatch is performed via the tag d_C of records, which can be seen as an *abstraction* of their minimal type $\forall\vartheta\colon\kappa.\ d_C[\Theta_C]$. Consequently, actual implementations can do without run-time types. Note that this would not be the case if dispatching was allowed inside type constructors, which is a natural extension of the system.

The key point of the operational semantics is that every expression e is evaluated both in a run-time environment Ω, giving both the value and the minimal type of variables, and in a constraint context Δ, so that when e evaluates to a run-time value, this value is tagged by the closure $\tau[\Delta]$ of the minimal type τ of e w.r.t. $(\Delta;\Omega)$. The crucial observation to establish subject reduction is that Δ is always a "strengthening" of the constraint context in which e has been type-checked statically. The following lemma relates well-formed types to their closure.

Lemma 49 (Closure) *Let $\Delta = \widehat{\vartheta}\colon\widehat{\kappa}$ be a well-formed context, $\tau = \forall\vartheta\colon\kappa.\ \theta$ be a well-formed type w.r.t. the trivial context such that ϑ and $\widehat{\vartheta}$ are disjoint, and τ' be a well-formed type w.r.t. Δ. Then (1) $\tau[\Delta]$ is closed, well-formed, and is equivalent to τ, (2) $\tau'[\Delta]$ is closed and well-formed, and (3) τ is a subtype of τ' w.r.t. Δ if and only if τ is a subtype of $\tau'[\Delta]$ w.r.t. the trivial context.*

A *run-time environment* Ω is a possibly empty, comma-separated list of bindings of the form $x = \omega\colon\tau$ denoting that the expression variable x has run-time value ω and closed run-time type τ. We assume that expression variables are bound at most once

$$\frac{\Delta; \Omega \vdash d_C.i\colon \tau}{\Omega \vdash \mathsf{prj}(d_C, i)\colon \tau}\ [\mathit{PrjType}]$$

$$\frac{\Delta; \Omega \vdash f\colon \tau \qquad \mathit{True} \vdash \tau[\Delta] \leq \tau'}{\Omega \vdash \mathsf{cls}(\Delta, f)\colon \tau'}\ [\mathit{ClsType}]$$

$$\frac{\Delta = \mathit{True} \qquad \Delta \vdash \mathit{app}_\Delta(d_C, \tau_1, \ldots, \tau_n) \qquad \Omega \vdash \omega_i\colon \tau_i \qquad (1 \leq i \leq n)}{\Omega \vdash \mathsf{rec}(d_C; \omega_1\colon \tau_1, \ldots, \omega_n\colon \tau_n)\colon \mathit{app}_\Delta(d_C, \tau_1, \ldots, \tau_n)}\ [\mathit{RecType}]$$

Figure 6.1: Well-typedness of run-time values

in run-time environments. Possible run-time values are closures $\mathsf{cls}(\Delta, f)$, tagged records $\mathsf{rec}(d_C; \omega_1\colon \tau_1, \ldots, \omega_n\colon \tau_n)$, and the projection $\mathsf{prj}(d_C, i)$ on the i-th component of tagged records. The free expression variables in closures are interpreted w.r.t. a run-time environment. Consequently, the evaluation process not only returns a run-time value and its type, but also the run-time environment in which to interpreted it[1]. More formally, figure 6.2 defines the judgement $\Delta; \Omega \vdash e \to \omega\colon \tau,\ \Omega'$, which reads: in constraint context Δ and run-time environment Ω, expression e evaluates to run-time value ω with run-time type τ in environment Ω'. Note that we sometimes omit τ when we have no use for it. In order to avoid the capture of variable names, we assume that retrieving a typed run-time value $\omega\colon \tau$ from a run-time environment Ω always creates fresh copies of ω and τ, that is, copies of ω and τ where all the type, constructor and expression variables which are not in the domain of Ω are renamed to fresh variables. For any run-time environment Ω, distinct expression variables $x_1, \ldots, x_n$, run-time values $\omega_1, \ldots, \omega_n$ and types $\tau_1, \ldots, \tau_n$ such that $x_1, \ldots, x_n$ are not bound in Ω, we denote by $\Omega[x_1 = \omega_1\colon \tau_1, \ldots, x_n = \omega_n\colon \tau_n]$ the run-time environment $(\Omega, x_1 = \omega_1\colon \tau_1, \ldots, x_n = \omega_n\colon \tau_n)$.

We say that an expression e is well-typed and has type τ w.r.t. $(\Delta; \Omega)$, written $\Delta; \Omega \vdash e\colon \tau$, if e has minimal type τ w.r.t. $(\Delta; \Gamma)$, where Γ is obtained by replacing every binding $x = \omega\colon \tau$ in Ω by the binding $x\colon \tau$. We say that a run-time value ω has type τ w.r.t. Ω if $\Omega \vdash \omega\colon \tau$ can be derived from the rules of figure 6.1. We say that a run-time environment Ω is *well-formed*, written $\vdash \Omega$, if every binding $x = \omega\colon \tau$ in Ω is such that ω has type τ w.r.t. Ω.

It is easy to see that the rules of figure 6.2 define an abstract machine with a global run-time environment Ω and a stack of pending evaluations $(\Delta_i, e_i)_{0 \leq i \leq n}$. Most of the rules are straightforward. The only remark is that rule *MethApp* performs dynamic dispatch by selecting the most specific alternative i such that the run-time type τ' of the actual parameter belongs both to the *closure* $\delta[\Delta]$ of the domain δ of the method w.r.t. the context Δ in its closure, and to the domain π_i of the alternative.

An abstract machine is said to be well-formed if its global run-time environment Ω is well-formed and every pending evaluation (Δ_i, e_i) is such that e_i is well-typed w.r.t.

[1] Doing so is admittedly not very elegant, and leads to the use of ever growing run-time environments. Nonetheless, we do not put Ω in closures to avoid the use of infinite, recursively defined syntactic run-time environments in the treatment of recursive let-expressions.

$$\Delta;\Omega\{x=\omega\colon\tau\}\vdash x\to\omega\colon\tau,\,\Omega\quad[\mathit{VarVal}]$$

$$\frac{\Delta;\Omega\{x_1=\omega_1\colon\tau_1,\ldots,x_n=\omega_n\colon\tau_n\}\vdash\rho(d_C;\vartheta_C;x_1,\ldots,x_n)\colon\tau}{\Delta;\Omega\vdash\rho(d_C;\vartheta_C;x_1,\ldots,x_n)\to\mathsf{rec}(d_C;\omega_1\colon\tau_1,\ldots,\omega_n\colon\tau_n)\colon\tau[\Delta],\,\Omega}\,[\mathit{RecVal}]$$

$$\frac{\Delta;\Omega\vdash d_C.i\colon\tau}{\Delta;\Omega\vdash d_C.i\to\mathsf{prj}(d_C,i)\colon\tau,\,\Omega}\,[\mathit{PrjVal}]$$

$$\frac{\Delta;\Omega\vdash f\colon\tau}{\Delta;\Omega\vdash f\to\mathsf{cls}(\Delta,f)\colon\tau[\Delta],\,\Omega}\,[\mathit{ClsVal}]$$

$$\frac{\begin{array}{c}\Delta;\Omega\vdash e_1\to\omega_1\colon\tau_1,\,\Omega_1\\ \Delta;\Omega_1[x_1=\omega_1\colon\tau_1]\vdash e_0\to\omega_0\colon\tau_0,\,\Omega_0\end{array}}{\Delta;\Omega\vdash(\mathsf{let}\ x_1=e_1\ \mathsf{in}\ e_0\ \mathsf{end})\to\omega_0\colon\tau_0,\,\Omega_0}\,[\mathit{LetVal}]$$

$$\frac{\Delta;\Omega[x_1=\mathsf{cls}(\Delta,f_1)\colon\tau_1[\Delta],\ldots,x_n=\mathsf{cls}(\Delta,f_n)\colon\tau_n[\Delta]]\vdash e_0\to\omega_0\colon\tau_0,\,\Omega_0}{\Delta;\Omega\vdash(\mathsf{letrec}\ x_1\colon\tau_1=f_1,\ldots,x_n\colon\tau_n=f_n\ \mathsf{in}\ e_0\ \mathsf{end})\to\omega_0\colon\tau_0,\,\Omega_0}\,[\mathit{LetRecVal}]$$

$$\frac{\begin{array}{c}\widehat{\Delta};\widehat{\Omega}\vdash\widehat{e}\to\mathsf{prj}(d_C,i),\,\Omega\\ \widehat{\Delta};\Omega\vdash\widehat{e}\,'\to\mathsf{rec}(d_C;\omega_1\colon\tau_1,\ldots,\omega_n\colon\tau_n),\,\Omega'\end{array}}{\widehat{\Delta};\widehat{\Omega}\vdash(\widehat{e}\,\widehat{e}\,')\to\omega_i\colon\tau_i,\,\Omega'}\,[\mathit{PrjApp}]$$

$$\frac{\begin{array}{c}\widehat{\Delta};\widehat{\Omega}\vdash\widehat{e}\to\mathsf{cls}(\Delta,\mathsf{fun}\ \{\vartheta\,|\,\kappa\}\,(x\colon\theta)\Rightarrow e),\,\Omega\\ \widehat{\Delta};\Omega\vdash\widehat{e}\,'\to\omega'\colon(\forall\vartheta'\colon\kappa'.\,\theta'),\,\Omega'\\ \Delta\wedge(\vartheta,\vartheta'\colon\kappa\wedge\kappa'\wedge\theta'\le\theta);\Omega'[x=\omega'\colon(\forall\vartheta'\colon\kappa'.\,\theta')]\vdash e\to\omega''\colon\tau'',\,\Omega''\end{array}}{\widehat{\Delta};\widehat{\Omega}\vdash(\widehat{e}\,\widehat{e}\,')\to\omega''\colon\tau'',\,\Omega''}\,[\mathit{FunApp}]$$

$$\frac{\begin{array}{c}\widehat{\Delta};\widehat{\Omega}\vdash\widehat{e}\to\mathsf{cls}(\Delta,\mathsf{meth}\ \{\vartheta\,|\,\kappa\}\,(x\colon\theta)\colon\theta''\Rightarrow[\pi_1\Rightarrow e_1;\ldots;\pi_n\Rightarrow e_n]),\,\Omega\\ \delta=\exists\vartheta\colon\kappa.\,\theta\qquad\pi_i=\exists\vartheta_i.\,\theta_i\qquad\widehat{\Delta};\Omega\vdash\widehat{e}\,'\to\omega'\colon\tau',\,\Omega'\\ \tau'=\forall\vartheta'\colon\kappa'.\,\theta'\qquad j=\mathit{disp}_{\Delta}(\tau';\delta[\Delta];\pi_1,\ldots,\pi_n)\\ \Delta\wedge(\vartheta,\vartheta',\vartheta_j\colon\kappa\wedge\kappa'\wedge\theta'\le\theta,\theta_j);\Omega'[x=\omega'\colon\tau']\vdash e_j\to\omega''\colon\tau'',\,\Omega''\end{array}}{\widehat{\Delta};\widehat{\Omega}\vdash(\widehat{e}\,\widehat{e}\,')\to\omega''\colon\tau'',\,\Omega''}\,[\mathit{MethApp}]$$

Figure 6.2: Operational semantics

$(\Delta_i; \Omega)$. An evaluation is said to fail when the machine becomes ill-formed as the result of applying a rule, or when no rule can be applied. The result of an evaluation is a run-time value ω together with its closed run-time type τ. The following theorem shows that the evaluation of well-typed expressions never fails.

Theorem 50 (Subject reduction) *Let Ω be a well-formed run-time environment, Δ be a well-formed constraint context, and e be a well-typed expression with minimal type τ w.r.t. $(\Delta; \Omega)$. Then the evaluation of e w.r.t. Δ either loops forever without failure or returns a run-time value ω' with closed and well-formed run-time type τ' such that τ' is a subtype of the closure of τ w.r.t. Δ.*

Chapter 7

Algorithms

7.1 Constraint implication

We have seen in chapter 3 that constraint implication can be reduced to a series of independent implications: an implication over type variables v, and an implication over C-constructors ϕ_C for each constructor class C in the type structure $\mathcal{T}$ (cf. theorem 14). This reduction phase uses standard unification techniques to extract the "shape" of variables, so there is no need to discuss it here[1]. Instead, we focus on a simpler problem which is common to each implication, namely, implications of the form $\forall\vartheta_1.\ \kappa_1 \models \kappa_2$ where κ_1 and κ_2 are conjunctions of inequalities between variables, ϑ_1 is a non-empty set of variables, and κ_1 is ϑ_1-closed. The axiomatization of interest for this restricted form of implication uses the rules of figure 7.1 plus the transitivity rule. We assume that the substitution σ of rule VIntro_0 maps variables to variables. Obviously, a constraint is always well-formed w.r.t. this new axiomatization. Moreover, implications over C-constructors can be reduced to this problem as follows[2]. First, each type constructor $t_C \in T_C$ can be considered as a "variable" in the variable set $\vartheta^C = T_C$. Second, the partial ordering over ϑ^C can be represented by a constraint κ^C containing all the conjuncts of the form $t_C \leq t'_C$ such that $t_C \sqsubseteq_C t'_C$ holds in C. Third, C-constructor variables can be identified to regular variables. The well-formedness of a constraint κ over C-constructors is thus equivalent to $\forall\vartheta^C.\ \kappa^C \models_0 \kappa$, whereas the implication of κ_2 by κ_1 for all ϑ_1 is equivalent to $\forall\vartheta^C, \vartheta_1.\ \kappa^C \wedge \kappa_1 \models_0 \kappa_2$.

For the sake of simplicity, we assume that the left-hand side κ_1 of the implication defines a partial order $\preceq_1$ on ϑ_1. In other words, for v_1 and v'_1 in ϑ_1, we say that $v_1 \preceq_1 v'_1$ when $v_1 \leq_1 v'_1$ is implied by κ_1 for all ϑ_1, and we assume that one cannot have both $v_1 \preceq_1 v'_1$ and $v'_1 \preceq_1 v_1$ for distinct variables v_1 and v'_1 in ϑ_1. Let ϑ_2 be the set of variables of κ_2 which do not occur in ϑ_1. The implication problem can be understood as the satisfaction of constraint κ_2 w.r.t. the partially ordered universe $(\vartheta_1, \preceq_1)$. This is because the only

[1] As a matter of fact, this first phase is very similar to the MATCH algorithm of [35].

[2] For the sake of simplicity, we ignore the minimality of data type constructors. Taking minimality into account can be done easily by adding $min(v)$ conjuncts axiomatized by the following axiom, which is the analogous of rule $CMin$

$$\forall\vartheta.\ \kappa\{min(v) \wedge v' \leq v\} \models \kappa \wedge v \leq v'$$

$$
\begin{array}{lll}
[\mathit{Approx}_0] & \forall\vartheta.\ \kappa\{\kappa'\} \models_0 \kappa' & \\
[\mathit{VIntro}_0] & \forall\vartheta.\ \kappa[\sigma] \models_0 \kappa & (\sigma \in \mathcal{S}(\vartheta)) \\
[\mathit{MRef}_0] & \forall\vartheta.\ \kappa \models_0 \kappa \wedge v \leq v & \\
[\mathit{MTrans}_0] & \forall\vartheta.\ \kappa\{v \leq v' \leq v''\} \models_0 \kappa \wedge v \leq v'' &
\end{array}
$$

Figure 7.1: Restricted implication

way to introduce variables is by abstracting a "solution" deduced from the left-hand side κ_1 using rule *VIntro*$_0$. Moreover, by corollary 5 (or its equivalent for $\models_0$), we know that this implication is equivalent to $\forall\vartheta_1.\ \kappa_1 \models_0 \kappa_1 \wedge \kappa_2$. Now, if v_1 and v_1' are two variables in ϑ_1 such that $v_1 \preceq_1 v_1'$ does not hold, but $v_1 \leq v_1'$ is implied by $\kappa_1 \wedge \kappa_2$ for all ϑ_1, it is easy to see that κ_1 does not imply κ_2 for all ϑ_1. We thus assume that no absurdities between variables of the universe $(\vartheta_1, \preceq_1)$ can be derived from $\kappa_1 \wedge \kappa_2$.

Consequently, the key problem to solve in order to decide implication can be abstracted as follows: given a finite partial order $(\vartheta, \preceq)$ and a partition $\{\vartheta_1, \vartheta_2\}$ of ϑ such that ϑ_1 is non-empty, is it possible to find a total map σ from ϑ to ϑ_1, called a *solution*, such that

$$
\begin{cases}
\forall v_1 \in \vartheta_1.\ \ \sigma(v_1) = v_1 \\
\forall v \in \vartheta. \forall v' \in \vartheta.\ \ v \preceq v' \implies \sigma(v) \preceq \sigma(v')
\end{cases}
$$

For instance, the problem of deciding the following implication

$$
\forall\alpha, \beta.\ \alpha \leq \alpha \wedge \beta \leq \beta \models_0 \alpha \leq \gamma \wedge \beta \leq \gamma
$$

amounts to trying to find a valuation $\sigma(\gamma) \in \{\alpha, \beta\}$ for γ such that $\alpha \preceq \sigma(\gamma)$ and $\beta \preceq \sigma(\gamma)$ where $\preceq$ is the partial order defined by $\alpha \preceq \alpha$, $\beta \preceq \beta$, $\gamma \preceq \gamma$, $\alpha \preceq \gamma$, and $\beta \preceq \gamma$, which is clearly impossible.

The most obvious solution to this problem, which is called PO-SAT and is (implicitly) shown to be NP-complete in [45], is of course to try all possible maps σ from ϑ to ϑ_1, but this algorithm is far from efficient, and we would rather find an algorithm which is polynomial "most of the time". To this end, we define the functional Φ from the finite complete lattice[3] $\vartheta \to \mathcal{P}(\vartheta_1)$ into itself as follows

$$
\begin{array}{lll}
\forall v_1 \in \vartheta_1, \ \ \Phi(F)(v_1) & = & \{v_1\} \\
\forall v_2 \in \vartheta_2, \ \ \Phi(F)(v_2) & = & (\bigcap_{v \preceq v_2} \bigcup_{v_1 \in F(v)} \uparrow v_1) \cap (\bigcap_{v_2 \preceq v} \bigcup_{v_1 \in F(v)} \downarrow v_1)
\end{array}
$$

where we define $\uparrow v_1$ (resp. $\downarrow v_1$) as the set of all $v_1' \in \vartheta_1$ such that $v_1 \preceq v_1'$ (resp. $v_1' \preceq v_1$).

We say that F is a *pre-solution* if F is a pre-fixpoint of Φ (that is, $F \subseteq \Phi(F)$) and $F(v)$ is non-empty for every $v \in \vartheta$. Since, Φ is monotonic, Φ always has a greatest fixpoint which is above all its pre-fixpoints, and this greatest fixpoint can be computed in polynomial time, since $\Phi(F)$ can be computed in polynomial time for every F, and the

[3] The ordering $\subseteq$ on maps is defined by $F \subseteq F'$ if and only if $F(v) \subseteq F'(v)$ for every $v \in \vartheta$.

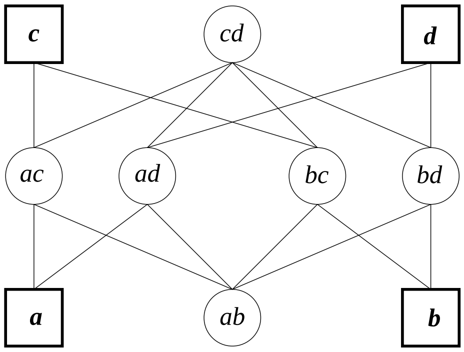

Figure 7.2: Counter-example for the greatest fixpoint algorithm

height of the lattice $\vartheta \to \mathcal{P}(\vartheta_1)$ is quadratic in the number of variables. The existence of a pre-solution is thus a polynomial problem, and the set of pre-solutions is either empty or has a maximum solution, called the maximum pre-solution.

Theorem 51 *For every solution σ, the function F defined by $F(v) = \{\sigma(v)\}$ for every variable $v \in \vartheta$ is a pre-solution.*

Consequently, the existence of a pre-solution is a prerequisite to the existence of a solution[4], and if G is the maximum pre-solution, then every solution σ of the problem is such that $\sigma(v) \in G(v)$ for every $v \in \vartheta$. Unfortunately, as implied by the complexity result of [45], the existence of a pre-solution is not sufficient to prove the existence of a solution. The simplest counter-example that we have found is given figure 7.2, with $\vartheta_1 = \{a, b, c, d\}$ and $\vartheta_2 = \{ac, ad, bc, bd, cd, ab\}$. This problem (which is built by simplifying the set of convex subsets of $\{a, b, c, d\}$ partially ordered by the Plotkin ordering on powerdomains) does not have a solution, but it can be shown that

$$\begin{array}{llllllll} G(ac) & = & \{a,c\} & G(bc) & = & \{b,c\} & G(ad) & = & \{a,d\} \\ G(bd) & = & \{b,d\} & G(ab) & = & \{a,b\} & G(cd) & = & \{c,d\} \end{array}$$

However, even if the existence of a pre-solution is not enough to decide the existence of a solution, the fact that $G(v)$ contains $\sigma(v)$ for every solution σ is a step in the right direction. As a matter of fact, if $|G(v_2)| = 1$ for every $v_2 \in \vartheta_2$, then the following theorem shows that G defines a solution.

Theorem 52 *If the maximum pre-solution G is such that $|G(v_2)| = 1$ for every $v_2 \in \vartheta_2$, then the problem has a unique solution σ.*

Now, if $|G(v_2)| > 1$ for some $v_2 \in \vartheta_2$, then a possible algorithm is to try to arbitrarily fix $G(v_2) = \{v_1\}$ for some $v_1 \in G(v_2)$, to reapply the greatest fixpoint algorithm starting from G instead of the maximum element of the lattice, backtracking if necessary. This

[4]This property, as well as the functional Φ, are already given in [22], but the authors conjecture incorrectly that the existence of a pre-solution is sufficient to ensure that the problem has a solution.

$$
\begin{array}{l}
\textsf{procedure } \mathit{Solve}(F) \textsf{ is} \\
\quad G \;=\; \bigcap_{i \geq 0} \Phi^i(F); \\
\quad \textsf{if } \exists v_2 \in \vartheta_2.\ |G(v_2)| = 0 \textsf{ then } \mathit{fail}; \\
\quad \textsf{if } \forall v_2 \in \vartheta_2.\ |G(v_2)| = 1 \textsf{ then } \mathit{succeed}; \\
\quad \textsf{for } v_2 \in \vartheta_2 \textit{ such that } |G(v_2)| > 1 \textsf{ do} \\
\quad\quad \textsf{for } v_1 \in G(v_2) \textsf{ do} \\
\quad\quad\quad \mathit{Solve}(G[v_2 \mapsto \{v_1\}]) \\
\textsf{end } \mathit{Solve}
\end{array}
$$

Figure 7.3: Algorithm to decide restricted implication

intuition is formalized by the algorithm of figure 7.3, where function *Solve* is initially called with the maximum element of the lattice, that is to say, the function F such that $F(v) = \vartheta$ for every $v \in \vartheta$. We expect this algorithm, which is potentially exponential, to be polynomial in practice. Note that efficient chaotic iteration strategies [9] can be used to compute greatest fixpoints efficiently. Our current implementation uses this algorithm, and we have never encountered an exponential blow-up on the examples we have tried, but remark that the same problem is true for ML type inference algorithm (it can be exponential, but is polynomial in practice).

7.2 Type inference

We believe that all the techniques developed for type inference with subtypes [5, 21, 22, 23, 26, 29, 33, 34, 35, 41, 42] can be used to perform the type inference of untyped $\mathrm{ML}_{\leq}$ programs which do not contain methods. In particular, the UNIFY and MATCH algorithms of [23, 26, 35] can be used, as well as various constraint simplification algorithms.

Consequently, we do not give a formal specification of the type inference algorithm. Rather, we give a few examples showing that type inference for methods is somewhat more subtle. For example, assume a ()-variant class C with data type constructors a and b, and type constructor c such that $a \sqsubseteq_C c$ and $b \sqsubseteq_C c$, and assume that method m is defined as follows

$$
\begin{array}{l}
m(x\colon a) = \mathsf{let}\ \cdots\ \mathsf{in}\ a(); \\
m(x\colon b) = \mathsf{let}\ \cdots\ \mathsf{in}\ b();
\end{array}
$$

as a private method of some module of the program (inferring the type of a method implemented in several modules does not make sense). The problem is to find a minimum type τ for m such that (1) $(\exists\,\emptyset.\ a, \exists\,\emptyset.\ b)$ is a partition of the domain δ of τ, and (2) the bodies of the two alternatives of m type-check. At first glance, one may think of giving m one of the two following types

$$
\begin{array}{rcl}
\tau_1 & = & \forall\alpha\colon \alpha \in C.\ \alpha \to \alpha \\
\tau_2 & = & \forall\alpha\colon \alpha \leq c.\ \alpha \to \alpha
\end{array}
$$

Note that the domain δ_2 of τ_2 is a strict subdomain of the domain δ_1 of τ_1, even though these two domains are intuitively equivalent in a "closed world". Also, remark that τ_i denotes the identity over δ_i, since the two alternatives of m return the same data type that the data type they take as argument. Now, an important subproblem of our original problem is to find a *maximum* domain δ such that $(\exists\,\emptyset.\ a, \exists\,\emptyset.\ b)$ is a partition of δ and the body of the alternatives type-check w.r.t. δ. This is the hard part, since each alternative is type-checked w.r.t. a different constraint context (cf. rule *Meth*), so that the "constraints" on δ inferred during the type-checking of each alternative are hard to "merge" automatically in a systematic and optimal way, and one has to "guess" this domain. Intuitively, if δ_i is the domain inferred for the i-th alternative, then δ should be the greatest lower bound of all the δ'_i, where δ'_i is just δ_i with the additional constraints imposed by rule *Meth* "removed", which is hard to define formally. The two domains δ_1 and δ_2 are both reasonable candidates for δ, and it is not clear which one to choose. Remark that this ambiguity does not occur in ML, since the unification algorithm would identify δ_1 and δ_2 to $\exists\,\emptyset.\ c$, assuming c is declared as follows

$$\mathsf{datatype}\ c = a\ \mathsf{of\ unit} \mid b\ \mathsf{of\ unit}$$

Worse, if m has a third "catch-all" alternative defined as follows

$$m(x\colon \_) = x;$$

then the minimum type of m is $\forall\alpha.\ \alpha \to \alpha$, which is much more general than the ML type $c \to c$ inferred by the unification algorithm. Note that this type would also have been inferred by an ML compiler without the third "catch-all" alternative, in which case ML compilers normally print a "match not exhaustive" warning.

Now, the rationale of choosing δ_1 is that it is the largest domain which contains a and b, and nothing else (w.r.t. to the current type structure). However, this domain may be too large to type-check the bodies of the alternatives of m, so we may want to choose δ_2 which is the smallest[5] domain containing a and b. Doing so guarantees that if one of the alternatives of m fails to type-check, and m is not recursive, then the method cannot type-check. However, doing so also restricts the possible uses of m elsewhere, e.g., if m is let-bound. In a sense, δ_1 and δ_2 are the upper and lower bounds for the choice of δ. In terms of abstract interpretation, the only solution to this problem is probably to design a widening operator [8, 15] on domains (or the dual of a widening operator, to be precise), and use efficient chaotic iteration strategies [9] to compute fixpoints. As a matter of fact, the unification algorithm of ML can be seen as a widening operator [37, 38]. We leave open the design of this widening operator for $\mathrm{ML}_{\leq}$. One possible track may be to use δ_1 and allow non-exhaustive matches, as in ML, but more sophisticated widening operators can certainly be designed.

[5]Informally speaking, of course, since domains do not form a sup-semi-lattice without disjunctive constraints. That is to say, the smallest domain containing a and b is $(\exists\alpha\colon \alpha \leq a \vee \alpha \leq b.\ \alpha)$, which cannot be expressed in $\mathrm{ML}_{\leq}$. The domain δ_2 is thus the most natural approximation of the least upper bound of a and b in a closed world.

Chapter 8

Extensions

8.1 Modules

In this section, we show how to define a module system for $\mathrm{ML}_{\leq}$ preserving some modularity in type-checking. Our goal is not to give a formal definition of the module system. Rather, we define the system informally and give hints on some key points of the design.

We propose to partition the program into *modules* and *interfaces*. Modules and interfaces are named, and we assume that each module/interface has a unique name in the same namespace, so that we confuse modules and interfaces with their name. We assume the existence of a module, called $Main$, defining a function $main$ with type $\mathsf{unit} \rightarrow \mathsf{unit}$. As opposed to chapter 5, where the name of functions, methods, and variables consisted in strings of alphanumeric characters, we assume that the name of every global object in the program is explicitly qualified by the name of a module or the name of an interface, except for the declarations of classes, types, data types, and the specification of functions and methods, which use unqualified names for the object they define, this name being implicitly qualified by the module or the interface in which the declaration occurs. Note that the examples of chapter 1 used a more liberal convention with an implicit qualification of the names in scope.

Interfaces are sequences of import statements, class, type, and data declarations and implementations in the style of figure 5.1, as well as specifications for methods and functions[1]. Import statement have the form $\mathsf{import}\ I$, where I is the name of an interface, and specify the list of interfaces that the interface can use. The specification of a function f in interface I is of the form $\mathsf{fun}\ f\colon \tau$. The specification of a method m in interface I is of the form $\mathsf{meth}\ m\colon \tau$, where τ is of the form $\forall\vartheta\colon \kappa.\ \theta_1 \rightarrow \theta_2$. The class, type, and data declarations of an interface I are valid if, as a whole, they form an admissible extension of the type structure imported by I. Every data type constructor d_C, declared in some interface I, must have exactly one implementation of the following form (in I or in some other interface importing I)

$$\mathsf{data}\ I.d_C[\vartheta_C]\ \mathsf{is}\ \cdots\ \mathsf{end}$$

[1]Note that values that are not abstractions cannot be defined at toplevel in modules or exported in interfaces. This restriction could be lifted with a lazy language allowing the recursive definition of arbitrary values.

Modules are sequences of import statements and letrec-bindings for methods and functions[2]. Import statements specify the list of interfaces that the module can use. A function f defined in interface I must have exactly one definition of the form

$$\mathsf{fun}\ I.f\ \{\vartheta \mid \kappa\}\ (x\colon \theta) = e$$

in some module M importing I. A method m defined in interface I can be implemented in several modules M_i, $i \in [1, n]$, importing I. The i-th implementation of method m in module M_i is of the form

$$\mathsf{meth}\ I.m(x\colon \pi_i) = e_i$$

where the type variables ϑ of the specification of the method are accessible in the scope of the body e_i.

The meaning of a collection of modules and interfaces is a program (in the sense of section 5.1) with a type structure built by gathering all the declarations in the interfaces of the program, together with a recursive let-expression[3] whose body is the expression *Main.main()*, and whose bindings are the specifications of all the interfaces and the letrec-bindings of all the modules of the program. The definition of function f is transformed into the following letrec-binding

$$I.f\colon \tau = \mathsf{fun}\ \{\vartheta \mid \kappa\}\ (x\colon \theta) \Rightarrow e$$

whereas the n implementations of method m are gathered as follows

$$I.m\colon \tau = \mathsf{meth}\ \{\vartheta \mid \kappa\}\ (x\colon \theta_1)\colon \theta_2 \Rightarrow [\pi_1 \Rightarrow e_1; \ldots; \pi_n \Rightarrow e_n]$$

In order to have a modular type-checking, we must first define the meaning of a module M independently of the rest of the program. We propose to consider that a module denotes the "partial" program formed by the interfaces it imports, together with its own body[4]. Note that the body of each alternative of a method can be type-checked independently from its other alternatives, so that it makes sense to type-check a partial method, provided of course that we relax the condition imposed by rule *Meth* that the patterns of a method form a partition of its domain, and report this check to link-time when all the modules are known. The key point of the system is that the global link-time type structure is an admissible extension of the imported type structure in which the type-checking of M is performed. Moreover, the minimal type of each expression in M is also the minimum type w.r.t. the link-time type structure, since the minimal derivation w.r.t. the type type structure of M is also the minimal derivation w.r.t. the link-time type structure. Consequently, modules can be type-checked in a modular fashion with the provision that methods must be checked for completeness and non-ambiguity at link-time.

[2]It would be easy to add class, type and data type definitions too, as in chapter 1.

[3]A hierarchy could also be defined on modules and interfaces to avoid the recursive definition of every toplevel object and allow values to be exported in interfaces.

[4]Note that in order to do so, we need to allow functions and methods to be declared, but not implemented, since a function declared in an interface I imported by M is not necessarily implemented in M.

Note that, as always, the use of multi-methods (see below) or the use of complex hierarchies of type constructors can lead to link-time failures. There as several ways to minimize the problem. For instance, we could restrict ourselves to tree hierarchies (that is, forbid multiple-inheritance). Or, as advocated in [14], we could impose that interfaces which extend the classes defined in some interface I explicitly say that they extend I, and use the notion of *resolving module* to resolve ambiguities raised by the use of multi-methods. In any case, these language design issues must be solved in a satisfactory way before multi-methods and multiple-inheritance can be accepted as a viable alternative to single-dispatch languages, and we shall discuss this problem at length in another report.

8.2 Multi-methods

Section 5.4 showed that defining curried methods with constrained types can be somewhat subtle and counter-intuitive. We thus need a direct way to define methods with several arguments. We are immediately confronted with a choice. We can either extend the theory to account for multi-methods, or take the view that multi-methods are just syntactic sugar for curried methods. We study the two views in sequence, and see that built-in multi-methods are more expressive than curried methods. For the sake of simplicity, we restrict ourselves to methods with two arguments, and we assume the existence of a $(\oplus, \oplus)$-variant class Pair with a single data type constructor pair implemented as follows

$$\mathsf{data\ pair}[\alpha, \beta]\ \mathsf{is}\ 1\colon \alpha;\ \ 2\colon \beta\ \mathsf{end}$$

where, as usual, the monotype $\mathsf{pair}[\theta_1, \theta_2]$ is written $\langle\theta_1, \theta_2\rangle$. So let m be the following multi-method

$$m = \mathsf{meth}\ \{\vartheta \mid \kappa\}\ (x\colon \langle\theta_1, \theta_2\rangle)\colon \theta \Rightarrow [\pi_1 \Rightarrow e_1; \dots \pi_n \Rightarrow e_n]$$

where for each $i \in [1, n]$, π_i is a *multi-pattern* of the form $\exists\vartheta_{i,1}, \vartheta_{i,2}.\ \langle\theta_{i,1}, \theta_{i,2}\rangle$ such that $\vartheta_{i,1}$ is disjoint from $\vartheta_{i,2}$ and $\pi_{i,j} = \exists\vartheta_{i,j}.\ \theta_{i,j}$ is a well-formed pattern for $j \in [1, 2]$. The restriction that $\vartheta_{i,1}$ and $\vartheta_{i,2}$ be disjoint corresponds to the fact that the two components of a multi-pattern must be independent, so that we can confuse π_i with $\langle\pi_{i,1}, \pi_{i,2}\rangle$. In other words, multi-patterns must be *linear*. For instance, $\exists\alpha.\ \langle\alpha, \alpha\rangle$ is not a multi-pattern. Extending the theory to allow multi-methods like m simply consists in adding multi-patterns and replacing condition (3) of definition 44 by the following condition

(3) *for every data type constructors d_{C_1} and d_{C_2} in $\mathcal{T}$ such that $\delta[\Delta]$ and the multi-pattern π defined as $\exists\vartheta_{C_1}, \vartheta_{C_2}.\ \langle d_{C_1}[\vartheta_{C_1}], d_{C_2}[\vartheta_{C_2}]\rangle$ are compatible, the set $\{\pi_i \mid \exists i \in [1, n].\ \pi \leq \pi_i\}$ has a minimum element*

which generalizes easily to multi-patterns with more than two components. The key point of this generalization is that theorems 43 and 45 remain valid[5], provided we define run-time types as closed types of the form $\forall\vartheta\colon \kappa.\ d_C[\Theta_C]$ when d_C is not the pair constructor, and closed types of the form $\forall\vartheta\colon \kappa.\ \langle d_{C_1}[\Theta_{C_1}], d_{C_2}[\Theta_{C_2}]\rangle$. It is then easy to check that

[5]Note that the fact that these theorems remain valid relies heavily on the hypothesis that every multi-pattern $\exists\vartheta_1, \vartheta_2.\ \langle\theta_1, \theta_2\rangle$ is such that θ_1 is ϑ_1-closed, θ_2 is ϑ_2-closed, and ϑ_1 is disjoint from ϑ_2.

the rest of the theory is unchanged, and that, in particular, minimal typing and subject reduction remain valid.

Another approach to the introduction of multi-methods like m is to find a way to write a curried version of m while preserving the "most specific" semantics for dynamic dispatch. Let $\widehat{\pi}_1, \ldots, \widehat{\pi}_m$ denote the elements of the set $\{\pi_{i,1} \mid i \in [1,n]\}$ (where, as usual, we identify equivalent patterns). To the light of section 5.4, we propose to write the curried form m' of m as follows

$$\begin{array}{l} m' = \mathsf{meth}\ \ \{\vartheta \mid \kappa\}\,(x_1 \colon \theta_1) \colon \theta_2 \to \theta \Rightarrow [\,\cdots \\ \quad \widehat{\pi}_i \Rightarrow \mathsf{meth}\ \ \{\vartheta' \mid \kappa' \wedge \theta_1 \le \theta'_1\}\,(x_2 \colon \theta'_2) \colon \theta' \Rightarrow [\,\cdots \\ \qquad \pi_{j,2} \Rightarrow \mathsf{let}\ x = \langle x_1, x_2 \rangle\ \mathsf{in}\ e_j\ \mathsf{end}; \\ \quad \cdots\,]; \\ \cdots\,] \end{array}$$

where $\forall\vartheta' \colon \kappa'.\ \langle\theta'_1, \theta'_2\rangle \to \theta'$ is a fresh renaming of the type of m, and for every $i \in [1,m]$, every j is the index of a pattern π_j whose first projection is equivalent to $\widehat{\pi}_i$. We say that m is well-typed w.r.t. Δ if this rewriting is well-typed w.r.t. Δ (giving a direct reformulation is too complex and not very intuitive). The problem with this implementation of multi-methods as curried methods is that some well-typed multi-methods do not have well-typed translations. To see why, just assume the existence of a ()-variant class A with type constructors a_1, a_2, and a_3, and data type constructor a_4 such that $a_4 \sqsubseteq_A a_2 \sqsubseteq_A a_1$ and $a_4 \sqsubseteq_A a_3 \sqsubseteq_A a_1$, as well as the existence of a ()-variant class B with type constructor b_1 and data type constructors b_2 and b_3 such that $b_2 \sqsubseteq_B b_1$ and $b_3 \sqsubseteq_B b_1$. Then the following multi-method (left) is well-typed, whereas its translation (right) is ill-typed

$$\begin{array}{ll} m_1 = \mathsf{meth}\ (x \colon \langle a_1, b_1 \rangle) \colon a_1 \Rightarrow [& m'_1 = \mathsf{meth}\ (x_1 \colon a_1) \colon b_1 \to a_1 \Rightarrow [\\ \quad \langle a_2, b_2 \rangle \Rightarrow \mathsf{pair.1}\ x; & \quad a_2 \Rightarrow \mathsf{meth}\ (x_2 \colon b_1) \colon a_4 \Rightarrow [b_2 \Rightarrow x_1]; \\ \quad \langle a_3, b_3 \rangle \Rightarrow \mathsf{pair.1}\ x; & \quad a_3 \Rightarrow \mathsf{meth}\ (x_2 \colon b_1) \colon a_4 \Rightarrow [b_3 \Rightarrow x_1] \\ ] &] \end{array}$$

since data type a_4 matches both a_2 and a_3, which are not comparable, so that a_2, a_3 is not a partition of a_1. Another interesting counter-example is the following polymorphic multi-method (left) and its translation (right)

$$\begin{array}{ll} m_2 = \mathsf{meth}\ \{\alpha\}\,(x \colon \langle \alpha, \alpha \rangle) \colon \alpha \Rightarrow [& m'_2 = \mathsf{meth}\ \{\alpha_1\}\,(x_1 \colon \alpha_1) \colon \alpha_1 \to \alpha_1 \Rightarrow [\\ \quad \langle \_, \_ \rangle \Rightarrow \mathsf{pair.1}\ x; & \quad \_ \Rightarrow \mathsf{meth}\ \{\alpha_2 \mid \alpha_1 \le \alpha_2\}\,(x_2 \colon \alpha_2) \colon \alpha_2 \Rightarrow [\\ \quad \langle \_, \mathsf{int} \rangle \Rightarrow \mathsf{pair.1}\ x; & \qquad \_ \Rightarrow x_1;\ \ \mathsf{int} \Rightarrow x_1;\ \ \mathsf{bool} \Rightarrow x_1 \\ \quad \langle \_, \mathsf{bool} \rangle \Rightarrow \mathsf{pair.1}\ x & \quad] \\ ] &] \end{array}$$

It is easy to see that m_2 is well-typed and that m'_2 is ill-typed, since the two inner alternatives have patterns which are incompatible with the domain $\exists\alpha_2 \colon \alpha_1 \le \alpha_2.\ \alpha_2$ w.r.t the context $(\alpha_1 \colon \mathit{true})$. This is because the inner method has no information whatsoever about the type α_1 of the first argument, so that they cannot dispatch on the second argument which supposedly has the same type. In contrast, m_2 dispatches on both arguments *simultaneously.* These examples thus show that built-in multi-methods are more expressive than curried multi-methods. However, we believe that both schemes should be equivalent in practice, since counter-examples are rather ad-hoc.

8.3 Abstract, concrete, and template classes

As we have already noted in the introduction, base types, that is, types built from type constructors with arity 0, are the equivalent of *abstract classes* in that no instance of such types can be instanciated at run-time. On the other hand, base data types correspond to *concrete classes* since they have implementations which can be instanciated at run-time. For classes with arity greater than 0, type constructors correspond to *abstract template classes* and data type constructors correspond to *concrete template classes.*

Note that template classes, as in $C{+}{+}$ for instance, are usually non-variant, that is, a $C{+}{+}$ class $C\langle T \rangle$ is a subclass of $C'\langle T' \rangle$ if $T = T'$ and C is a subclass of C'. This is because $C{+}{+}$ is an imperative language with side-effects, so that T can be used as the type of a mutable field, which makes in non-variant. Indeed, assume that a function f expects an argument r which is a reference to a real (that is, a pointer to a real). This means that this function assumes that the contents of r is at most a real, but also that any real can be stored into r. Now, if the type "reference to integer" was a subtype of "reference to real", then a reference r' to an integer could be safely passed as argument to f, so that f could store a real into r', which is unsound.

8.4 Side-effects and references

As in ML, state can be added to $\mathrm{ML}_{\leq}$ by adding a $(\otimes)$-variant type constructor ref, together with two functions *set* and *get* to set the value and retrieve the value of references. However, it is very well-known that when used in conjunction with predicative type systems like ML or $\mathrm{ML}_{\leq}$, references cannot be polymorphic. Consequently, if we were to add monomorphic references to $\mathrm{ML}_{\leq}$, we should add a new variance specifier $\odot$ denoting both non-variance and monomorphism, and declare the ref constructor as a member of the $(\odot)$-variant class Ref. For example, the class IList of imperative lists, that is, lists with mutable elements, could be declared as a $(\odot)$-variant class and the imperative cons would be implemented as follows, in pretty much the same way as it would be implemented in $C{+}{+}$

$$\mathsf{data\ icons}[\alpha]\ \mathsf{is}\ 1\!:\ \mathsf{ref}[\alpha];\ 2\!:\ \mathsf{ilist}[\alpha]\ \mathsf{end}$$

8.5 Nonvirtual, virtual, and purely virtual methods

$\mathrm{ML}_{\leq}$ functions are similar to the *nonvirtual member functions* of $C{+}{+}$, or to the *frozen routines* of Eiffel, that is, operations with a default behavior that cannot be overridden. $\mathrm{ML}_{\leq}$ methods are similar to the *virtual member functions* of $C{+}{+}$, or to the *effective routines* of Eiffel, that is, operations with a default behavior that can be overridden.

Finally, $ML_{\leq}$ methods with no default behavior (that is, methods with no "catch-all" alternative with pattern $\exists\alpha.\ \alpha$) are similar to the *purely virtual member functions* of $C{+}{+}$, to the *deferred routines* of Eiffel, and to the *abstract methods* of Smalltalk.

We could imagine requiring the explicit declaration of purely virtual methods in interfaces to impose the implementation of these methods over every type in their domain. Other annotations could also be created to give better hints to the programmer about how exported methods should be refined when new types are created.

8.6 The dot notation

The *dot notation* used in single-dispatch languages to invoke methods can also be adapted to $ML_{\leq}$. Indeed, a natural use of the module system of section 8.1 is to associate every constructor class C to an interface I exporting the public interface of C and a module M implementing the public interface of C, as well as the private functions and methods of C. In general, C and I will have the same name, as for the example of figure 1.1.

To recover the kind of encapsulation enjoyed by class-based languages such as $C{+}{+}$, one possibility is to define the notation $a.b$ as syntactic sugar for $(I.b\ a)$, where I is the interface in which the class of a is defined, where the class of a is the unique class C such that the minimal type of a is of the following form $\forall\vartheta : \kappa.\ t_C[\Theta_C]$ (if the type of a is not of this form, e.g., if a has type $\forall\alpha.\ \alpha$, then it is a compile-time error).

This scheme will work very well if all the methods of the class are defined in the same interface. If not, this scheme can be refined to match the convention of standard single-dispatch languages. Note that our scheme is very flexible and avoids the need for a separate mechanism to manage *name spaces*, as in $C{+}{+}$ for instance.

8.7 Type classes

Our notion of type constructor class is quite unusual in object-oriented languages. However, this notion is important in $ML_{\leq}$, since $ML_{\leq}$ supports multiple-inheritance and does not have a notion of maximum type or root class. However, if hierarchies of type constructors were restricted to trees, the root of every tree could be identified with the class associated to it, and constructor classes could be eliminated. Nonetheless, note that constructor classes can be useful to allow multiple-inheritance in a much broader sense than what was used in this report. Indeed, we have always defined classes like Unit, Num, Bool with the same variance and different "meanings", but we could just as well have defined one single ()-variant class Base with type and data type constructors unit, real, true, etc. Doing so would allow the definition of "overloaded" functions with types like $\forall\alpha : \alpha \in \mathsf{Base}.\ \alpha \to \alpha$. A straightforward generalization of $ML_{\leq}$ would thus be to define a partial order between constructor classes, and to add constructor class variables. However, a more interesting generalization might be to add *type classes* [28, 39, 44], that is, user-defined predicates P on monotypes. Since $ML_{\leq}$ is based on constraint implication, adding type classes simply amounts to defining new conjuncts of the form $P(\theta)$ and giving a stronger, but still decidable, axiomatization of the implication while maintaining the good properties of the system, namely, minimal typing and subject reduction.

```
// Predicates
pred String;
pred Print;

// Axiomatization
rule String(Num);
rule Print(Bool);

rule Print(α) if String(α);
rule String(List[α]) iff String(α);

meth string : ∀α: String(α). α → string;
meth string(x: _) = "*";
meth string(x: nil) = "";
meth string(x: cons) =
   concat (string (cons.1 x)) (string (cons.2 x));

meth print : ∀α: Print(α). α → unit;
meth print(x: String) = write (string x);
meth print(x: true) = write "true";
meth print(x: false) = write "false";
```

Figure 8.1: Type classes

For instance, assume that Print is a predicate denoting the set of monotypes corresponding to printable objects, and String is the set of monotypes denoting objects which can be converted to strings (and thus be printed). We may want to define this predicate inductively as in figure 8.1. The first two rules assert that every number can be converted to a string and that booleans can be printed. This user-defined rules are taken into account by the type system through the addition of the following axioms to figure 3.1

$$\forall\vartheta.\ \kappa \models \kappa \wedge \mathsf{String}(t_{\mathsf{Num}}) \qquad\qquad \forall\vartheta.\ \kappa \models \kappa \wedge \mathsf{Print}(t_{\mathsf{Bool}})$$

The third rule, which states that every monotype convertible to a string can also be printed, translates to the following axiom

$$\forall\vartheta.\ \kappa\{\mathsf{String}(\theta)\} \models \kappa \wedge \mathsf{Print}(\theta)$$

The last rule states that a list can be converted to a string if and only if its elements are convertible to strings, which is reflected by the following axioms

$$\forall\vartheta.\ \kappa\{\mathsf{String}(\theta)\} \models \kappa \wedge \mathsf{String}(t_{\mathsf{List}}[\theta])$$
$$\forall\vartheta.\ \kappa\{\mathsf{String}(\phi_{\mathsf{List}}[\theta])\} \models \kappa \wedge \mathsf{String}(\theta)$$

used both the "construct" monotypes and to "destruct" them, in pretty much the same way as rules *MIntro* and *VElim* do. Obviously, these rules forbid any other rule of the form

```
rule String(List[Bool]);
```

asserting that lists of booleans can be converted to strings without asserting that booleans can be converted to strings. This example shows that proper restrictions must be put on user-defined rules for the system to be sound and tractable. Finally, assuming that every type class is non-variant[6], the following rule holds for every type class P

$$\forall\vartheta.\ \kappa\{P(\theta) \wedge \theta = \theta'\} \models \kappa \wedge P(\theta')$$

[6]This is simpler, but not necessary. Type classes could have explicitely declared variance as for constructor classes.

With these definitions, one may for instance define the following program (provided, of course, that a meaning can be assigned to it and that subject reduction holds, which we do not claim here).

8.8 Implementation inheritance

Finally, we want to mention how implementation inheritance could be added to $\mathrm{ML}_{\leq}$ by sugaring the syntax of the language. As noted by many authors, the best way to define implementation inheritance is to define an inheritance hierarchy which is distinct from the subtyping hierarchy. However, for the sake of simplicity, we assume a one-to-one mapping between the subtype relation and the inheritance relation, which is what is done in a lot of object-oriented languages, e.g. C++. The idea is then to take the convention that every data type declaration of the form

$$\mathsf{data}\ d_C[\vartheta_C]\ \mathsf{is}\ \cdots\ f : \theta\ \cdots\ \mathsf{end}$$

declares both a *type constructor* $\$d_C$ and a *data type constructor* d_C with the fields listed in its declaration plus the fields of all its superconstructors (assuming, of course, that the fields are named instead of being numbered). For every field f, this declaration would automatically declare a *virtual* extractor f implemented as a method dispatching on all the data type constructors d'_C, d''_C, etc., below $\$d_C$, that is to say

$$\begin{array}{l}
\mathsf{meth}\ \ f : \forall\vartheta_C.\ \$d_C[\vartheta_C] \to \theta;\\
\mathsf{meth}\ f\,(x : d_C) = d_C.f\ x;\\
\mathsf{meth}\ f\,(x : d'_C) = d'_C.f\ x;\\
\mathsf{meth}\ f\,(x : d''_C) = d''_C.f\ x;\\
\qquad\quad\vdots
\end{array}$$

Note that our model of implementation inheritance is not based on extensible records, since the field f of two distinct data type constructors below d_C are semantically unrelated.

Chapter 9

Related work and conclusion

9.1 Related work

The object-oriented model developed in this report is very similar to that of the programming languages CLOS [17] and Cecil [14], in particular by its distinction between concrete and abstract classes, the distinction between subtyping and implementation inheritance, the use of method specifications, and the use of modules to provide encapsulation. However, the type system proposed by Chambers and Leavens is only first-order and monomorphic, and the specification of methods by means of sets of monomorphic signatures is less expressive and more "ad-hoc" than ours. Nonetheless, many of the techniques developed in [14] could be adapted to our system, in particular techniques for true separate compilation of multi-methods and compilation of dynamic dispatch.

Another system which has interesting links with $\mathrm{ML}_{\leq}$ is of course $\mathrm{F}_{\leq}$ [16]. Both systems have a subsumption rule, a notion of type application, and above all, minimal typing. However, $\mathrm{F}_{\leq}$ deals with explicit polymorphism (that is, types are passed as arguments to functions but do not otherwise interfere with the execution of the program), whereas $\mathrm{ML}_{\leq}$ deals with implicit polymorphism (that is, types are not passed as arguments to functions), and the dynamic type of objects is used to perform dynamic dispatch. Moreover, $\mathrm{F}_{\leq}$ allows polymorphic formal arguments, whereas $\mathrm{ML}_{\leq}$, as every predicative type system, does not. These differences make the two systems rather hard to compare. However, one major difference between the two systems is that $\mathrm{F}_{\leq}$ is undecidable, whereas $\mathrm{ML}_{\leq}$ is decidable, and we have seen in section 3.2 that subtyping can probably be decided in polynomial time in practice. Castagna and Pierce have tried in [11] to design a decidable variant of $\mathrm{F}_{\leq}$, but they have acknowledged in [12] that this variant does not have minimal typing. Nonetheless, the comparison of our subtyping rule to the three variants $\forall$-orig, $\forall$-top and $\forall$-Fun of the subtyping rule for universally quantified types given in [12] is interesting. Let U_1 and U_2 be two F-bounded types of the form $\forall(X \leq T_i)\, S_i$ such that T_i and S_i are monomorphic (i.e., do not contain quantifiers). We naturally identify U_i with the polymorphic constrained type $\tau_i = \forall X : X \leq T_i.\ S_i$. The original subtyping rule of [16] states that U_1 is a subtype of U_2 w.r.t. some environment Γ if and only if

$$\Gamma \vdash T_2 \leq T_1 \qquad \Gamma, X \leq T_2 \vdash S_1 \leq S_2$$

that it to say, identifying Γ with a context Δ of the form $(\vartheta: \kappa)$,

$$\forall \vartheta.\, \kappa \models T_2 \leq T_1 \qquad \forall \vartheta, X.\, \kappa \wedge X \leq T_2 \models S_1 \leq S_2$$

or, by the fact that X does not occur free in T_1 and T_2 and corollary 4

$$\forall \vartheta, X.\, \kappa \wedge X \leq T_2 \models T_2 \leq T_1 \wedge S_1 \leq S_2$$

and by corollary 5

$$\forall \vartheta, X.\, \kappa \wedge X \leq T_2 \models X \leq T_2 \wedge T_2 \leq T_1 \wedge S_1 \leq S_2$$

so that by transitivity

$$\forall \vartheta, X.\, \kappa \wedge X \leq T_2 \models X \leq T_1 \wedge S_1 \leq S_2$$

which, by rule *VIntro*, implies that τ_1 is a subtype of τ_2 w.r.t. Δ. The original subtyping rule of $F_{\leq}$ is thus sound w.r.t. ours, and so are the two others. Moreover, our type application operator app_{Δ} is monotonic, whereas this is not so at the term level for $F_{\leq}$, where type application is just syntactic substitution.

Another difference between $F_{\leq}$ and $ML_{\leq}$ is that $F_{\leq}$ lacks both least upper bounds and greatest lower bounds, whereas any two types with an upper bound have a least upper bound in $ML_{\leq}$. Finally, another important difference between $F_{\leq}$ and $ML_{\leq}$ is that $F_{\leq}$ is not based on a user-defined type hierarchy. This makes the system simpler and more self-contained, but we think that it looses the fact that class hierarchies are at the heart of object-orientation, since they somehow specify the objects manipulated by the program. Moreover, the openness of these hierarchies is at the heart of $ML_{\leq}$ thanks to theorem 22 which shows that our axiomatization of the implication, in a sense, captures all true properties w.r.t. an open world.

Castagna [10] has defined an extension of $F_{\leq}$ allowing function overloading in a higher-order setting with explicit polymorphism and primitive subtyping. His model is quite powerful but technically rather tricky, as all impredicative models. Moreover, type-checking is undecidable and methods lack specifications, which can be a problem for modularity and scalability. Nonetheless, an extension of $ML_{\leq}$ where types are collections of compatible polytypes may be worth considering to have a better typing of methods, and, maybe, to allow a precise type inference for methods. It is not clear that such an extension would remain decidable, though.

Mitchell [34] has studied containment-based type inference which has interesting connections with our system. In particular, Mitchell gives containment rules and axioms for second-order lambda-calculus which are valid in all *simple inference models.* The following rules, using the notations of [34],

(*sub*)	$\forall\, \mathbf{t}.\, \sigma \subseteq \forall\, \mathbf{r}.[\tau/\mathbf{t}]\, \sigma$	where $\mathbf{r}$ are not free in $\forall\, \mathbf{t}.\, \sigma$
(*arrow*)	$\sigma_1 \subseteq \sigma, \tau \subseteq \tau_1 \;\vdash\; \sigma \rightarrow \tau \subseteq \sigma_1 \rightarrow \tau_1$	
(*ref*)	$\sigma \subseteq \sigma$	
(*trans*)	$\rho \subseteq \sigma, \sigma \subseteq \tau \;\vdash\; \rho \subseteq \sigma$	
(*congruence*)	$\sigma \subseteq \tau \;\vdash\; \forall t.\, \sigma \subseteq \forall t.\, \tau$	

are valid in $\mathrm{ML}_{\leq}$ when restricted to types for which they make sense. Rule (*sub*) is an instance of our subtyping rule. In particular, the restriction that $\mathbf{r}$ is not free in $\forall \mathbf{t}.\,\sigma$ corresponds to the use of a ϑ-substitution in rule *VIntro*, where ϑ are the universally quantified variables of the context.

Mitchell, Meldal, and Madhav [36] have proposed an extension of the Standard ML module system identifying types with specifications (which are a generalization of signatures with constraints) and objects with structures. Their system introduces two separate mechanisms for subtyping and inheriting specifications, and allows polymorphic specifications (i.e., templates). As for $\mathrm{ML}_{\leq}$, the main idea is to separate specification from implementation, and both systems allow programming using both objects and abstract data types. As a whole, the system is more expressive than ours, since structures with type components are first-class (whereas the module system we propose for $\mathrm{ML}_{\leq}$ only allows types to be defined at toplevel, as for Standard ML). Moreover, the encapsulation mechanism provided by structures is superior to that offered by $\mathrm{ML}_{\leq}$, and $\mathrm{ML}_{\leq}$ only provides a weak form of implementation inheritance. The impredicative treatment of explicit type parameters of specifications and methods is based on bounded polymorphism. This has the advantage of allowing parametric methods like *sort* with the following specification

$$sort\,[\mathsf{type}\ \alpha \leq \mathsf{ordered}[\alpha]]\colon \mathsf{list}[\alpha] \rightarrow \mathsf{list}[\alpha]$$

which is possible in $\mathrm{ML}_{\leq}$ in a restricted form (the most general form would require the introduction of type classes as in section 8.7). On the other hand, explicit type parameters can be a disturbance, and the $\mathrm{ML}_{\leq}$ approach making use of implicit parameters with an explicitly declared variance looks more pleasant. Moreover, turning structures into first-class objects (1) complicates the definition of type equality somewhat and leads to an operational definition which is not very intuitive, and (2) duplicates the functionality of records, which can lead to further confusion. Finally, the approach advocated in [36] defines a single-dispatch language, so that the kind of precise typing of binary operators allowed in $\mathrm{ML}_{\leq}$ is not possible, and methods are not ordinary functions which can be passed as parameter to other functions.

$\mathrm{ML}_{\leq}$ also has strong links with all systems derived from the Hindley-Milner type system, in particular, systems of overloaded functions built around the notion of type class, first proposed by Kaes [28], and then by Wadler and Blott [39, 44], or constructor class, proposed by Jones [27]. In essence, these systems [27, 39, 44] allow the instantiation of overloaded specifications which consist in type templates with a single type variable, such as the instantiation of the template $\langle \alpha, \alpha \rangle \rightarrow \alpha$ for $\alpha = \mathsf{int}$ and $\alpha = \mathsf{real}$. In the absence of any subtyping relation between int and real, such simple specifications cannot express complex types such as the type of the *sub* method and, in particular, do not allow the typing of mixed operations like $sub(1.2, 3)$. Multi-parameter type classes, i.e., type templates with more than one variable, have thus been proposed to lift this constraint. The idea is to use a template like $\langle \alpha, \beta \rangle \rightarrow \gamma$ and instantiate it with all possible interesting combinations of α, β and γ, for example $(\mathsf{int}, \mathsf{int}, \mathsf{int}), (\mathsf{int}, \mathsf{real}, \mathsf{real})$, etc., which, in fact, is fairly similar to the signatures of [14]. However, the problem with multi-parameter type classes is that type-checking is undecidable in general, that it is possible to overload functions with structurally different signatures, e.g., $(\mathsf{bool}, \mathsf{int} \rightarrow \mathsf{bool}, \mathsf{unit})$, and that the overloading resolution algorithm can be quite tricky and unintuitive, a mixture which has already proven very dangerous in C++. In contrast, a type like

$$\forall\alpha,\beta,\gamma\colon (\mathsf{int} \leq \gamma \wedge \alpha \leq \gamma \wedge \beta \leq \gamma).\ \alpha \to \beta \to \gamma$$

(which is equivalent to the type of the *sub* method) can be seen as a multi-parameter constrained template with restricted instantiations.

Duggan [19, 20, 40], and then Odersky, Wadler, and Wehr [39] have proposed the use of "kinded types", which are polymorphic constrained types with constraints on available instances of the operations used by function bodies. This approach is also related to the "methods as assertions" view developed by Abadi and Lamping [4]. For instance, the *insert* function on sets requires that the equality operator be instantiated at the type of the inserted element. This approach can be made to work under the "open world" assumption [19, 39], but types are rather hard to read, since they mention program functions, and lead to method specifications which are dependent on the program's text, which may be a problem for modularity and scalability. On the other hand, type inference is made easier by such an approach.

Another approach which is related to $\mathrm{ML}_{\leq}$ is the addition of *dynamics* to polymorphic languages. The theory of dynamics of Abadi, Cardelli, Pierce, Plotkin, and Rémy [1, 3] introduces a new type, `Dynamic`, which is the type of pairs of objects of any type together with a type tag. Dynamics are built using (`dynamic` $e\colon \tau$) expressions, where the type tag is explicit. The requirement that type tags be explicit is necessary to have minimal typing. Intuitively, `Dynamic` can be thought of as the existential type $\exists\alpha.\ \alpha$, or as the **`REFANY`** type of Modula 3. Dynamics allow the use of heterogeneous lists, the typing of I/O functions, etc., and dynamics are thus strictly more expressive than $\mathrm{ML}_{\leq}$. On the other hand, $\mathrm{ML}_{\leq}$ can have overloaded coercion functions like *string*, with type $\forall\alpha.\ \alpha \to \mathsf{string}$, dispatching on the outermost type constructor of the run-time type of arbitrary objects, which covers an important use of dynamics. The advantage of the $\mathrm{ML}_{\leq}$ approach is its simplicity and its uniformity, since no ad-hoc mechanism is needed to handle dynamics. One could argue, though, that the drawback of this approach is that every object has to be tagged by its minimal type at run-time, but, as noted in chapter 6, this is only partly true since only the outermost type constructor has to be maintained at run-time, and by a proper static analysis of programs, or explicit user declarations, we are convinced that performances similar to that of classical ML implementations can be achieved. Dispatching on dynamics is performed via `typecase` expressions which match the run-time tag of dynamics against patterns in sequence. These expressions have a mandatory "catch-all" clause which is applied when pattern-matching fails. This approach is thus less flexible than the $\mathrm{ML}_{\leq}$ approach which requires completeness and non-ambiguity of methods, and has a notion of "best-match", which, we believe, is important for object-oriented programming. On the other hand, patterns can be any monotype, including function types, and they can have repeated and second-order variables (although the use of tuple-variables in the latter case is rather complex). Consequently, patterns do not have to be linear and allow the inspection of functions. The price to pay for such features is that dynamic dispatch requires an exact match of patterns. In the presence of primitive subtyping, this exact match can be followed by a simple inequality constraint on the result of the match, which offers some of the flexibility offered in $\mathrm{ML}_{\leq}$. In contrast, $\mathrm{ML}_{\leq}$ does not "match" patterns. Rather, a consistency check $\tau \in \pi$ between the run-time type τ of objects and patterns π

is performed, and the possible "matches", even when ambiguous, are left implicit through the addition of a constraint to the run-time constraint context Δ. Also, we believe that the explicit tagging of dynamics is potentially dangerous, since tags influence the behavior of the program, and could be the source of malicious bugs in languages with primitive subtyping, since programmers may tag values by strict supertypes of their minimal type. Finally, the typing rule for `typecase` expressions of [1] has an infinite number of premisses, whereas our type-checking for methods is considerably simpler.

Leroy and Mauny [30] propose a system for adding dynamics to ML that unifies `typecase` statements with pattern-matching. Their first proposal, implemented in the CAML compiler [46], uses only universally quantified patterns. Their second proposal introduces existentially quantified patterns, as in $\mathrm{ML}_{\leq}$, and mixed patterns with universally and existentially quantified variables, which are very powerful. Patterns can be non-linear, and tagging of dynamics can sometimes be left implicit. However, patterns and polymorphic tags have to be closed, so that curried functions cannot share pattern variables, two restrictions that $\mathrm{ML}_{\leq}$ does not have, thanks to the notion of constraint context (cf. section 5.4). Again, this can probably be attributed to the fact that $\mathrm{ML}_{\leq}$ tags objects and closes polymorphic types automatically, and also to the fact that $\mathrm{ML}_{\leq}$ does not "match" run-time types to patterns (cf. supra). As for Abadi et al., their system does not have principal types without some explicit type annotations, which, as shown in section 7.2, is also probably the case for $\mathrm{ML}_{\leq}$ methods, although more work is required on that subject.

The "extensional polymorphism" approach advocated by Dubois, Rouaix, and Weis [18] is also related to $\mathrm{ML}_{\leq}$. For instance, in their system, the method of section 7.2 can be given the same type scheme $\forall\alpha.\ \alpha \to \alpha$ as in $\mathrm{ML}_{\leq}$. The patterns on which dynamic dispatch is performed are more general than ours, and a type inference algorithm is proposed. This algorithm is based on a global abstract interpretation of the program to check consistency, since the implementations of methods are required to be non-ambiguous, but are not required to be complete (cf. section 5.1). However, we believe that a generalization of this approach to primitive subtyping is unlikely (cf. theorems 43 and 50). Moreover, our approach does not rely on a global abstract interpretation, and is thus more modular.

Mitchell [33, 35], and Fuh and Mishra [22, 23] have studied the problem of type inference in the presence of primitive subtyping. Their models are not quite comparable to ours since primitive subtyping is not used to define methods, and our primary goal is not to perform type inference. Nonetheless, a comparison is interesting in that the instance relation between typing judgements is an instance of our subtyping rule. Using our notations (the cited papers use C for constraints κ, A for type assignments Γ, τ for monotypes θ, and S for substitutions σ), a judgement of the form $\kappa', \Gamma' \vdash e\colon \theta'$ is an *instance* of a judgement of the form $\kappa, \Gamma \vdash e\colon \theta$ if there exists some substitution σ such that $\Gamma' = \Gamma[\sigma]$, $\theta' = \theta[\sigma]$, $\forall\vartheta'.\ \kappa' \models \kappa[\sigma]$, and $\kappa[\sigma]$ is ϑ'-closed (where ϑ and ϑ' are the sets of free variables of κ and κ', which we assume to be disjoint). This way of proceeding is different from ours in that constraints are not part of polytypes, which seems perfectly natural in the context of type inference, but much less so in the context of an explicitely typed language, or in the simpler context of a modular language where the type of every function or method f has to be matched against the specification of f in the interface exporting f. Moreover, the fact that κ' and $\kappa[\sigma]$ are ϑ'-closed corresponds to the fact that the axiomatization of the implication used above is essentially the reflexive and transitive closure of constraints. In any case, our definition is more general in that it allows θ' to be a supertype of $\theta[\sigma]$ instead of requiring equality. As a matter of fact, the instance relation defined above implies that

$$\forall\vartheta'.\ \kappa' \models \kappa[\sigma] \wedge \theta[\sigma] = \theta'$$

so that, σ being a ϑ'-substitution,

$$\forall\vartheta'.\ \kappa' \models \kappa[\sigma] \wedge \theta[\sigma] \leq \theta'[\sigma]$$

and by rule *VIntro*

$$\forall\vartheta'.\ \kappa' \models \kappa \wedge \theta \leq \theta'$$

which proves that $\forall\vartheta\colon \kappa.\ \theta$ is a subtype of $\forall\vartheta'\colon \kappa'.\ \theta'$ w.r.t. the trivial constraint context *True*. Hence, principal typing up to equivalence w.r.t. the instance relation is somewhat related to our notion of minimal typing. However, our definition is more general and is extended to deal with existentially quantified constraint contexts occurring in the typing of explicitely typed functions and methods. Finally, the notion of "matching" and the MATCH algorithm of [35] is also present in our axiomatization of constraint implication, as shown by rule (*arrow-inverse*), which is an instance of our monotype elimination rule *MElim*.

Based on the same ideas, Mitchell and Jategaonkar [26] have developed a model which has interesting connections with ours. The idea is to extend ML pattern matching with flexible records and primitive subtyping, in order to allow some form of object-oriented programming. Their system can support *built-in* containment relations between base types and *built-in* operations with constrained types like the type $\forall\alpha,\beta\colon \alpha \leq \mathsf{real}.\ \alpha \rightarrow \alpha \rightarrow \alpha$ of the addition operator. They do not clearly have a semantic notion of containment between polymorphic constrained types, although they mention that the INSTANCE algorithm to decide the instance relation is NP-hard. At the end of the paper, they briefly and informally show how abstract types and subtypes can be defined, and show how flexible records allow the definition of a *move* method like the one of figure 1.2, with type

$$\forall t \subseteq \mathsf{point}.\ t \rightarrow t$$

so that color points are mapped to color points. However, methods cannot be overridden (that is, they can only have one implementation), the underlying model is inherently single-dispatch, with all the problems associated to the typing of multi-methods, and there is no provision for parameterized classes. Also, allowing multiple implementations of methods, as the authors mention in the conclusion, would probably raise difficult problems for type inference. Nonetheless, we believe that adding flexible records to $\mathrm{ML}_{\leq}$ may be an interesting direction in order to treat implementation inheritance in a more primitive way. The notion of "matching" developed in [35, 26] shows that rules *VElim* and *MElim* can probably be generalized to flexible records.

Kaes [29] has tackled the decidability of type inference in the context of overloading, subtyping, and recursive types, using constrained types which are more expressive than ours, and with a precise typing of arithmetic operators. Moreover, his notion of "structural similarity" is fairly close to our notion of constructor class. However, his paper does not address the problem of defining methods and performing dynamic dispatch, and does not provide an operational semantics.

Eifrig, Smith, and Trifonov [21] have developed a polymorphic class-based object-oriented system whose goals are fairly similar to ours. This system avoids the problems associated with matching [6], allows multiple-inheritance, has a decidable mimimal typing as well as a sound and complete inference algorithm based on recursive types. However, there are no parameterized classes (i.e., templates), and the treatment of binary methods is still not very satisfactory, which is inherent to all single-dispatch languages, and the authors acknowledge in the conclusion that the design of a module system should be difficult. Moreover, it is worth mentioning that the subtyping rule between recursively constrained types (definition 2.2) is sound w.r.t. ours, but less powerful in the sense that this rule relates monomorphic types w.r.t. some existentially quantified constraint, whereas our subtyping rule relates polymorphic types. In particular, we have seen that our rule is what is required to check that the specification of a method in an interface is matched by the implementation of the method, which gives modularity for free.

Aiken and Wimmers [5] have addressed the problem of performing type inference via set constraints. Their constraints have union, intersection and recursive types, and data constructors as nil and cons are part of their constraints, but they do not have primitive subtyping. They have a semantic ideal model of polymorphic constrained types which implicitly relates polytypes. However, this subtyping relation is not explicit, nor is it axiomatized, and the authors do not consider methods. An interesting question is whether set constraints could be used in $\mathrm{ML}_{\leq}$. We do not think so for two reasons. One is that if the conditional (which is not part of the expression language of [5]) had type $\forall X.\ \mathsf{bool} \to X \to X \to X$, then the expression ($\mathsf{cond\ true}\ 1\ ()$) would be well-typed since the set constraint $\{\mathsf{int} \subseteq X, \mathsf{unit} \subseteq X\}$ has a solution, which is rather annoying. The second reason is that set constraints are interpreted w.r.t. sets of terms of a free algebra, whereas $\mathrm{ML}_{\leq}$ constraints are interpreted w.r.t. terms built from an arbitrary partial order which is a given of the problem. This seems to make it very difficult, with set constraints, to reason about admissible extensions of the type hierarchy and axiomatize constraint implication (and, hence, subtyping) with results similar to theorem 22. Further work is required, however, to clarify the relationship between the two approaches.

9.2 Conclusion

We have presented a new object-oriented extension of a typed-version of ML, called $\mathrm{ML}_{\leq}$, based on a reinterpretation of datatype declarations as abstract and concrete class declarations, and of pattern-matching as dynamic dispatch. We believe that this extension, with a minimum number of concepts, is natural and powerful, and would be easy to learn for ML programmers, but the model we have developed is also very close to models emerging in the OO community. The static type system we propose for this language is a generalization of the classical Hindley-Milner predicative type system at the crossroad of two trends, namely type/constructor classes and primitive subtyping. Our typing of methods is stronger and more semantic than that offered by type classes or constructor classes, and provides for a precise typing of arithmetic operators. Moreover, we do not rely on the kind of complex overloading resolution strategies that must be used in the context of multi-parameter type classes and have proven harmful in languages like C++. Our methods are just ordinary functions which can be used as such, and in particular, can be passed as arguments to other functions, a property that most single-dispatch languages lack.

We believe that the *universally quantified implication of existentially quantified constraints*, which is at the heart of the system, is a unifying framework for many object-oriented extensions of ML, since the entire construction, and in particular the language, its typing rules, and its operational semantics, seems very robust with respect to the addition of new kinds of constraints and a modification of the axiomatization of the implication. In fact, only a few lemmas about this axiomatization, in particular theorems 43 and 45, are actually used in the proofs of crucial theorems. Also, the notion of functional type application, which turns functional types into monotonic type transformers, allows a more semantic and systematic presentation of the system, and in particular, of the application rule.

Finally, $\mathrm{ML}_{\leq}$ may be seen as a reference, explicitely typed language for research on type inference with subtypes, as well as a link between predicative and impredicative type systems, since the subtyping rule of $\mathrm{F}_{\leq}$ matches ours in an interesting way. It may also be interesting to compare $\mathrm{F}_{\leq}$ to the pure functional subset of $\mathrm{ML}_{\leq}$ which does not contain methods or constructor classes. Since monotypes can only be type variables in this system, the axiomatization of the implication is that given figure 7.1, which is simple enough for theoretical investigations.

On the minus side, the language we propose looses some of the simplicity that has made ML so popular. Universally quantified constrained types can be hard to read and understand, and type-checking errors can become hard to explain. Moreover, our language is explicitly typed, and it is not completely clear which type annotations could be automatically inferred by a compiler. We believe that the type of functions could be inferred. As for methods, we believe that type inference may be possible over some class of ML-like types, but more work is required on this topic. Finally, even though methods can be given very precise constrained types, programmers may choose to use classical ML types exclusively, unless precise types are required, so that programmers can only pay for what they actually use.

In order to allow the definition of new classes and new constructors, our system is based on an implicit "open-world" assumption. However, it may be desirable to treat some classes, for instance built-in classes, as "closed", in order to allow more sophisticated methods to type-check. This can be done easily by a stronger axiomatization of the implication.

Many interesting extensions could be studied, the first one being type classes, even if they are not as badly needed as they are in ML, since a generic *string* method with type $\forall\alpha.\ \alpha \to \mathsf{string}$ can be implemented in $\mathrm{ML}_{\leq}$. However, this type does not provide the compile-time safety offered by a type like $\forall\alpha\colon \mathit{String}(\alpha).\ \alpha \to \mathsf{string}$.

Adding the ability to define a partial order between constructor classes with the same arity and the same variance looks like another interesting extension, as are references (with the classical restriction on polymorphic references), dynamic dispatch inside covariant type constructors, and implementation inheritance. Adding recursive types should be more problematic w.r.t. decidability, since rule *VElim* cannot be stated with recursive types. Existential types may also be worth considering to handle I/O functions. We believe all the machinery is in place to integrate them easily. Finally, we would like to give an ideal-based or a PER-based semantic model of $ML_{\leq}$.

In conclusion, we want to emphasize the fact that all the examples of this report have been type-checked using a prototype implementation of $ML_{\leq}$ written in Caml Special Light [31] and implementing the greatest fixpoint algorithm of section 7.1. This implementation is just a "proof of concept" implementation that will be used as a testbed for a more efficient implementation in the type-checker of the new hardware description language $2z$, which originally motivated this work.

9.3 Acknowledgments

We wish to thank François Fages for stimulating discussions on the algorithm of section 7.1 for deciding implication.

Bibliography

[1] M. Abadi, L. Cardelli, B. Pierce, G. Plotkin. Dynamic Typing in a Statically-Typed Language. ACM Transactions on Programming Languages and Systems, 13(2) (1991) 237–268

[2] M. Abadi, L. Cardelli. A Theory of Primitive Objects: Second-Order Systems. Proc. of the European Symposium on Programming, Springer-Verlag (1994) 1–25

[3] M. Abadi, L. Cardelli, B. Pierce, D. Rémy. Dynamic Typing in a Dynamic Typing in Polymorphic Languages. Journal of Functional Programming, 5(1) (1995) 111–130

[4] M. Abadi, J. Lamping. Methods as Assertions. Theory and Practice of Object Systems 1, 1 (1995) 5–18

[5] A. Aiken, E. Wimmers. Type Inclusion Constraints and Type Inference. FPCA'93 (1993) 31–41

[6] K. B. Bruce. A paradigmatic object-oriented programming language: design, static typing and semantics. Journal of Functional Programming 4(2) (1994) 127–206

[7] K. B. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. T. Leavens, B. C. Pierce. On binary methods. Technical report, LIENS-95-14 (1995)

[8] François Bourdoncle. Abstract Interpretation By Dynamic Partitioning. Journal of Functional Programming, 2(4) (1992) 407–435

[9] François Bourdoncle. Efficient Chaotic Iteration Strategies with Widenings. Proc. of the Int. Conf. on Formal Methods in Programming and their Applications, LNCS 735, Springer-Verlag (1993) 128–141

[10] G. Castagna. $F^{\&}_{\leq}$: integrating parametric and ad-hoc second order polymorphism. Proc. of the 4th International Workshop on Database Programming Languages, Workshops in Computing, Springer-Verlag, (1993) 335–355

[11] G. Castagna, B. C. Pierce. Decidable Bounded Quantification. Proc. of the 21st Symp. on Principles of Programming Languages (1994) 151–162

[12] G. Castagna, B. C. Pierce. Corrigendum: Decidable Bounded Quantification. Proc. of the 21nd Symp. on Principles of Programming Languages (1995) 408–408

[13] G. Castagna. Covariance and contravariance: conflict without a cause. ACM Transactions on Programming Languages and Systems 17(3) (1995) 431–447

[14] C. Chambers, G. Leavens. Typechecking and Modules for Multi-Methods. Technical Report UW-CS TR 95-08-05, University of Washington (1995)

[15] P. Cousot, R. Cousot. Abstract Interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. Proc. of the 4th ACM Symp. on POPL (1977) 238–252

[16] P. L. Curien, G. Ghelli. Coherence of subsumption, minimum typing and the type checking of $F_{\leq}$. Mathematical Structures in Computer Science 2(1) (1992)

[17] L. G. DeMichiel, R. P. Gabriel. Common lisp object system overview. ECOOP'87, LNCS 276 (1987) 151–170

[18] C. Dubois, F. Rouaix, P. Weis. Extensional Polymorphism. Proc. of the 22nd Symp. on Principles of Programming Languages (1995)

[19] D. Duggan, J. Ophel. Kinded Parametric Overloading. Technical Report CS-94-35, University of Waterloo (1994)

[20] D. Duggan. Polymorphic Methods With Self Types for ML-like Languages. Technical Report CS-95-03, University of Waterloo (1995)

[21] J. Eifrig, S. Smith, V. Trifonov. Sound Polymorphic Type Inference for Objects. Proc. of OOPSLA'95 (1995) 169–184

[22] Y.-C. Fuh, P. Mishra. Type inference with Subtypes. 2nd European Symp. on Programming, LNCS 300 (1988) 94–114

[23] Y.-C. Fuh, P. Mishra. Polymorphic Subtype Inference: Closing the Theory-Practice Gap. TAPSOFT'89, LNCS 352 (1988) 167–183

[24] K. Hammond, editor. Report on the Programming Language Haskell, version 1.3 (1995)

[25] R. Hindley. The principal type-scheme of an object in combinatory logic. Trans. Amer. Math. Soc., 146 (1969) 29–60

[26] L. Jategaonkar, J. C. Mitchell. Type Inference with extended pattern matching and subtypes. Fundamenta Informaticae. 19 (1, 2) (1993) 127–166

[27] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. FPCA'93 (1993)

[28] S. Kaes. Parametric Polymorphism. Proc. of 2nd European Symp. on Programming, LNCS 300 (1988)

[29] S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. Proc. of Conf. on Lisp and Functional Programming (1992) 193–204

[30] X. Leroy, M. Mauny. Dynamics in ML. Journal of Functional Programming, 3(4) (1993) 109–122

[31] X. Leroy, Le systme Caml Special Light: modules et compilation efficace en Caml. Research Report 2721, INRIA (1995)

[32] R. Milner. A theory of type polymorphism in programming. Journal of Computer and System Sciences, vol. 17 (1978) 348–375

[33] J. C. Mitchell. Coercion and Type Inference (Summary). Proc. of the 11th ACM Symp. on Principles of Programming Languages (1984) 175–185

[34] J. C. Mitchell. Polymorphic Type Inference and Containment. Information and Computation 76 (1988) 211–249

[35] J. C. Mitchell. Type inference with simple subtypes. Journal of Functional Programming, 1(3) (1991) 245–285

[36] J. C. Mitchell, S. Meldal, N. Madhav. An extension of Standard ML modules with subtyping and inheritance. Proc. of the 18th ACM Symp. on Principles of Programming Languages (1991) 270–278

[37] B. Monsuez. Polymorphic Typing by Abstract Interpretation. Proc. of the 12th Conf. FST&TCS, LNCS Vol. 652, Springer Verlag (1992)

[38] B. Monsuez. Polymorphic Types and Widening Operators. Proc. of the Workshop on Static Analysis, LNCS Vol. 724, Springer Verlag (1993)

[39] M. Odersky, P. Wadler, M. Wehr. A second look at overloading. Proc. of the 7th conf. on Functional Programming and Computer Architecture (1995) 135–146

[40] J. Ophel, D. Duggan. Multi-Parameter Parametric Overloading. Technical report, University of Waterloo (1995) (submitted to publication)

[41] J. Palsberg. Efficient inference of object types. Proc. LICS'94 (1994) 186–195

[42] F. Pottier. Simplifying subtyping constraints. Proc. of 1996 ACM SIGPLAN Int. Conf. on Functional Programming (1996) (to appear)

[43] J. Vuillemin. On circuits and numbers. IEEE Transactions on Computers, 43(8) (1994) 868–879

[44] P. Wadler, S. Blott. How to make ad-hoc polymorphism less ad-hoc. Proc. of the 16th ACM Symp. on Principles of Programming Languages (1989) 60–76

[45] M. Wand, P. O'Keefe. On the complexity of type inference with coercion. Proc. of FPCA'89 (1989) 293–298

[46] P. Weis et al. The CAML reference manual, version 2.6.1. Technical Report 121, INRIA (1990)

Appendix A

Proofs

Proof of lemma 1

The only rules in which variable sets appear are *VIntro* and *VElim*. Moreover, every (ϑ, ϑ')-substitution is trivially a ϑ-substitution, and any constructor variable v'_C and C-variable set ϑ'_C satisfying $v'_C \# \vartheta'_C \# (\vartheta, \vartheta', \kappa)$ trivially satisfies $v'_C \# \vartheta'_C \# (\vartheta, \kappa)$. Therefore, any derivation for $\forall \vartheta, \vartheta'.\ \kappa \models \kappa'$ is a also derivation for $\forall \vartheta.\ \kappa \models \kappa'$. ■

Proof of lemma 2

Let us assume that the judgement $\forall \widehat{\vartheta}.\ \kappa \models \kappa'$ is derivable, and let ϑ' be a variable set such that $\kappa' \# \vartheta'$. Without loss of generality, we can assume that $\widehat{\vartheta}$ and ϑ' are disjoint. Let ϑ be the set of free variables of κ which do not belong to $\widehat{\vartheta}$, σ be a $\widehat{\vartheta}$-substitution mapping every v' in ϑ' to a fresh (that is, a variable which does not belong to $(\widehat{\vartheta}, \vartheta, \vartheta')$) and unique variable of the same kind (and every other variable to itself), and σ' be the $(\widehat{\vartheta}, \vartheta')$ substitution mapping every $\sigma(v')$ to v' (and every other variable to itself). It is easy to see that $\sigma' \circ \sigma$ is the identity. Moreover, by rule *VIntro*, we have

$$\forall \widehat{\vartheta}, \vartheta'.\ \kappa[\sigma' \circ \sigma] \models \kappa[\sigma]$$

that is,

$$\forall \widehat{\vartheta}, \vartheta'.\ \kappa \models \kappa[\sigma]$$

Now, by rule *VIntro* and the fact that σ is a $\widehat{\vartheta}$-substitution, we also have that

$$\forall \widehat{\vartheta}.\ \kappa[\sigma] \models \kappa$$

so that the judgement $\forall \widehat{\vartheta}.\ \kappa[\sigma] \models \kappa'$ is derivable. But since $\vartheta' \# \widehat{\vartheta}$ and no variable in ϑ' occurs free in $\kappa[\sigma]$ or in κ', we can assume that there exists a derivation $\mathcal{D}$ for this judgement which does not make use of any variable in ϑ', and in particular, we can assume that every $\widehat{\vartheta}$-substitution used in $\mathcal{D}$ is a $(\widehat{\vartheta}, \vartheta')$-substitution so that $\mathcal{D}$ can be readily translated into a derivation for the judgement $\forall \widehat{\vartheta}, \vartheta'.\ \kappa[\sigma] \models \kappa'$. We thus conclude by transitivity that the judgement $\forall \widehat{\vartheta}, \vartheta'.\ \kappa \models \kappa'$ is derivable. ■

Proof of lemma 3

We first start to note that a derivation $\mathcal{D}_j$, $j \in [1,2]$, for the implication $\forall\vartheta.\ \kappa_j \models \kappa'_j$ is a sequence of constraints $\kappa_j^0, \ldots, \kappa_j^{n_j}$ such that $\kappa_j = \kappa_j^0$, $\kappa'_j = \kappa_j^{n_j}$, and for every $i \in [1, n_j]$, the implication

$$\forall\vartheta.\ \kappa_j^{i-1} \models \kappa_j^i$$

is an instanciation of one of the rules and axioms of figure 3.1.

We first prove the theorem assuming that $\kappa_1 \# \kappa_2\ [\vartheta]$. Since $\kappa'_1 \# \kappa'_2\ [\vartheta]$, we can assume without loss of generality that the only variables shared between $\mathcal{D}_1$ and $\mathcal{D}_2$ are in ϑ. To prove the implication

$$\forall\vartheta.\ \kappa_1 \wedge \kappa_2 \models \kappa'_1 \wedge \kappa'_2$$

we first prove that $(\kappa_1^i \wedge \kappa_2)_{i \in [0,n_1]}$ is a derivation for $\forall\vartheta.\ \kappa_1 \wedge \kappa_2 \models \kappa'_1 \wedge \kappa_2$. This is trivially true initially, since $\kappa_1^0 = \kappa_1$ and thus $\kappa_1^0 \wedge \kappa_2 = \kappa_1 \wedge \kappa_2$. So let us assume that $\kappa_1^{i-1} \wedge \kappa_2$, $i \in [1, n_1]$, is implied by $\kappa_1 \wedge \kappa_2$ for all ϑ, and let us consider the rule R of which the implication $\forall\vartheta.\ \kappa_1^{i-1} \models \kappa_1^i$ in $\mathcal{D}_1$ is an instance. If R is anything but *VElim* or *VIntro*, it is easy to check that $\forall\vartheta.\ \kappa_1^{i-1} \wedge \kappa_2 \models \kappa_1^i \wedge \kappa_2$ is also an instance of R, which proves that $\kappa_1^i \wedge \kappa_2$ is implied by $\kappa_1 \wedge \kappa_2$ for all ϑ. If R is rule *VElim*, then obviously the newly introduced variables v'_C and ϑ'_C satisfy $v'_C \# \vartheta'_C \# (\vartheta, \kappa_1^{i-1})$ and by hypothesis on the variables of $\mathcal{D}_1$ and $\mathcal{D}_2$, clearly satisfy $v'_C \# \vartheta'_C \# (\vartheta, \kappa_1^{i-1}, \kappa_2)$, which shows that $\forall\vartheta.\ \kappa_1^{i-1} \wedge \kappa_2 \models \kappa_1^i \wedge \kappa_2$ is also an instance of rule *VElim*. Now, since any ϑ-substitution σ used in an instance of rule *VIntro* in derivation $\mathcal{D}_1$ leaves κ_2 invariant, we obviously have

$$\begin{array}{lll} \forall\vartheta. & \kappa_1^{i-1} \wedge \kappa_2 & (\kappa_1^{i-1} = \kappa_1^i[\sigma]) \\ \models & \kappa_1^i[\sigma] \wedge \kappa_2 & (\textit{Hypothesis}) \\ \models & (\kappa_1^i \wedge \kappa_2)[\sigma] & (\textit{VIntro}) \\ \models & \kappa_1^i \wedge \kappa_2 & \end{array}$$

which proves that $\kappa_1^i \wedge \kappa_2$ is implied by $\kappa_1 \wedge \kappa_2$ for all ϑ. We have therefore proven that

$$\forall\vartheta.\ \kappa_1 \wedge \kappa_2 \models \kappa'_1 \wedge \kappa_2$$

and we conclude the proof by showing by the same technique that

$$\forall\vartheta.\ \kappa'_1 \wedge \kappa_2 \models \kappa'_1 \wedge \kappa'_2$$

We have thus proven the theorem assuming that $\kappa_1 \# \kappa_2\ [\vartheta]$. If this condition does not hold, let S be the set of variables which belong to both κ_1 and κ_2 but do not belong to ϑ, σ be a ϑ-substitution mapping every variable in S to a fresh and unique variable of the same kind (and every other variable to itself), and σ' be the ϑ-substitution mapping every $\sigma(s)$ to s for every $s \in S$ (and every other variable to itself). Then σ' is the identity over the free variables of κ_1 and $\sigma' \circ \sigma$ is the identity over the entire set of variables. By rule *VIntro*, we thus have

$$\forall\vartheta.\ (\kappa_1 \wedge \kappa_2[\sigma])[\sigma'] \models \kappa_1 \wedge \kappa_2[\sigma]$$

that is

$$\forall\vartheta.\ \kappa_1[\sigma'] \wedge \kappa_2[\sigma' \circ \sigma] \models \kappa_1 \wedge \kappa_2[\sigma]$$

or else,

$$\forall\vartheta.\ \kappa_1 \wedge \kappa_2 \models \kappa_1 \wedge \kappa_2[\sigma]$$

But since, by rule *VIntro*, $\forall\vartheta.\ \kappa_2[\sigma] \models \kappa_2$, our hypothesis $\forall\vartheta.\ \kappa_2 \models \kappa_2'$ translates into $\forall\vartheta.\ \kappa_2[\sigma] \models \kappa_2'$ and since $\kappa_1 \mathrel{\#} \kappa_2[\sigma]\ \ [\vartheta]$, we conclude by our initial proof of the theorem that

$$\forall\vartheta.\ \kappa_1 \wedge \kappa_2[\sigma] \models \kappa_1' \wedge \kappa_2'$$

and finally

$$\forall\vartheta.\ \kappa_1 \wedge \kappa_2 \models \kappa_1' \wedge \kappa_2'$$

■

Proof of corollary 4

Since we treat constraints as sets of conjuncts, we have

$$\forall\vartheta.\ \kappa \models \kappa \wedge \kappa$$

and this corollary is thus a trivial consequence of lemma 3. ■

Proof of corollary 5

Since $\forall\vartheta.\ \kappa \models \kappa'$ and $\kappa'' \mathrel{\#} \kappa'\ \ [\vartheta]$, this is also a trivial consequence of lemma 3. ■

Proof of lemma 8

For the axioms of figure 3.1, one verifies that the right-hand sides are well-kinded if the left-hand sides are well-kinded. In particular, concerning *VIntro*, if γ is a unifier for $\lceil\kappa[\sigma]\rceil$, then $\gamma \circ \lceil\sigma\rceil$ is a unifier for $\lceil\kappa\rceil$, where $\lceil\sigma\rceil(v)$ is defined as $\lceil\sigma(v)\rceil$ for every variable v. Regarding *VElim*, if $\lceil\kappa\{v \simeq \phi_C[\Theta_C]\}\rceil$ has unifier γ, then $v[\gamma]$ must be of the form $C[\lceil\Theta\rceil[\gamma]]$. Hence, $\lceil\kappa \wedge v = v_C'[\vartheta_C']\rceil$ is unifiable (using the fact that v_C' and ϑ_C' were chosen as fresh variables). Finally, a straightforward inductive proof over the assumed derivation of $\forall\vartheta.\ \kappa_1 \models \kappa_2$ completes the proof of the lemma. ■

Proof of corollary 9

Immediate consequence of lemma 8, since *true* (which is $\mathsf{unit} \sqsubseteq \mathsf{unit}$) is obviously well-kinded. ■

Proof of lemma 10

Let $x \approx y$ denote either of $x \sqsubseteq y$, $y \sqsubseteq x$, $x \leq y$, or $y \leq x$. By induction on i, we prove for every constraint κ', all expressions x and y, and every $i \geq 0$ that

$$\forall\vartheta.\ \kappa' \wedge x \approx y \wedge v = \theta \models \kappa' \wedge \tilde{x}^{(i)} \approx \tilde{y}^{(i)} \wedge v = \theta$$

and

$$\forall\vartheta.\ \kappa' \wedge \tilde{x}^{(i)} \approx \tilde{y}^{(i)} \wedge v = \theta \models \kappa' \wedge x \approx y \wedge v = \theta$$

hold, where $\tilde{x}^{(i)}$ denotes the expression that results from x by substituting θ for all occurrences of v in x whose depth is at most i.

For occurrences of v in x of depth 0, the assertion follows using either *MTrans* or *CTrans*.

Otherwise, x or y (w.l.o.g. x) must be of the form $\phi_C[\Theta]$. By the assumption that κ is well-kinded, y must either be of the form $\phi'_C[\Theta']$, or it must be a type variable v'. If y is $\phi'_C[\Theta']$, we have

$$\begin{aligned}
\forall\vartheta.\ \kappa \wedge x \approx y \wedge v = \theta &\models \kappa \wedge \phi_C \sqsubseteq \phi'_C \wedge \Theta \leq_C \Theta' \wedge v = \theta && (\textit{MElim})\\
&\models \kappa \wedge \widetilde{\phi_C}^{(i-1)} \sqsubseteq \widetilde{\phi'_C}^{(i-1)} \wedge \widetilde{\Theta}^{(i-1)} \leq_C \widetilde{\Theta'}^{(i-1)} \wedge v = \theta && (\text{ind. hyp.})\\
&\models \kappa \wedge \tilde{x}^{(i)} = \tilde{y}^{(i)} \wedge v = \theta && (\textit{MIntro})
\end{aligned}$$

and the converse direction follows similarly. Note that the induction hypothesis is applied several times in the second step of the proof.

If y is v', the proof is similar, but applies *VElim* before *MElim*. ■

Proof of lemma 11

We consider each of the rules (N_1), (N_2), and (N_3) in turn. For (N_1), the assertion follows immediately, using rules *MElim* and *MIntro*. For (N_2), we have

$$\begin{aligned}
\forall\vartheta.\ S \wedge \kappa\{v \simeq \phi_C[\Theta]\} &\models S \wedge \kappa \wedge v = v_C[\vartheta_C] && (\textit{VElim})\\
&\models S \wedge v = v_C[\vartheta_C] \wedge \kappa[v_C[\vartheta_C]/v] && (\text{lemma 10})
\end{aligned}$$

and the converse direction is immediate by lemma 10 and rule *Approx*. The proof for (N_3) is similar, once we observe that

$$\forall\vartheta.\ S[v_C[\vartheta_C]/v] \wedge \kappa[v_C[\vartheta_C]/v] \models S[v_C[\vartheta_C]/v] \wedge \kappa[v_C[\vartheta_C]/v] \wedge v = v_C[\vartheta_C]$$

which follows using rules *MRef* and *VIntro*, since $v \notin \vartheta$.

Finally, the well-kindedness of κ' follows from the assumption that κ is well-kinded, using lemma 8. ■

Proof of lemma 12

Assume that

$$(S_0, \kappa_0) \Rightarrow_\vartheta \cdots \Rightarrow_\vartheta (S_i, \kappa_i) \Rightarrow_\vartheta \cdots \Rightarrow_\vartheta (S_n, \kappa_n)$$

where $(S_0, \kappa_0) = (\textit{True}, \kappa)$ and $(S_n, \kappa_n) = (S', \kappa')$. A straightforward inductive proof shows that for every $i \geq 0$, the constraint S_i is of the form $v_i^1 = \theta_i^1 \wedge \ldots \wedge v_i^k = \theta_i^k$ for some $k \geq 0$ such that all v_i^j are pairwise different and no v_i^j occurs in some θ_i^l or in κ_i, and $v_i^j \in \vartheta$ for all $j \in [1, k]$. In particular, $S_n \wedge \kappa_n$ satisfies the variable condition required of a constraint in prenormal form.

Since $S_0 \wedge \kappa_0$ is well-kinded, lemma 11 implies that $S_i \wedge \kappa_i$ is well-kinded for all $i \geq 0$. Therefore, if κ_n contained some atomic constraint which were not of the form $\phi_C \sqsubseteq \phi'_C$ or $v \leq v'$, it would have to either $\phi_C[\Theta] \leq \phi'_C[\Theta']$ or $v \simeq \phi_C[\Theta]$, both of which is impossible, because $S_n \wedge \kappa_n$ was assumed to be irreducible. ■

Proof of lemma 13

For an expression x over the algebra $\lceil \mathcal{T} \rceil$, we define its size $|x|$ inductively as follows:

$$|v| = |C| = 1, \quad |C[\theta_1, \ldots, \theta_n]| = 1 + |\theta_1| + \cdots + |\theta_n|$$

Assume, contrary to the assertion, that κ_0 is a well-kinded constraint such that there exists an infinite rewrite sequence

$$(S_0, \kappa_0) \Rightarrow_\vartheta (S_1, \kappa_1) \Rightarrow_\vartheta \cdots$$

where $S_0 \equiv \mathit{True}$. By lemma 11, we know that $S_i \wedge \kappa_i$ is well-kinded for all $i \geq 0$, hence there exist most general unifiers γ_i for $\lceil S_i \wedge \kappa_i \rceil$.

For each constraint κ_i, consider the number

$$|\kappa_i| = \sum_{x \in TV(\kappa_i)} |x[\gamma_i]|$$

where $TV(\kappa_i)$ denotes the set of type variables that occur in κ_i (and therefore also in $\lceil \kappa_i \rceil$). We will now prove that

1. $|\kappa_{i+1}| < |\kappa_i|$ if κ_{i+1} results from κ_i by an application of rule (N_2) or (N_3) and
2. $|\kappa_{i+1}| = |\kappa_i|$ if κ_{i+1} results from κ_i by an application of rule (N_1).

For the rules (N_2) or (N_3), we remark that

$$TV(\kappa_{i+1}) = (TV(\kappa_i) \setminus \{v\}) \cup \{v_C^1, \ldots, v_C^n\}$$

assuming $\vartheta_C = \{v_C^1, \ldots, v_C^n\}$, and that γ_i and γ_{i+1} agree on all variables x except v and $v_C^1, \ldots, v_C^n$. On the other hand, the fact that γ_i and γ_{i+1} are most general unifiers, and the definitions of $\lceil \kappa_i \rceil$ and $\lceil \kappa_{i+1} \rceil$ imply $v[\gamma_i] = (C[\vartheta_C])[\gamma_{i+1}]$ and therefore

$$\begin{array}{rcl} |v[\gamma_i]| & = & |(C[v_C^1, \ldots, v_C^n])[\gamma_{i+1}]| \\ & = & 1 + |v_C^1[\gamma_{i+1}]| + \cdots + |v_C^n[\gamma_{i+1}]| \\ & > & |v_C^1[\gamma_{i+1}]| + \cdots + |v_C^n[\gamma_{i+1}]| \end{array}$$

which implies $|\kappa_i| > |\kappa_{i+1}|$.

For the case of rule (N_1), it is easy to see that κ_i and κ_{i+1} contain the same set of variables, and that the unifiers γ_i and γ_{i+1} are identical, which proves the assertion.

On the other hand, applications of rule (N_1) strictly reduce the complexity of κ_{i+1} w.r.t. that of κ_i. (Formally, the complexity $\|\kappa\|$ of a constraint κ can be defined as the sequence of heights of expressions contained in κ, written in decreasing order. The lexicographic ordering on such sequences is well-founded.)

Therefore, defining the norm of κ_i as the pair $(|\kappa_i|, \|\kappa_i\|)$ and taking the lexicographic ordering on norms, which is well-founded, every application of a rewrite rule strictly decreases the norm of κ_{i+1} w.r.t. that of κ_i. The assumption of an infinite rewrite sequence therefore yields an infinite descending sequence of norms, which gives the desired contradiction and proves the lemma. ■

Proof of theorem 14

From lemmas 11, 12, and 13, it follows that rewriting with $\Rightarrow_\vartheta$ yields a constraint in prenormal form for any well-kinded constraint κ. By corollary 9, this holds *a fortiori* for any well-formed constraint κ. ■

Proof of lemma 15

Straightforward by simultaneous induction on (the length of) the derivations. ■

Proof of lemma 16

The proof is by induction on the length of the assumed derivation of $\forall\vartheta.\ \kappa_1 \models \kappa_2$. Assume, therefore, that $\forall\vartheta.\ \kappa 1 \models \kappa 3$ has a shorter derivation (possibly of length 0, if $\kappa 1 \equiv \kappa 3$), and that $\forall\vartheta.\ \kappa 3 \models \kappa 2$ is an instance of one of the axioms of figure 3.1. By the induction hypothesis (or trivially, if $\kappa 1 \equiv \kappa 3$), there exists some substitution σ_3 that agrees with σ_1 for the variables in ϑ such that $\kappa_3[\sigma_3]$ is ground and satisfied in $\mathcal{T}^*$.

Approx Then $\kappa_3\{\kappa_2\}$, and the assertion follows immediately, choosing $\sigma_2 = \sigma_3$.

CRef Then $\kappa_2 \equiv \kappa_3 \wedge \phi_C \sqsubseteq \phi_C$. If $\phi_C[\sigma_3]$ is ground, then we may choose $\sigma_2 = \sigma_3$, and the assertion follows by the reflexivity of $\sqsubseteq_C$ in $\mathcal{T}^*$. Otherwise, $\phi_C \equiv v_C$ for some constructor variable v_C of type constructor class C that does not occur in κ_3 (otherwise $\kappa_3[\sigma_3]$ would not be ground) or ϑ (otherwise, $\phi_C[\sigma_1]$ and therefore $\phi_C[\sigma_3]$ would be ground). Let t_C be some type constructor of class C (which is non-empty by definition), and let $\sigma_2 = \sigma_3 \circ \{v_C \mapsto t_C\}$. Then σ_2 agrees with σ_1 for the variables in ϑ, $\kappa_3[\sigma_2] \equiv \kappa_3[\sigma_3]$, and $\kappa_2[\sigma_2]$ is a ground constraint satisfied in $\mathcal{T}^*$.

CTrans Immediate from the induction hypothesis and the transitivity of $\sqsubseteq_C$, choosing $\sigma_2 = \sigma_3$.

CTriv Then $\kappa_2 \equiv \kappa_3 \wedge t_C \sqsubseteq t'_C$ for some ground type constructors such that $t_C \sqsubseteq_C t'_C$ holds in $\mathcal{T}$. By the definition of admissible extension, $t_C \sqsubseteq_C t'_C$ holds in $\mathcal{T}^*$, too, which implies the assertion, choosing $\sigma_2 = \sigma_3$.

CMin Then $\kappa_2 \equiv \kappa_3 \wedge d_C \sqsubseteq \phi_C$, where $\kappa_3\{\phi_C \sqsubseteq d_C\}$. Since $\kappa_3[\sigma_3]$ is ground and satisfied in $\mathcal{T}^*$, we have $\phi_C[\sigma_3] \sqsubseteq_C d_C$ in $\mathcal{T}^*$. The requirement that data type constructors be minimal implies $d_C \sqsubseteq_C \phi_C[\sigma_3]$ and therefore the assertion, choosing $\sigma_2 = \sigma_3$.

VIntro Then $\kappa_3 \equiv \kappa_2[\pi]$ for some substitution $\pi \in \mathcal{S}(\vartheta)$. Let $\sigma_2 = \sigma_3 \circ \pi$. Then for all $v \in \vartheta$ we have $v[\sigma_2] = v[\sigma_3] = v[\sigma_1]$ and $\kappa_2[\sigma_2] \equiv \kappa_3[\sigma_3]$, which proves the assertion.

VElim Then $\kappa_3\{v \simeq \phi_C[\Theta_C]\}$, w.l.o.g. we may assume $\kappa_3\{v \leq \phi_C[\Theta_C]\}$. By the assumption, $(v \leq \phi_C[\Theta_C])[\sigma_3]$ is ground and holds in $\mathcal{T}^*$. Since $\phi_C[\Theta_C][\sigma_3]$ is of the form $t''_C[\Theta''_C]$ for some ground type constructor t''_C and ground monotypes Θ''_C, the definition of the standard order on $\mathcal{T}^*$ implies that $v[\sigma_3]$ is of the form $t'_C[\Theta'_C]$. Further, $\kappa_2 \equiv \kappa_3 \wedge v = v'_C[\vartheta'_C]$, and the side condition of rule *VElim* implies that neither v'_C nor any of the ϑ'_C occur in ϑ or κ_3. Let therefore $\sigma_2 = \sigma_3 \circ \{v'_C \mapsto t'_C, \vartheta'_C \mapsto \Theta'_C\}$, then σ_2 and σ_3 (and therefore σ_2 and σ_1) agree on all variables in ϑ, $\kappa_2[\sigma_2] \equiv \kappa_3[\sigma_3]$, and $\kappa_2[\sigma_2]$ is a ground constraint that is satisfied in $\mathcal{T}^*$.

MRef Then $\kappa_2 \equiv \kappa_3 \wedge \theta \leq \theta$. Let ϑ_θ denote the set of variables that occur in $\theta[\sigma_3]$, and let the substitution σ_2 differ from σ_3 in that it maps constructor variables v_C in ϑ_θ to some type constructor t_C of appropriate sort, and type variables v in ϑ_θ to type unit. Then $\kappa_3[\sigma_2] \equiv \kappa_3[\sigma_3]$, and $\kappa_2[\sigma_2]$ is a ground constraint that is satisfied in $\mathcal{T}^*$.

MTrans Immediate from the induction hypothesis and the transitivity of $\leq$, choosing $\sigma_2 = \sigma_3$.

MIntro, MElim From the fact that $\kappa_3[\sigma_3]$ is satisfied in $\mathcal{T}^*$ and the definition of the standard order in $\mathcal{T}^*$, choosing $\sigma_2 = \sigma_3$.

■

Proof of lemma 17

First, we have to show that $\mathcal{T}^\vartheta$ is indeed a type structure. By definition, every set C^ϑ and (if present) the set T^ϑ is non-empty, hence also $C^\vartheta / \tilde{=}_C$ and $T^\vartheta / \tilde{=}_T$ are non-empty sets. Rule *CTrans* and the definition of $\tilde{=}_C$ ensure that $\phi_C \sqsubseteq^1_C \phi'_C$ whenever $\phi''_C \sqsubseteq^1_C \phi'_C$ and $\phi_C \tilde{=}_C \phi''_C$. Therefore, the relations $\sqsubseteq^1_C$ are well-defined. A similar argument shows that $\leq^1$ is well-defined. All relations $\sqsubseteq^1_C$ and $\leq^1$ are reflexive and transitive by rules *CRef*, *CTrans*, *MRef*, and *MTrans*, and antisymmetric by the definition of their domains as equivalence classes.

We now show that data type constructors are minimal. This is trivial for the class T whose set of data type constructors is empty. Assume that $\phi'_C \sqsubseteq^1_C \phi_C$ and $\phi_C \tilde{=}_C d_C$ for some $d_C \in D_C$. Hence $\forall\vartheta.\ \kappa_1 \models \phi'_C \sqsubseteq d_C$, and rule *CMin* implies $\forall\vartheta.\ \kappa_1 \models d_C \sqsubseteq \phi'_C$. Again using the definitions of $\sqsubseteq^1_C$ and $\tilde{=}_C$, it follows that $\phi_C \sqsubseteq^1_C \phi'_C$, which proves that ϕ_C is minimal. Hence, $\mathcal{T}^\vartheta$ is indeed a type structure.

To see that $\mathcal{T}^\vartheta$ is an admissible extension of $\mathcal{T}$, let t_C and t'_C be two type constructors of class C contained in $\mathcal{T}$. If $t_C \sqsubseteq_C t'_C$, rule *CIntro* implies $t_C \sqsubseteq^1_C t'_C$. Conversely, let us show that $t_C \sqsubseteq^1_C t'_C$ implies $t_C \sqsubseteq_C t'_C$. We know that $\forall\vartheta.\ \kappa_1 \models t_C \sqsubseteq t'_C$ by definition of $\sqsubseteq^1_C$, and thus $\forall\emptyset.\ \kappa_1 \models t_C \sqsubseteq t'_C$ by lemma 1. On the other hand, we have $\forall\emptyset.\ \mathit{true} \models \kappa_1$ by the assumption that κ_1 is well-formed, and therefore *Trans* implies $\forall\emptyset.\ \mathit{true} \models t_C \sqsubseteq t'_C$. By lemma 15, it follows that $t_C \sqsubseteq_C t'_C$ holds in $\mathcal{T}$. Hence, $\mathcal{T}^\vartheta$ contains (an isomorphic copy of, due to the quotient modulo $\tilde{=}_C$) a superset of every type constructor class of $\mathcal{T}$ such that the orderings of type constructors of $\mathcal{T}$ are preserved. ■

Proof of lemma 18

The "if" part follows because $\mathcal{T}^\vartheta$ is an admissible extension of $\mathcal{T}$ (up to isomorphism), and therefore lemma 16 ensures that $\mathcal{T}^\vartheta$ satisfies κ_1 if $\forall\vartheta.\ \kappa_1 \models \kappa$ holds. (Observe that we choose σ_1 as the obvious substitution that maps ϕ_C to $[\phi_C]$ and v to $[v]$, and that there are no variables to substitute for in $\kappa[\sigma_1]$ due to the variable assumption.) For the proof of the "only if" part we proceed by induction on the structure of κ.

$\kappa \equiv \phi_C \sqsubseteq \phi'_C$ In this case, the assertion follows immediately by the definition of $\sqsubseteq^1_C$.

$\kappa' \equiv \theta \simeq \theta'$ We proceed by induction on the definition of $\leq^1$. Assume that $\theta \leq^1 \theta'$. Since θ and θ' are ground monotypes of type structure $\mathcal{T}^\vartheta$, it follows that $\theta \equiv \phi_C[\Theta_C]$

and $\theta' \equiv \phi'_C[\Theta'_C]$ (where C might be T if Θ_C and Θ'_C are empty). Then rule *VElim* applied w.r.t. $\mathcal{T}^\vartheta$ ensures $\phi_C \sqsubseteq^1_C \phi'_C$ and $\Theta_C \leq^1_C \Theta'_C$. By the definition of $\sqsubseteq^1_C$, it follows that $\forall\vartheta.\ \kappa_1 \models \phi_C \sqsubseteq \phi'_C$, and the induction hypothesis and corollary 4 yield $\forall\vartheta.\ \kappa_1 \models \Theta_C \leq_C \Theta'_C$. Another application of corollary 4 gives us $\forall\vartheta.\ \kappa_1 \models \phi_C \sqsubseteq \phi'_C \wedge \Theta_C \leq_C \Theta'_C$, and finally, $\forall\vartheta.\ \kappa_1 \models \theta \leq \theta'$ follows by rule *MIntro*.

$\kappa' \equiv \kappa'_1 \wedge \kappa'_2$ By induction hypothesis, we obtain $\forall\vartheta.\ \kappa_1 \models \kappa'_1$ and $\forall\vartheta.\ \kappa_1 \models \kappa'_2$, from which corollary 4 yields the assertion.

■

Proof of lemma 19

By lemma 17, $\mathcal{T}^\vartheta$ is an admissible extension of $\mathcal{T}$. We interpret κ_1 as a $\mathcal{T}^\vartheta$-ground constraint (somewhat loosely identifying ϕ_C and $[\phi_C]$ and v and $[v]$, which we may do by lemma 17). By the assumption, there exists some $\mathcal{T}^\vartheta$-ground substitution σ_2 of the variables of κ_2 that are not in ϑ such that $\kappa_2[\sigma_2]$ is a ground constraint (over $\mathcal{T}^\vartheta$) satisfied in $\mathcal{T}^\vartheta$. Therefore, by lemma 18, it follows that $\forall\vartheta.\ \kappa_1 \models \kappa_2[\sigma_2]$, and by the variable assumptions, we may apply *VIntro* to derive $\forall\vartheta.\ \kappa_1 \models \kappa_2$. ■

Proof of lemma 20

The "if" part is trivial. For the "only if" part, let ϑ' be the set of variables in ϑ which are distinct from v. By rules *MRef* and *VIntro*, we can easily show that $\forall\vartheta'.\ \kappa_1 \models v = \theta \wedge \kappa_1$, and by hypothesis and lemma 1, we conclude that $\forall\vartheta'.\ v = \theta \wedge \kappa_1 \models \kappa_2$, so that by transitivity, $\forall\vartheta'.\ \kappa_1 \models \kappa_2$, and by lemma 2, we finally have $\forall\vartheta.\ \kappa_1 \models \kappa_2$. ■

Proof of lemma 21

Assume that $\kappa_1 \equiv \kappa_s \wedge \kappa_b$, where $\kappa_s \equiv v_1 = \theta_1 \wedge \ldots \wedge v_n = \theta_n$ and κ_b is a constraint in base form. Let σ be the substitution that maps v_i to θ_i for $i = [1, n]$.

$$\begin{array}{lll} & \forall\vartheta.\ \kappa_s \wedge \kappa_b \models \kappa_2 & \\ \iff & \forall\vartheta.\ \kappa_s \wedge \kappa_b \models \kappa_2 \wedge \kappa_s & \text{(corollary 4, } \textit{Approx}\text{)} \\ \iff & \forall\vartheta.\ \kappa_s \wedge \kappa_b \models \kappa_2[\sigma] \wedge \kappa_s & \text{(lemma 10)} \\ \iff & \forall\vartheta.\ \kappa_s \wedge \kappa_b \models \kappa_2[\sigma] & \text{(corollary 4, } \textit{Approx}\text{)} \\ \iff & \forall\vartheta.\ \kappa_b \models \kappa_2[\sigma] & \text{(lemma 20)} \end{array}$$

where lemma 20 is applied several times for the last step of the derivation. Finally, since κ_b is a constraint in base form, lemma 19 implies the assertion for $\forall\vartheta.\ \kappa_b \models \kappa_2[\sigma]$. ■

Proof of theorem 22

Direct consequence of theorem 14 and lemma 21. ■

Proof of corollary 23

Trivial consequence of theorem 22. ■

Proof of lemma 24

A somewhat tedious proof by induction on the derivation shows that whenever $\forall\vartheta.\kappa_1 \models \kappa_2$, then there is a substitution $\sigma \in \mathcal{S}(\vartheta)$ and a derivation of $\forall\vartheta.\ \kappa_1 \models \kappa_2[\sigma] \wedge \kappa_3$, for some constraint κ_3, without applications of *VIntro*. ■

Proof of lemma 25

We only have to show the "only if" part.

Consider a normal derivation of $\forall\vartheta.\ \kappa_1 \models \kappa_2$ (which exists by lemma 24). It ends in an application of *VIntro*, followed by *Approx*, so there exists a substitution $\sigma \in \mathcal{S}(\vartheta)$ such that $\forall\vartheta.\ \kappa_1 \models \kappa_2[\sigma] \wedge \kappa_3$. But since κ_2 is ϑ-closed, σ must be the identity substitution, so we may construct a derivation that does not use *VIntro* at all.

Now, a straightforward inductive proof shows that all constraints κ that appear in this derivation are simple and that $\forall\vartheta.\ \kappa_1 \models \mathit{atomize}(\kappa)$ can be shown by a normal derivation that does not use either *VIntro*, *MIntro*, or *MElim* (note that each κ is well-kinded by lemma 8). In particular, rule *VElim* is never applied. But since κ_2 is in base form, $\mathit{atomize}(\kappa_2) \equiv \kappa_2$, and we need never apply rules *VIntro*, *VElim*, *MIntro*, or *MElim*, which completes the proof of the lemma. Further more, note that *MRef* is only applied to monotypes θ which are type variables. ■

Proof of lemma 26

By lemma 25, $\forall\vartheta.\ \kappa_1 \models \kappa_2$ can only hold if there is a derivation of $\forall\vartheta.\ \kappa_1 \models \kappa_2$ that uses only the rules *Approx*, *CRef*, *CTrans*, *CTriv*, *CMin*, *MRef*, and *MTrans*. But applications of these rules do not create new symbols except base type constructors of classes in $\mathcal{T}$ and variables in ϑ (note that *MRef* need only be applied to monotypes θ which are type variables). Since the set of available symbols is thus finite, we can systematically apply these rules (deleting possible duplicates) until no application of any rule results in a new constraint. Call the resulting constraint κ_1^*. By the definition of $\mathcal{T}^\vartheta$ in lemma 17 of section 3.4 and lemma 25 above, κ_1^* is a finite syntactic representation of the type structure $\mathcal{T}^\vartheta$. From lemma 18, we know that $\forall\vartheta.\ \kappa_1 \models \kappa_2$ holds iff the ϑ-closed base constraint κ_2 is satisfied in $\mathcal{T}^\vartheta$. Therefore, $\forall\vartheta.\ \kappa_1 \models \kappa_2$ holds iff κ_2 is a subconstraint of κ_1^*, which is effectively decidable. ■

Proof of lemma 27

Let us first prove that $\forall\vartheta.\ \kappa_1 \models \kappa_2$ holds iff there exists a substitution $\sigma \in \mathcal{S}(\vartheta)$ such that $\kappa_2[\sigma]$ is ϑ-closed and $\forall\vartheta.\ \kappa_1 \models \kappa_2[\sigma]$ holds. The "if" direction is simply an instance of *VIntro*. For the "only if" part, lemma 18 implies that the type structure $\mathcal{T}^\vartheta$ satisfies κ_1 (because κ_1 is ϑ-closed), and by lemma 17, $\mathcal{T}^\vartheta$ is an admissible extension of $\mathcal{T}$. Therefore, the assumption $\forall\vartheta.\ \kappa_1 \models \kappa_2$ implies (by theorem 22) that there exists some substitution $\sigma \in \mathcal{S}(\vartheta)$ such that $\kappa_2[\sigma]$ is ϑ-closed and satisfied in $\mathcal{T}^\vartheta$. By lemma 18, this implies $\forall\vartheta.\ \kappa_1 \models \kappa_2[\sigma]$ as required.

Even more, we can require σ to be *flat*, that is, $v[\sigma]$ need never be of the form $\phi_C[\Theta_C]$ for a non-empty C-monotype list Θ_C of terms, as we show now. Denote by ϑ_2 the set of variables that occur in κ_2 but not in ϑ, and by ϑ_2' the (largest) set of variables in ϑ_2 that only occur in constraints of the form $v \leq v'$ or $v_C \sqsubseteq v_C'$ where both v and v' (or v_C and

v'_C) are in ϑ'_2. Clearly, all constraints involving variables from ϑ'_2 are "useless" in that they can be satisfied by assigning unit (resp., some consistently chosen type constructor of class C, which is non-empty) to all these variables. For the remaining variables $v \in \vartheta_2$, there exist chains $v_0 \approx v_1, \ldots, v_{n-1} \approx v_n$ such that $v_0 \equiv v$ and $v_n \in \vartheta$ (where $\approx$ denotes either $\sqsubseteq$, $\sqsupseteq$, $\leq$, or $\geq$). A simple inductive proof on the length of these chains shows that $v[\sigma]$ must be flat, for any substitution σ such that $\forall\vartheta.\ \kappa_1 \models \kappa_2[\sigma]$ holds (or, equivalently, such that $\kappa_2[\sigma]$ holds in $\mathcal{T}^\vartheta$).

However, there are only finitely many substitutions $\sigma \in \mathcal{S}(\vartheta)$ that map all variables of κ_2 to flat ground terms in $\mathcal{T}^\vartheta$, and for any such substitution σ, lemma 26 tells us that $\forall\vartheta.\ \kappa_1 \models \kappa_2[\sigma]$ is decidable, which completes the proof. ■

Proof of theorem 28

Without loss of generality, we may assume that $\kappa_1 \not\# \kappa_2\ [\vartheta]$. The assumption that κ_1 is well-formed and theorem 14 imply that there exists a constraint $\overline{\kappa_1} \equiv \kappa_s \wedge \kappa_b$ in ϑ-prenormal form which is ϑ-equivalent to κ_1. Again w.l.o.g., we may choose $\overline{\kappa_1}$ such that $\overline{\kappa_1} \not\# \kappa_2\ [\vartheta]$. Denoting the set of variables that appear in $\overline{\kappa_1}$ but not in ϑ by ϑ_1, lemmas 1 and 2 tell us that $\forall\vartheta.\ \overline{\kappa_1} \models \kappa_2$ holds iff $\forall\vartheta,\vartheta_1.\ \overline{\kappa_1} \models \kappa_2$ holds.

The constraint κ_s is of the form $v_1 = \theta_1 \wedge \ldots \wedge v_n = \theta_n$. Let us denote by σ the substitution that maps v_i to θ_i for $i = [1, n]$. Then the same chain of equivalences used in the proof of lemma 21 shows that $\forall\vartheta,\vartheta_1.\ \overline{\kappa_1} \models \kappa_2$ holds iff $\forall\vartheta,\vartheta_1.\ \kappa_b \models \kappa_2[\sigma]$ holds.

If $\kappa_2[\sigma]$ is not well-kinded, $\forall\vartheta,\vartheta_1.\ \kappa_b \models \kappa_2[\sigma]$ cannot hold, because κ_1, and therefore $\overline{\kappa_1}$ and κ_b, are well-kinded (by corollary 9 and theorem 14), and therefore $\kappa_2[\sigma]$ would have to be well-kinded by lemma 8.

Otherwise, by theorem 14 there exists some constraint $\kappa' \equiv \kappa'_s \wedge \kappa'_b$ in (ϑ,ϑ_1)-prenormal form that is (ϑ,ϑ_1)-equivalent to $\kappa_2[\sigma]$. If κ'_s is non-trivial, it contains some constraint of the form $v = \theta$ for $v \in (\vartheta,\vartheta_1)$ such that v does not occur in θ nor in κ_b (because $\overline{\kappa_1}$ is in (ϑ,ϑ_1)-prenormal form). In this case, $\forall\vartheta,\vartheta_1.\ \kappa_b \models \kappa'$ (and therefore $\forall\vartheta,\vartheta_1.\ \kappa_b \models \kappa_2[\sigma]$) cannot hold, because for any substitution σ' that satisfies $\kappa_b \wedge \kappa'$ (over some admissible extension $\mathcal{T}^*$ of $\mathcal{T}$) we may find a v-variant σ'' of σ' that satisfies κ_b, but falsifies $v = \theta$, and therefore κ'.

Otherwise, $\forall\vartheta,\vartheta_1.\ \kappa_b \models \kappa'$ is equivalent to $\forall\vartheta,\vartheta_1.\ \kappa_b \models \kappa'_b$, and by lemma 27, the latter is decidable.

Finally, decidability of well-formedness is an immediate corollary, since a constraint κ is well-formed iff $\forall\emptyset.\ \mathit{true} \models \kappa$, which is decidable, as shown above. ■

Proof of lemma 29

Let $\Delta = (\widehat{\vartheta} : \widehat{\kappa})$ be an implicit well-formed context, τ_i, $i \in [1, 2]$, be two well-formed types $\forall\vartheta_i : \kappa_i.\ \theta_i$, and $\widehat{v}$ be a fresh type variable not in $(\widehat{\vartheta}, \vartheta_1, \vartheta_2)$. Let us first assume that τ_1 is a subtype of τ_2. There must exist a renaming $\sigma \in \mathcal{R}(\widehat{\vartheta}; \vartheta_1; \vartheta_2)$ such that

$$\forall\widehat{\vartheta},\vartheta_2.\ \widehat{\kappa} \wedge \kappa_2 \models \kappa_1[\sigma] \wedge \theta_1[\sigma] \leq \theta_2$$

but $\widehat{v}$ being fresh, we can assume without loss of generality that σ is also a $(\widehat{\vartheta}, \widehat{v})$-substitution. By rule *Approx* and lemma 2, we deduce that

$$\forall\widehat{\vartheta},\vartheta_2,\widehat{v}.\ \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v} \models \kappa_1[\sigma] \wedge \theta_1[\sigma] \leq \theta_2$$

and by corollary 5, we deduce that

$$\forall \widehat{\vartheta}, \vartheta_2, \widehat{v}.\ \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v} \models \kappa_1[\sigma] \wedge \theta_1[\sigma] \leq \theta_2 \wedge \theta_2 \leq \widehat{v}$$

which shows, by rule *MTrans* and lemma 1, that

$$\forall \widehat{\vartheta}, \widehat{v}.\ \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v} \models \kappa_1[\sigma] \wedge \theta_1[\sigma] \leq \widehat{v}$$

but σ being a $(\widehat{\vartheta}, \widehat{v})$-substitution, we have

$$\forall \widehat{\vartheta}, \widehat{v}.\ \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v} \models (\kappa_1 \wedge \theta_1 \leq \widehat{v})[\sigma]$$

and finally, by rule *VIntro*,

$$\forall \widehat{\vartheta}, \widehat{v}.\ \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v} \models \kappa_1 \wedge \theta_1 \leq \widehat{v}$$

Let us now assume that

$$\forall \widehat{\vartheta}, \widehat{v}.\ \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v} \models \kappa_1 \wedge \theta_1 \leq \widehat{v}$$

Since τ_1 and τ_2 are well-formed and $\widehat{v}$ is fresh, ϑ_1 and ϑ_2 are thus disjoint from $(\widehat{\vartheta}, \widehat{v})$, so that $\mathcal{R}(\widehat{\vartheta}, \widehat{v}; \vartheta_1; \vartheta_2)$ is non-empty. Let σ be any renaming in $\mathcal{R}(\widehat{\vartheta}, \widehat{v}; \vartheta_1; \vartheta_2)$. We know that σ is a bijection and that σ^{-1} is a $(\widehat{\vartheta}, \widehat{v})$-substitution. Now, by rule *MRef*

$$\forall \widehat{\vartheta}, \vartheta_2.\ \widehat{\kappa} \wedge \kappa_2 \models \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \theta_2 \wedge \theta_2 \leq \theta_2$$

and by rule *VIntro*

$$\forall \widehat{\vartheta}, \vartheta_2.\ \widehat{\kappa} \wedge \kappa_2 \models \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v} \wedge \widehat{v} \leq \theta_2$$

But by hypothesis

$$\forall \widehat{\vartheta}, \widehat{v}.\ \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v} \models \kappa_1 \wedge \theta_1 \leq \widehat{v}$$

so that since $\sigma^{-1} \circ \sigma$ is the identity over variables

$$\forall \widehat{\vartheta}, \widehat{v}.\ \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v} \models (\kappa_1 \wedge \theta_1 \leq \widehat{v})[\sigma^{-1} \circ \sigma]$$

and by rule *VIntro*

$$\forall \widehat{\vartheta}, \widehat{v}.\ \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v} \models (\kappa_1 \wedge \theta_1 \leq \widehat{v})[\sigma]$$

that is

$$\forall \widehat{\vartheta}, \widehat{v}.\ \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v} \models \kappa_1[\sigma] \wedge \theta_1[\sigma] \leq \widehat{v}$$

so that by lemma 2, noting that

$$(\kappa_1[\sigma] \wedge \theta_1[\sigma] \leq \widehat{v}) \mathrel{\#} \vartheta_2$$

we deduce that

$$\forall \widehat{\vartheta}, \vartheta_2, \widehat{v}.\ \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v} \models \kappa_1[\sigma] \wedge \theta_1[\sigma] \leq \widehat{v}$$

which, by rule *Approx*, shows that

$$\forall \widehat{\vartheta}, \vartheta_2, \widehat{v}.\ \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v} \wedge \widehat{v} \leq \theta_2 \models \kappa_1[\sigma] \wedge \theta_1[\sigma] \leq \widehat{v}$$

and thus, by corollary 5,

$$\forall \widehat{\vartheta}, \vartheta_2, \widehat{v}.\ \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v} \wedge \widehat{v} \leq \theta_2 \models \kappa_1[\sigma] \wedge \theta_1[\sigma] \leq \widehat{v} \wedge \widehat{v} \leq \theta_2$$

that is, by rules *MTrans* and *Approx* and lemma 1,

$$\forall \widehat{\vartheta}, \vartheta_2.\ \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v} \wedge \widehat{v} \leq \theta_2 \models \kappa_1[\sigma] \wedge \theta_1[\sigma] \leq \theta_2$$

and finally,

$$\forall \widehat{\vartheta}, \vartheta_2.\ \widehat{\kappa} \wedge \kappa_2 \models \kappa_1[\sigma] \wedge \theta_1[\sigma] \leq \theta_2$$

which proves that τ_1 is a subtype of τ_2. ■

Proof of lemma 30

Let $\Delta = (\widehat{\vartheta} \colon \widehat{\kappa})$ be an implicit well-formed context, $\tau = \forall \vartheta \colon \kappa.\ \theta$ be a well-formed type w.r.t. Δ, ϑ' be a variable set disjoint from $\widehat{\vartheta}$, $\sigma \in \mathcal{R}(\widehat{\vartheta}; \vartheta; \vartheta')$, and $\widehat{v}$ be a fresh variable such that σ is also in $\mathcal{R}(\widehat{\vartheta}, \widehat{v}; \vartheta; \vartheta')$. Then σ is a bijection and σ^{-1} is a $(\widehat{\vartheta}, \widehat{v})$-substitution. By rule *Approx* we have

$$\forall \widehat{\vartheta}, \widehat{v}.\ \widehat{\kappa} \wedge \kappa \wedge \theta \leq \widehat{v} \models (\kappa \wedge \theta \leq \widehat{v})[\sigma^{-1} \circ \sigma]$$

and by rule *VIntro*

$$\forall \widehat{\vartheta}, \widehat{v}.\ \widehat{\kappa} \wedge \kappa \wedge \theta \leq \widehat{v} \models (\kappa \wedge \theta \leq \widehat{v})[\sigma]$$

that is

$$\forall \widehat{\vartheta}, \widehat{v}.\ \widehat{\kappa} \wedge \kappa \wedge \theta \leq \widehat{v} \models \kappa[\sigma] \wedge \theta[\sigma] \leq \widehat{v}$$

which shows that $\tau' = \forall \vartheta[\sigma] \colon \kappa[\sigma].\ \theta[\sigma]$ is a subtype of τ. Similarly, since by rule *VIntro*

$$\forall \widehat{\vartheta}, \widehat{v}.\ (\widehat{\kappa} \wedge \kappa \wedge \theta \leq \widehat{v})[\sigma] \models (\widehat{\kappa} \wedge \kappa \wedge \theta \leq \widehat{v})$$

we deduce that

$$\forall \widehat{\vartheta}, \widehat{v}.\ \widehat{\kappa} \wedge \kappa[\sigma] \wedge \theta[\sigma] \leq \widehat{v} \models (\widehat{\kappa} \wedge \kappa \wedge \theta \leq \widehat{v})$$

which shows that τ is a subtype of τ'. ■

Proof of lemma 31

Trivial consequence of theorem 28. ■

Proof of lemma 32

Let $\Delta = \widehat{\vartheta} \colon \widehat{\kappa}$ be an implicit well-formed context, θ_1, θ_2 be two ground monotypes and $\widehat{v}$ be a fresh type variable. If we assume that $\theta_1 \leq \theta_2$ for the standard ordering, then the proof tree for this fact can be readily translated into a derivation for the judgement

$$\forall \widehat{\vartheta}, \widehat{v}.\ \widehat{\kappa} \wedge \theta_2 \leq \widehat{v} \models \widehat{\kappa} \wedge \theta_2 \leq \widehat{v} \wedge \theta_1 \leq \theta_2$$

since rules *StdOrd* and *MIntro* are essentially the same. Using *Trans* and *Approx*, we can thus show that

$$\forall \widehat{\vartheta}, \widehat{v}.\ \widehat{\kappa} \wedge \theta_2 \leq \widehat{v} \models \theta_1 \leq \widehat{v}$$

which proves that $\forall \emptyset.\ \theta_1$ is a subtype of $\forall \emptyset.\ \theta_2$. ■

Proof of theorem 33

Let $\Delta = \widehat{\vartheta} \colon \widehat{\kappa}$ be an implicit well-formed context, $\tau = \forall\vartheta \colon \kappa.\ \theta$ and $\tau' = \forall\vartheta' \colon \kappa'.\ \theta'$ be two well-formed types and σ be a $\widehat{\vartheta}$-substitution such that $\kappa' = \kappa[\sigma]$ and $\theta' = \theta[\sigma]$. Let $\widehat{v}$ be a fresh variable. Since σ is also a $(\widehat{\vartheta}, \widehat{v})$-substitution, rule *VIntro* shows that

$$\forall\widehat{\vartheta}, \widehat{v}.\ (\widehat{\kappa} \wedge \kappa \wedge \theta \leq \widehat{v})[\sigma] \models \widehat{\kappa} \wedge \kappa \wedge \theta \leq \widehat{v}$$

but since σ is a $\widehat{\vartheta}$-substitution and $\widehat{\kappa}$ is $\widehat{\vartheta}$-closed, we have $\widehat{\kappa}[\sigma] = \widehat{\kappa}$, and thus

$$\forall\widehat{\vartheta}, \widehat{v}.\ \widehat{\kappa} \wedge \kappa[\sigma] \wedge \theta[\sigma] \leq \widehat{v} \models \widehat{\kappa} \wedge \kappa \wedge \theta \leq \widehat{v}$$

or else, by rule *Approx*

$$\forall\widehat{\vartheta}, \widehat{v}.\ \widehat{\kappa} \wedge \kappa[\sigma] \wedge \theta[\sigma] \leq \widehat{v} \models \kappa \wedge \theta \leq \widehat{v}$$

which proves that

$$\widehat{\vartheta} \colon \widehat{\kappa} \vdash (\forall\vartheta \colon \kappa.\ \theta) \leq (\forall\vartheta' \colon \kappa[\sigma].\ \theta[\sigma])$$

■

Proof of theorem 34

This is trivial consequence of theorem 22 and lemma 29. ■

Proof of theorem 35

Let $\Delta = \widehat{\vartheta} \colon \widehat{\kappa}$ be an implicit well-formed context, $\tau_i = \forall\vartheta_i \colon \kappa_i.\ \theta_i$, $i \in [1,3]$, be three well-formed types, and v and $\widehat{v}$ be two distinct and fresh type variables. We first remark that the subtyping relation is reflexive, since by rule *Approx*, we trivially have

$$\forall\widehat{v}, \widehat{\vartheta}.\ \widehat{\kappa} \wedge \kappa_i \wedge \theta_i \leq \widehat{v} \models \kappa_i \wedge \theta_i \leq \widehat{v}$$

To show that it is also transitive, let us assume that $\tau_1 \leq \tau_2$ and $\tau_2 \leq \tau_3$. For $i \in [1,2]$, we have

$$\forall\widehat{v}, \widehat{\vartheta}.\ \widehat{\kappa} \wedge \kappa_i \wedge \theta_i \leq \widehat{v} \models \kappa_{i+1} \wedge \theta_{i+1} \leq \widehat{v}$$

but since Δ is well-formed, we know that $\widehat{\kappa}$ is $(\widehat{v}, \widehat{\vartheta})$-closed, and by corollary 5

$$\forall\widehat{v}, \widehat{\vartheta}.\ \widehat{\kappa} \wedge \kappa_i \wedge \theta_i \leq \widehat{v} \models \widehat{\kappa} \wedge \kappa_{i+1} \wedge \theta_{i+1} \leq \widehat{v}$$

and thus by rules *Trans* and *Approx*, we have $\tau_1 \leq \tau_3$.

The set of well-formed types modulo equivalence is thus a partial order. Let us now show that it is also a sup-semi-lattice. So let us assume that τ_1 and τ_2 have a common well-formed supertype $\tau = \forall\vartheta \colon \kappa.\ \theta$. Without loss of generality, we assume that $\vartheta_1 \# \vartheta_2$. For $i \in [1,2]$, we have

$$\forall\widehat{v}, \widehat{\vartheta}.\ \widehat{\kappa} \wedge \kappa \wedge \theta \leq \widehat{v} \models \kappa_i \wedge \theta_i \leq \widehat{v}$$

but since by hypothesis

$$(\kappa_1 \wedge \theta_1 \leq \widehat{v}) \# (\kappa_2 \wedge \theta_2 \leq \widehat{v})\ \ [\widehat{v}, \widehat{\vartheta}]$$

we have by corollary 4

$$\forall \widehat{v}, \widehat{\vartheta}.\ \widehat{\kappa} \wedge \kappa \wedge \theta \leq \widehat{v} \models (\kappa_1 \wedge \theta_1 \leq \widehat{v}) \wedge (\kappa_2 \wedge \theta_2 \leq \widehat{v})$$

and therefore, by rule *MRef*

$$\forall \widehat{v}, \widehat{\vartheta}.\ \widehat{\kappa} \wedge \kappa \wedge \theta \leq \widehat{v} \models (\kappa_1 \wedge \theta_1 \leq \widehat{v} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v}) \wedge \widehat{v} \leq \widehat{v}$$

and by rule *VIntro* with $\sigma = \{v \mapsto \widehat{v}\}$

$$\forall \widehat{v}, \widehat{\vartheta}.\ \widehat{\kappa} \wedge \kappa \wedge \theta \leq \widehat{v} \models (\kappa_1 \wedge \theta_1 \leq v \wedge \kappa_2 \wedge \theta_2 \leq v) \wedge v \leq \widehat{v}$$

which proves that

$$\forall v, \vartheta_1, \vartheta_2 \colon (\kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq v \wedge \theta_2 \leq v).\ v$$

is a subtype of τ provided that it is well-formed. But, by lemma 1 and rule *Approx*, the above implication implies that

$$\forall \widehat{\vartheta}.\ \widehat{\kappa} \wedge \kappa \wedge \theta \leq \widehat{v} \models \kappa_1 \wedge \theta_1 \leq v \wedge \kappa_2 \wedge \theta_2 \leq v$$

and since

$$\begin{array}{lll}
\forall \widehat{\vartheta}. & \widehat{\kappa} & (\tau \ \textit{well-formed}) \\
\models & \widehat{\kappa} \wedge \kappa & (\textit{MRef}) \\
\models & \widehat{\kappa} \wedge \kappa \wedge \theta \leq \theta & (\textit{VIntro with } \sigma = \{\widehat{v} \mapsto \theta\}) \\
\models & \widehat{\kappa} \wedge \kappa \wedge \theta \leq \widehat{v} &
\end{array}$$

we thus have

$$\forall \widehat{\vartheta}.\ \widehat{\kappa} \models \kappa_1 \wedge \theta_1 \leq v \wedge \kappa_2 \wedge \theta_2 \leq v$$

which proves that

$$\forall v, \vartheta_1, \vartheta_2 \colon (\kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq v \wedge \theta_2 \leq v).\ v$$

is well-formed. Consequently, every upper bound of τ_1 and τ_2 is an upper bound of this type, but conversely, by a trivial application of rules *Approx* and *Trans*, we have

$$\forall \widehat{v}, \widehat{\vartheta}.\ \widehat{\kappa} \wedge (\kappa_1 \wedge \theta_1 \leq v \wedge \kappa_2 \wedge \theta_2 \leq v) \wedge v \leq \widehat{v} \models \kappa_i \wedge \theta_i \leq \widehat{v}$$

for $i \in [1, 2]$, and thus

$$\forall v, \vartheta_1, \vartheta_2 \colon \widehat{\kappa} \wedge (\kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq v \wedge \theta_2 \leq v).\ v$$

is an upper bound of both τ_1 and τ_2, which shows that it is the least upper bound. ∎

Proof of theorem 36

Let $\Delta = \widehat{\vartheta} \colon \widehat{\kappa}$ be an implicit well-formed context, $\tau = \forall \vartheta \colon \kappa.\ \theta$ and $\tau' = \forall \vartheta' \colon \kappa'.\ \theta'$ be two well-formed types such that $\tau \leq \tau'$ and $\vartheta \# \vartheta'$, and v_1, v_2 be two distinct and fresh type variables. Let

$$\widehat{\tau} \quad = \quad \forall v_1, v_2, \vartheta \colon \kappa \wedge \theta \leq (v_1 \rightarrow v_2).\ (v_1 \rightarrow v_2)$$

and

$$\widehat{\tau}' \quad = \quad \forall v_1', v_2', \vartheta' \colon \kappa' \wedge \theta' \leq (v_1 \rightarrow v_2).\ (v_1 \rightarrow v_2)$$

We assume that $\widehat{\tau}'$ is well-formed and we want to show that $\widehat{\tau}$ is well-formed and is a subtype of $\widehat{\tau}'$. So let $\widehat{v}$ be a fresh type variable. We have

$$\begin{array}{lll}
\forall \widehat{v}, \widehat{\vartheta}. & \widehat{\kappa} \wedge \kappa' \wedge \theta' \leq (v_1 \to v_2) \leq \widehat{v} & (\mathit{MRef}) \\
\models & \widehat{\kappa} \wedge \kappa' \wedge \theta' \leq (v_1 \to v_2) \leq (v_1 \to v_2) \leq \widehat{v} & (\mathit{VIntro\ with}\ \sigma = \{v \mapsto (v_1 \to v_2)\}) \\
\models & (\widehat{\kappa} \wedge \kappa' \wedge \theta' \leq v) \wedge (v \leq v_1 \to v_2 \leq \widehat{v}) &
\end{array}$$

but since

$$\begin{cases} \forall v, \widehat{v}, \widehat{\vartheta}.\ \widehat{\kappa} \wedge \kappa' \wedge \theta' \leq v \models \kappa \wedge \theta \leq v \\ \forall v, \widehat{v}, \widehat{\vartheta}.\ v \leq v_1 \to v_2 \leq \widehat{v} \models v \leq v_1 \to v_2 \leq \widehat{v} \end{cases}$$

and

$$(\kappa \wedge \theta \leq v) \not\# (v \leq v_1 \to v_2 \leq \widehat{v})\ \ [v, \widehat{v}, \widehat{\vartheta}]$$

lemma 3 implies that

$$\forall v, \widehat{v}, \widehat{\vartheta}.\ (\widehat{\kappa} \wedge \kappa' \wedge \theta' \leq v) \wedge (v \leq v_1 \to v_2 \leq \widehat{v}) \models \kappa \wedge \theta \leq v \leq v_1 \to v_2 \leq \widehat{v}$$

and lemma 1 shows that

$$\forall \widehat{v}, \widehat{\vartheta}.\ (\widehat{\kappa} \wedge \kappa' \wedge \theta' \leq v) \wedge (v \leq v_1 \to v_2 \leq \widehat{v}) \models \kappa \wedge \theta \leq v_1 \to v_2 \leq \widehat{v}$$

which finally shows that

$$\forall \widehat{v}, \widehat{\vartheta}.\ \widehat{\kappa} \wedge \kappa' \wedge \theta' \leq v_1 \to v_2 \leq \widehat{v} \models \kappa \wedge \theta \leq v_1 \to v_2 \leq \widehat{v}$$

which proves that $\widehat{\tau}$ is a subtype of τ' provided that it is also well-formed. But lemma 1 and rule *Approx* imply that

$$\forall \widehat{\vartheta}.\ \widehat{\kappa} \wedge \kappa' \wedge \theta' \leq v_1 \to v_2 \leq \widehat{v} \models \kappa \wedge \theta \leq v_1 \to v_2$$

and by rules *MRef*, *VIntro* and the fact that $\widehat{\tau}'$ is well-formed, it is easy to show that

$$\forall \widehat{\vartheta}.\ \widehat{\kappa} \models \widehat{\kappa} \wedge \kappa' \wedge \theta' \leq v_1 \to v_2 \leq \widehat{v}$$

which finally shows that

$$\forall \widehat{\vartheta}.\ \widehat{\kappa} \models \kappa \wedge \theta \leq v_1 \to v_2$$

and thus that $\widehat{\tau}$ is well-formed. ■

Proof of theorem 40

Let $\Delta = \widehat{\vartheta} \colon \widehat{\kappa}$ be an implicit well-formed context and v_1, v_2, v_1', v_2', v and v' be six distinct and fresh type variables. Let $\tau_i = \forall \vartheta_i \colon \kappa_i.\ \theta_i$ ($i \in [1,2]$) be two types such that $\tau_1 \leq \tau_2$ and τ_2 is prefunctional, that is, $\Delta \vdash \mathit{fun}_\Delta(\tau_2)$. Then by theorem 36, τ_1 is also prefunctional. Consequently the domains of τ_1 and τ_2 defined by

$$\mathit{dom}_\Delta(\tau_i) \ = \ \forall v_i, v_i', \vartheta_i \colon \kappa_i \wedge \theta_i \leq v_i \to v_i'.\ v_i$$

are well-formed. Now, since τ_1 is a subtype of τ_2 by hypothesis, we have

$$\forall \widehat{\vartheta}, v'.\ \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq v' \models \kappa_1 \wedge \theta_1 \leq v'$$

but

$$
\begin{array}{lll}
\forall\widehat{\vartheta}, v. & \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq v_2 \rightarrow v_2' \wedge v \leq v_2 & (\mathit{MRef}) \\
\models & \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq v_2 \rightarrow v_2' \wedge v \leq v_2 \wedge v_2' \leq v_2' & (\mathit{MIntro}) \\
\models & \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq v_2 \rightarrow v_2' \wedge v_2 \rightarrow v_2' \leq v \rightarrow v_2' & (\mathit{Trans}) \\
\models & \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq v \rightarrow v_2' & (\mathit{MRef}) \\
\models & (\widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq v \rightarrow v_2') \wedge (v \rightarrow v_2' \leq v \rightarrow v_2') & (\mathit{VIntro\ with}\ \sigma = \{v' \mapsto (v \rightarrow v_2')\}) \\
\models & (\widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq v') \wedge (v' \leq v \rightarrow v_2') &
\end{array}
$$

and thus, since

$$(\kappa_1 \wedge \theta_1 \leq v') \mathrel{\#} (v' \leq v \rightarrow v_2')\ [\widehat{\vartheta}, v, v', v_2']$$

we conclude by lemma 3 and the fact that $\tau_1 \leq \tau_2$ that

$$
\begin{array}{ll}
\forall\widehat{\vartheta}, v, v', v_2'. & (\widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq v') \wedge (v' \leq v \rightarrow v_2') \\
\models & (\kappa_1 \wedge \theta_1 \leq v') \wedge (v' \leq v \rightarrow v_2')
\end{array}
$$

and by lemma 1

$$
\begin{array}{ll}
\forall\widehat{\vartheta}, v. & (\widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq v') \wedge (v' \leq v \rightarrow v_2') \\
\models & (\kappa_1 \wedge \theta_1 \leq v') \wedge (v' \leq v \rightarrow v_2')
\end{array}
$$

and finally

$$
\begin{array}{lll}
\forall\widehat{\vartheta}, v. & \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq v_2 \rightarrow v_2' \wedge v \leq v_2 & \\
\models & (\kappa_1 \wedge \theta_1 \leq v') \wedge (v' \leq v \rightarrow v_2') & (\mathit{Trans}) \\
\models & \kappa_1 \wedge \theta_1 \leq v \rightarrow v_2' & (\mathit{MRef}) \\
\models & \kappa_1 \wedge \theta_1 \leq v \rightarrow v_2' \wedge v \leq v & (\mathit{VIntro\ with}\ \sigma = \{v_1 \mapsto v, v_1' \mapsto v_2'\}) \\
\models & \kappa_1 \wedge \theta_1 \leq v_1 \rightarrow v_1' \wedge v \leq v_1 &
\end{array}
$$

which shows that $dom_\Delta(\tau_2)$ is a subdomain of $dom_\Delta(\tau_1)$. ■

Proof of lemma 38

Similar to the proof of lemma 32. ■

Proof of theorem 39

Let $\Delta = \widehat{\vartheta} : \widehat{\kappa}$ be an implicit well-formed context, and for any i in the range $[1, 3]$, let $\tau_i = \forall\vartheta_i : \kappa_i.\ \theta_i$ and $\delta_i = \exists\vartheta_i : \kappa_i.\ \theta_i$. Without loss of generality, we may assume that $\vartheta_1 \mathrel{\#} \vartheta_2 \mathrel{\#} \vartheta_3$. Let $\widehat{v}$ be a fresh type variable. Let us first assume that $\tau_1 \leq \tau_2$ and $\tau_2 \in \delta_3$. We have

$$\forall\widehat{\vartheta}.\ \widehat{\kappa} \models \kappa_2 \wedge \kappa_3 \wedge \theta_2 \leq \theta_3$$

and thus, by corollary 5

$$
\begin{array}{lll}
\forall\widehat{\vartheta}. & \widehat{\kappa} & \\
\models & \widehat{\kappa} \wedge \kappa_2 \wedge \kappa_3 \wedge \theta_2 \leq \theta_3 & (\mathit{MRef}) \\
\models & (\widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \theta_2) \wedge (\kappa_3 \wedge \theta_2 \leq \theta_3) & (\mathit{VIntro\ with}\ \sigma = \{\widehat{v} \mapsto \theta_2\}) \\
\models & (\widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v}) \wedge (\kappa_3 \wedge \widehat{v} \leq \theta_3) &
\end{array}
$$

but since by hypothesis

$$\forall \widehat{v}, \widehat{\vartheta}.\ \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v} \models \kappa_1 \wedge \theta_1 \leq \widehat{v}$$

and

$$(\kappa_1 \wedge \theta_1 \leq \widehat{v}) \not\# (\kappa_3 \wedge \widehat{v} \leq \theta_3)\ [\widehat{v}, \widehat{\vartheta}]$$

we have by lemma 3

$$\forall \widehat{v}, \widehat{\vartheta}.\ (\widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v}) \wedge (\kappa_3 \wedge \widehat{v} \leq \theta_3) \models (\kappa_1 \wedge \theta_1 \leq \widehat{v}) \wedge (\kappa_3 \wedge \widehat{v} \leq \theta_3)$$

and by lemma 1

$$\forall \widehat{\vartheta}.\ (\widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \leq \widehat{v}) \wedge (\kappa_3 \wedge \widehat{v} \leq \theta_3) \models (\kappa_1 \wedge \theta_1 \leq \widehat{v}) \wedge (\kappa_3 \wedge \widehat{v} \leq \theta_3)$$

which shows that

$$\begin{array}{ll} \forall \widehat{\vartheta}. & \widehat{\kappa} \\ \models & (\kappa_1 \wedge \theta_1 \leq \widehat{v}) \wedge (\kappa_3 \wedge \widehat{v} \leq \theta_3) \quad (\mathit{Trans}) \\ \models & \kappa_1 \wedge \kappa_3 \wedge \theta_1 \leq \theta_3 \end{array}$$

and finally $\tau_1 \in \delta_3$. Let us now assume that $\tau_1 \in \delta_2$ and $\delta_2 \leq \delta_3$. We have

$$\forall \widehat{\vartheta}.\ \widehat{\kappa} \models \kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq \theta_2$$

and thus, by corollary 5

$$\begin{array}{lll} \forall \widehat{\vartheta}. & \widehat{\kappa} & \\ \models & \widehat{\kappa} \wedge \kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq \theta_2 & (\mathit{MRef}) \\ \models & (\kappa_1 \wedge \theta_1 \leq \theta_1) \wedge (\widehat{\kappa} \wedge \kappa_2 \wedge \theta_1 \leq \theta_2) & (\mathit{VIntro\ with}\ \sigma = \{\widehat{v} \mapsto \theta_2\}) \\ \models & (\kappa_1 \wedge \theta_1 \leq \widehat{v}) \wedge (\widehat{\kappa} \wedge \kappa_2 \wedge \widehat{v} \leq \theta_2) & \end{array}$$

but since by hypothesis

$$\forall \widehat{v}, \widehat{\vartheta}.\ \widehat{\kappa} \wedge \kappa_2 \wedge \widehat{v} \leq \theta_2 \models \kappa_3 \wedge \widehat{v} \leq \theta_3$$

and

$$(\kappa_1 \wedge \theta_1 \leq \widehat{v}) \not\# (\kappa_3 \wedge \widehat{v} \leq \theta_3)\ [\widehat{v}, \widehat{\vartheta}]$$

we have by lemma 3

$$\forall \widehat{v}, \widehat{\vartheta}.\ (\kappa_1 \wedge \theta_1 \leq \widehat{v}) \wedge (\widehat{\kappa} \wedge \kappa_2 \wedge \widehat{v} \leq \theta_2) \models (\kappa_1 \wedge \theta_1 \leq \widehat{v}) \wedge (\kappa_3 \wedge \widehat{v} \leq \theta_3)$$

and by lemma 1

$$\forall \widehat{\vartheta}.\ (\kappa_1 \wedge \theta_1 \leq \widehat{v}) \wedge (\widehat{\kappa} \wedge \kappa_2 \wedge \widehat{v} \leq \theta_2) \models (\kappa_1 \wedge \theta_1 \leq \widehat{v}) \wedge (\kappa_3 \wedge \widehat{v} \leq \theta_3)$$

which shows that

$$
\begin{array}{ll}
\forall \widehat{\vartheta}. & \widehat{\kappa} \\
\models & (\kappa_1 \wedge \theta_1 \leq \widehat{v}) \wedge (\kappa_3 \wedge \widehat{v} \leq \theta_3) \quad (\mathit{Trans}) \\
\models & \kappa_1 \wedge \kappa_3 \wedge \theta_1 \leq \theta_3
\end{array}
$$

and finally $\tau_1 \in \delta_3$. ■

Proof of lemma 37

Trivial consequence of theorem 28. ■

Proof of theorem 41

Let $\Delta = \widehat{\vartheta} \colon \widehat{\kappa}$ be an implicit well-formed context, and for $\imath \in [1, 2]$, τ_i and τ_i' be well-formed types. Assuming that $\tau_1 \leq \tau_2$, $\tau_1' \leq \tau_2'$ and τ_2' belongs to the domain of τ_2, let us show that

$$
\mathit{app}_\Delta(\tau_1, \tau_1') \quad \leq \quad \mathit{app}_\Delta(\tau_2, \tau_2')
$$

We first remark that the hypothesis $\tau_2' \in \mathit{dom}_\Delta(\tau_2)$ implies that $\mathit{dom}_\Delta(\tau_2)$ is well-formed, which implies that $\mathit{fun}_\Delta(\tau_2)$ is well-formed, which, together with the hypothesis and theorem 40, implies that $\mathit{dom}_\Delta(\tau_2)$ is a subdomain of $\mathit{dom}_\Delta(\tau_1)$. Consequently, theorem 39 implies that $\tau_1' \in \mathit{dom}_\Delta(\tau_1)$ and $\mathit{app}_\Delta(\tau_1, \tau_1')$ and $\mathit{app}_\Delta(\tau_2, \tau_2')$ are thus well-formed. So let us assume that $\mathit{fun}_\Delta(\tau_i)$ is of the form

$$
\mathit{fun}_\Delta(\tau_i) = \forall \vartheta_i \colon \kappa_i.\ \theta_i \to \theta_i''
$$

and τ_i' is of the form

$$
\tau_i' = \forall \vartheta_i' \colon \kappa_i'.\ \theta_i'
$$

with $\vartheta_1 \mathrel{\#} \vartheta_2 \mathrel{\#} \vartheta_1' \mathrel{\#} \vartheta_2' \mathrel{\#} \widehat{\vartheta}$, and v and v' be two distinct and fresh type variables. Since fun_Δ is covariant, we have $\mathit{fun}_\Delta(\tau_1) \leq \mathit{fun}_\Delta(\tau_2)$, that is

$$
\forall v', \widehat{\vartheta}.\ \widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \to \theta_2'' \leq v' \models \kappa_1 \wedge \theta_1 \to \theta_1'' \leq v'
$$

Now,

$$
\begin{array}{lll}
\forall \widehat{\vartheta}, v. & \widehat{\kappa} \wedge \kappa_2 \wedge \kappa_2' \wedge \theta_2' \leq \theta_2 \wedge \theta_2'' \leq v & (\mathit{MRef}) \\
\models & (\kappa_2 \wedge \theta_2 \leq \theta_2 \wedge \theta_2'' \leq v) \wedge (\widehat{\kappa} \wedge \kappa_2' \wedge \theta_2' \leq \theta_2) & (\mathit{VIntro\ with}\ \sigma = \{v' \mapsto \theta_2\}) \\
\models & (\kappa_2 \wedge v' \leq \theta_2 \wedge \theta_2'' \leq v) \wedge (\widehat{\kappa} \wedge \kappa_2' \wedge \theta_2' \leq v') &
\end{array}
$$

But by that fact that $\tau_1' \leq \tau_2'$, by corollary 5 and by the fact that v is fresh

$$
\begin{array}{ll}
\forall \widehat{\vartheta}, v, v'. & \widehat{\kappa} \wedge \kappa_2' \wedge \theta_2' \leq v' \\
\models & \widehat{\kappa} \wedge \kappa_1' \wedge \theta_1' \leq v'
\end{array}
$$

and thus, since

$$
(\kappa_2 \wedge v' \leq \theta_2 \wedge \theta_2'' \leq v) \mathrel{\#} (\widehat{\kappa} \wedge \kappa_1' \wedge \theta_1' \leq v') \ [\widehat{\vartheta}, v, v']
$$

we have, thanks to lemma 3

$$
\begin{array}{lll}
\forall \widehat{\vartheta}, v, v'. & (\kappa_2 \wedge v' \leq \theta_2 \wedge \theta_2'' \leq v) \wedge (\widehat{\kappa} \wedge \kappa_2' \wedge \theta_2' \leq v') & \\
\models & (\kappa_2 \wedge v' \leq \theta_2 \wedge \theta_2'' \leq v) \wedge (\widehat{\kappa} \wedge \kappa_1' \wedge \theta_1' \leq v') & (\mathit{Trans}) \\
\models & \widehat{\kappa} \wedge \kappa_2 \wedge \kappa_1' \wedge \theta_1' \leq \theta_2 \wedge \theta_2'' \leq v &
\end{array}
$$

and by lemma 1

$$
\begin{array}{lll}
\forall \widehat{\vartheta}, v. & \widehat{\kappa} \wedge \kappa_2 \wedge \kappa_2' \wedge \theta_2' \leq \theta_2 \wedge \theta_2'' \leq v & \\
\models & \widehat{\kappa} \wedge \kappa_2 \wedge \kappa_1' \wedge \theta_1' \leq \theta_2 \wedge \theta_2'' \leq v & (\mathit{MIntro}) \\
\models & \widehat{\kappa} \wedge \kappa_2 \wedge \kappa_1' \wedge \theta_2 \to \theta_2'' \leq \theta_1' \to v & (\mathit{MRef}) \\
\models & (\kappa_1' \wedge \theta_2 \to \theta_2'' \leq \theta_1' \to v) \wedge & (\mathit{VIntro\ with} \\
 & (\widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \to \theta_2'' \leq \theta_2 \to \theta_2'') & \quad \sigma = \{v' \mapsto (\theta_2 \to \theta_2'')\}) \\
\models & (\kappa_1' \wedge v' \leq \theta_1' \to v) \wedge (\widehat{\kappa} \wedge \kappa_2 \wedge \theta_2 \to \theta_2'' \leq v') & (\mathit{fun}_\Delta(\tau_1) \leq \mathit{fun}_\Delta(\tau_2), \mathit{lemma\ 3}) \\
\models & (\kappa_1' \wedge v' \leq \theta_1' \to v) \wedge (\kappa_1 \wedge \theta_1 \to \theta_1'' \leq v') & (\mathit{Trans}, \mathit{MElim}) \\
\models & \kappa_1 \wedge \kappa_1' \wedge \theta_1' \leq \theta_1 \wedge \theta_1'' \leq v &
\end{array}
$$

which proves that

$$
\mathit{app}_\Delta(\tau_1, \tau_1') \quad \leq \quad \mathit{app}_\Delta(\tau_2, \tau_2')
$$

■

Proof of theorem 42

Let $\Delta = (\widehat{\vartheta} \colon \widehat{\kappa})$ be a well-formed context, ϑ_C be a fresh C-variable set and $(\forall \vartheta_i \colon \kappa_i.\ \theta_i)$, $i \in [1, n]$, be n well-formed types such that $\vartheta_1 \# \ldots \# \vartheta_n$ and $\mathit{app}_\Delta(d_C, \tau_1, \ldots, \tau_n)$ is well formed, that is

$$
\forall \vartheta_1, \ldots, \vartheta_n, \vartheta_C \colon \kappa_1 \wedge \ldots \wedge \kappa_n \wedge \theta_1 \leq d_C^1 \langle \vartheta_C \rangle \wedge \ldots \wedge \theta_n \leq d_C^n \langle \vartheta_C \rangle.\ d_C[\vartheta_C]
$$

is well-formed w.r.t. Δ. For any $i \in [1, n]$ and any fresh C-variable set ϑ_C', the type $\mathit{app}_\Delta(d_C^i, \mathit{app}_\Delta(d_C, \tau_1, \ldots, \tau_n))$ is defined by

$$
\begin{array}{ll}
\forall & (\vartheta_1, \ldots, \vartheta_n, \vartheta_C, \vartheta_C'). \\
 & (\kappa_1 \wedge \ldots \wedge \kappa_n) \wedge \\
 & (\theta_1 \leq d_C^1 \langle \vartheta_C \rangle) \wedge \ldots \wedge (\theta_n \leq d_C^n \langle \vartheta_C \rangle) \wedge \\
 & d_C[\vartheta_C] \leq d_C[\vartheta_C'] \\
. & d_C^i \langle \vartheta_C' \rangle
\end{array}
$$

which is a supertype of

$$
\forall \vartheta_1, \ldots, \vartheta_n, \vartheta_C, \vartheta_C' \colon \kappa_i \wedge \theta_i \leq d_C^i \langle \vartheta_C \rangle \wedge d_C[\vartheta_C] \leq d_C[\vartheta_C'].\ d_C^i \langle \vartheta_C' \rangle
$$

which, in turn, is a supertype of $(\forall \vartheta_i \colon \kappa_i.\ \theta_i)$ since, for any fresh variable $\widehat{v}$,

$$
\begin{array}{lll}
\multicolumn{3}{l}{\forall \widehat{v}, \vartheta_C, \vartheta_C', \widehat{\vartheta}.} \\
 & \widehat{\kappa} \wedge \kappa_i \wedge \theta_i \leq d_C^i \langle \vartheta_C \rangle \wedge d_C[\vartheta_C] \leq d_C[\vartheta_C'] \wedge d_C^i \langle \vartheta_C' \rangle \leq \widehat{v} & (\mathit{MElim}) \\
\models & \kappa_i \wedge \theta_i \leq d_C^i \langle \vartheta_C \rangle \wedge \vartheta_C \leq_C \vartheta_C' \wedge d_C^i \langle \vartheta_C' \rangle \leq \widehat{v} & (\mathit{Hypothesis,\ lemma\ 3}) \\
\models & \kappa_i \wedge \theta_i \leq d_C^i \langle \vartheta_C \rangle \wedge d_C^i \langle \vartheta_C \rangle \leq d_C^i \langle \vartheta_C' \rangle \wedge d_C^i \langle \vartheta_C' \rangle \leq \widehat{v} & (\mathit{Trans}) \\
\models & \kappa_i \wedge \theta_i \leq \widehat{v} &
\end{array}
$$

■

Proof of theorem 43

Let $\Delta = \widehat{\vartheta}\colon \widehat{\kappa}$ be an implicit well-formed context, $\tau = \forall\vartheta\colon \kappa.\ \theta_1 \to \theta_2$ be a functional type, $\delta = \exists\vartheta\colon \kappa.\ \theta_1$ be the domain of τ, $\pi' = \exists\vartheta'.\ \theta'$ be a pattern which is strongly compatible with δ and $\tau'' = \forall\vartheta''\colon \kappa''.\ d''_C[\Theta''_C]$ be a run-time type both in δ and π'. As usual, we assume that $\vartheta \# \vartheta' \# \vartheta'' \# \widehat{\vartheta}$. We have

$$\begin{cases} app(res(\tau,\pi'),\tau'') & = & \forall\vartheta,\vartheta',\vartheta''\colon \kappa \wedge \kappa'' \wedge \theta'' \leq \theta_1 \wedge \theta'' \leq \theta'.\ \theta_2 \\ app(\tau,\tau'') & = & \forall\vartheta,\vartheta''\colon \kappa \wedge \kappa'' \wedge \theta'' \leq \theta_1.\ \theta_2 \end{cases}$$

and it is thus easy to see using rule *Approx* that

$$app(\tau,\tau'') \ \leq\ app(res(\tau,\pi'),\tau'')$$

so let us prove the other inequality. Let $\widehat{v}$ be a fresh type variable. There are two cases. If θ' is a simple variable v', then obviously

$$\begin{array}{lll} \forall\widehat{v},\widehat{\vartheta}. & \widehat{\kappa} \wedge \kappa \wedge \kappa'' \wedge \theta'' \leq \theta_1 \wedge \theta_2 \leq \widehat{v} & (\mathit{MRef}) \\ \models & \kappa \wedge \kappa'' \wedge \theta'' \leq \theta_1 \wedge \theta'' \leq \theta'' \wedge \theta_2 \leq \widehat{v} & (\mathit{VIntro\ with}\ \sigma = \{v' \mapsto \theta''\}) \\ \models & \kappa \wedge \kappa'' \wedge \theta'' \leq \theta_1 \wedge \theta'' \leq v' \wedge \theta_2 \leq \widehat{v} & (\theta' = v') \\ \models & \kappa \wedge \kappa'' \wedge \theta'' \leq \theta_1 \wedge \theta'' \leq \theta' \wedge \theta_2 \leq \widehat{v} & \end{array}$$

which proves the theorem. Now, if θ' is of the form $t'_C[\vartheta'_C]$, then since τ'' belongs to π' by hypothesis, it is easy to see that $d''_C \sqsubseteq_C t'_C$. Consequently

$$\begin{array}{lll} \forall\widehat{v},\widehat{\vartheta}. & \widehat{\kappa} \wedge \kappa \wedge \kappa'' \wedge \theta'' \leq \theta_1 \wedge \theta_2 \leq \widehat{v} & (\theta'' = d''_C[\Theta''_C], \mathit{MIntro}) \\ \models & \kappa \wedge \kappa'' \wedge \theta'' \leq \theta_1 \wedge \theta'' \leq t'_C[\Theta''_C] \wedge \theta_2 \leq \widehat{v} & (\mathit{VIntro\ with}\ \sigma = \{\vartheta'_C \mapsto \Theta''_C\}) \\ \models & \kappa \wedge \kappa'' \wedge \theta'' \leq \theta_1 \wedge \theta'' \leq t'_C[\vartheta'_C] \wedge \theta_2 \leq \widehat{v} & (\theta' = t'_C[\vartheta'_C]) \\ \models & \kappa \wedge \kappa'' \wedge \theta'' \leq \theta_1 \wedge \theta'' \leq \theta' \wedge \theta_2 \leq \widehat{v} & \end{array}$$

which proves that

$$app(res(\tau,\pi'),\tau'') \ \leq\ app(\tau,\tau'')$$

since obviously

$$\begin{array}{lll} \forall\widehat{\vartheta}. & \widehat{\kappa} & (\tau'' \in \delta) \\ \models & \widehat{\kappa} \wedge \kappa \wedge \kappa'' \wedge \theta'' \leq \theta_1 & (\mathit{MRef}, \mathit{VIntro}) \\ \models & \widehat{\kappa} \wedge \kappa \wedge \kappa'' \wedge \theta'' \leq \theta_1 \wedge \theta_2 \leq \widehat{v} & (\mathit{Above\ proof}) \\ \models & \kappa \wedge \kappa'' \wedge \theta'' \leq \theta_1 \wedge \theta'' \leq \theta' & \end{array}$$

which shows that $app(res(\tau,\pi'),\tau'')$ is well-formed and that τ'' belongs to $\delta \wedge \pi'$. ■

Proof of theorem 45

Let Δ be a well-formed context, δ' be a well-formed domain w.r.t. Δ, τ be a closed run-time type in $\delta'[\Delta]$, and $\pi_1,\ldots,\pi_n$, $n \geq 1$, be a partition of δ' w.r.t. Δ. We assume that Δ is of the form $(\widehat{\vartheta}\colon \widehat{\kappa})$, δ' is of the form $\exists\vartheta'\colon \kappa'.\ \theta'$, τ is of the form $\forall\vartheta\colon \kappa.\ d_C[\Theta_C]$, and π_i is of the form $\exists\vartheta_i.\ \theta_i$. Without loss of generality, we assume that $\vartheta, \vartheta', \vartheta_1, \ldots, \vartheta_n$ are disjoint pairwise. From the fact that τ belongs to $\delta'[\Delta]$, we have

$$\forall\emptyset.\ \mathit{true} \models \widehat{\kappa} \wedge \kappa \wedge \kappa' \wedge d_C[\Theta_C] \leq \theta'$$

which shows, by rules *MRef* and *VIntro*, that

$$\forall\emptyset.\ \mathit{true} \models \widehat{\kappa} \wedge \kappa' \wedge v \leq \theta' \wedge v \leq d_C[\vartheta_C]$$

for some fresh type variable v, which shows that $\delta'[\Delta]$ and $\delta'' = \exists\vartheta_C.\ d_C[\vartheta_C]$ are compatible. By condition (3) of definition 44, we deduce the existence of an index $i \in [1, n]$ such that δ'' is a subdomain of π_i. But (1) since τ' trivially belongs to δ'', then τ' also belongs to π_i, and (2) since δ'' is a subdomain of π_i and $\delta'[\Delta]$ are compatible, then $\delta'[\Delta]$ and π_i are compatible. Consequently, theorem 43 shows that τ' belongs to $\delta'[\Delta] \wedge \pi_i$. By the same reasoning, we can show that for every $j \in [1, n]$ such that τ belongs to $\delta'[\Delta] \wedge \pi_j$, δ'' is compatible with $\delta'[\Delta] \wedge \pi_j$, so that δ'' is compatible with π_j. Since π_j is a pattern and d_C is minimal, it is easy to see that this compatibility implies that δ'' is a subdomain of π_j, which shows that π_i is a subdomain of π_j. ■

Proof of lemma 46

Let $(\Delta; \Gamma)$ be a well-formed typing context, $\Delta = \widehat{\vartheta}\colon \widehat{\kappa}$, and $\Delta' = \vartheta'\colon \kappa'$ be a context such that Δ' is well-formed w.r.t. Δ. By rule *Approx*, we have

$$\forall\widehat{\vartheta}, \vartheta'.\ \widehat{\kappa} \wedge \kappa' \models \widehat{\kappa}$$

So let $(x\colon \tau)$, with $\tau = \forall\vartheta\colon \kappa.\ \theta$, be a subterm of Γ. By definition, τ is well-formed w.r.t. Δ. Consequently

$$\forall\widehat{\vartheta}.\ \widehat{\kappa} \models \kappa$$

and since the variables in ϑ' do not occur in $\widehat{\vartheta}$, $\widehat{\kappa}$ and κ, we obviously have

$$\forall\vartheta', \widehat{\vartheta}.\ \widehat{\kappa} \models \kappa$$

and by transitivity

$$\forall\widehat{\vartheta}, \vartheta'.\ \widehat{\kappa} \wedge \kappa' \models \kappa$$

which shows that τ is well-formed w.r.t. $\Delta[\Delta']$, which is well-formed. Consequently, $\Delta[\Delta']; \Gamma$ is a well-formed typing context. So let $(\Delta; \bar{\Gamma})$ be a subcontext $(\Delta; \Gamma)$. Let $\bar{\tau} = \forall\bar{\vartheta}\colon \bar{\kappa}.\ \bar{\theta}$ denote the type of an expression variable x in $\bar{\Gamma}$ and τ denote the type of x in Γ. By definition, $\bar{\tau}$ is a subtype of τ w.r.t. Δ, that is,

$$\forall\widehat{v}, \widehat{\vartheta}.\ \widehat{\kappa} \wedge \kappa \wedge \theta \leq \widehat{v} \models \bar{\kappa} \wedge \bar{\theta} \leq \widehat{v}$$

for any fresh type variable $\widehat{v}$, and thus

$$\forall\widehat{v}, \widehat{\vartheta}, \vartheta'.\ \widehat{\kappa} \wedge \kappa \wedge \theta \leq \widehat{v} \models \bar{\kappa} \wedge \bar{\theta} \leq \widehat{v}$$

and by rule *Approx*

$$\forall\widehat{v}, \widehat{\vartheta}, \vartheta'.\ (\widehat{\kappa} \wedge \kappa') \wedge \kappa \wedge \theta \leq \widehat{v} \models \bar{\kappa} \wedge \bar{\theta} \leq \widehat{v}$$

which shows that $\bar{\tau}$ is a subtype of τ w.r.t. $\Delta[\Delta']$, which finally proves that $\Delta[\Delta']; \Gamma'$ is a subcontext of $\Delta[\Delta']; \Gamma$. ■

Proof of lemma 47

Let $(\widehat{\Delta};\widehat{\Gamma})$, $\widehat{\Delta} = (\widehat{\vartheta}\colon \widehat{\kappa})$, be a well-formed typing context, e be a well-typed expression with type τ w.r.t. $(\widehat{\Delta};\widehat{\Gamma})$, and $\Delta' = (\vartheta'\colon \kappa')$ be a context (not necessarily well-formed) such that ϑ' is fresh and $\widehat{\Delta} \wedge \Delta'$ is well-formed. Let $\mathcal{D}$ be a derivation for $\widehat{\Delta};\widehat{\Gamma} \vdash e\colon \tau$. We first show that the derivation $\mathcal{D}'$ obtained by replacing each context Δ occurring in $\mathcal{D}$ by $\Delta \wedge \Delta'$ is a derivation for $\widehat{\Delta} \wedge \Delta';\widehat{\Gamma} \vdash e\colon \tau$. We reason by induction on the depth of $\mathcal{D}$ by analyzing the last rule of $\mathcal{D}$.

First of all, we remark that ϑ' being fresh (1) if a context Δ is well-formed w.r.t. $\widehat{\Delta}$, then Δ is well-formed w.r.t. $\widehat{\Delta} \wedge \Delta'$, (2) if τ is well-formed w.r.t. $\widehat{\Delta}$, then τ is well-formed w.r.t. $\widehat{\Delta} \wedge \Delta'$, (3) if τ is a subtype of τ' w.r.t. $\widehat{\Delta}$, then τ is a subtype of τ' w.r.t. $\widehat{\Delta} \wedge \Delta'$, (4) if τ belongs to δ w.r.t. $\widehat{\Delta}$, then τ belongs to δ w.r.t. $\widehat{\Delta} \wedge \Delta'$, (5) app_Δ and dom_Δ depend on Δ exclusively to avoid the lexical capture of variables, (6) if $\widehat{\Delta};\widehat{\Gamma}$ is well-formed, then $\widehat{\Delta} \wedge \Delta';\widehat{\Gamma}$ is well-formed, and (7) if $\pi_1, \ldots, \pi_n$ is a partition of δ w.r.t. Δ, then it is also a partition of δ w.r.t. $\widehat{\Delta} \wedge \Delta'$.

To see why remark (7) holds, note that condition (1) of definition 44 holds for $\widehat{\Delta} \wedge \Delta'$ whenever it holds for $\widehat{\Delta}$, that condition (2) is independent from $\widehat{\Delta}$, and that if $(\exists\vartheta_C.\ d_C[\vartheta_C])$ is compatible with $\delta[\widehat{\Delta} \wedge \Delta']$, then it is obviously compatible with $\delta[\widehat{\Delta}]$, so that condition (3) holds for $\widehat{\Delta} \wedge \Delta'$ whenever it holds for $\widehat{\Delta}$.

Now, since rules *Var*, *Rec* and *Prj* do not depend on $\widehat{\Delta}$ (except for the freshness of ϑ_C) the theorem is trivially true if $\mathcal{D}$ has depth one, and by our preliminary remarks, all the other rules prove the theorem by induction. ■

Proof of theorem 48

Let $(\Delta;\Gamma)$, where $\Delta = \widehat{\vartheta}\colon \widehat{\kappa}$, be an implicit well-formed typing context and $\widehat{e}$ be a well-formed expression w.r.t. $(\Delta;\Gamma)$. We first remark that consecutive instances of the subsumption rule *Sub* in typing derivations can always be replaced by a single instance thanks to theorem 35. Consequently, it is easy to see that an expression is typable if and only if there exists a typing derivation such that (1) the subsumption rule is never used consecutively more than once and (2) the subsumption rule is neither the first nor the last rule of this derivation. Moreover, since the subsumption rule can only increase the type of a typable expression $\widehat{e}$ when it is used as the last rule of a typing derivation for $\widehat{e}$, we can restrict ourselves to such derivations for proving the minimal typing property. Similarly, since the subtyping judgement of the subsumption rule does not depend on Γ, we can also restrict ourselves to such derivations for proving the monotonicity of minimal typing w.r.t. typing contexts. Finally, we remark that there are only two rules applicable to any given expression: the subsumption rule and one of the rules *Var*, *Rec*, *Fun*, *App*, *Let* and *Meth*. Consequently, we can prove typing decidability, minimal typing and the monotonicity of minimal typing by induction on the syntax, showing that each "specific" rule either (1) decidably proves that $\widehat{e}$ is ill-typed or (2) decidably gives $\widehat{e}$ a minimal type $\widehat{\tau}$, in which case the same rule gives $\widehat{e}$ a subtype of $\widehat{\tau}$ in every subcontext of the current typing context.

If $\widehat{e}$ is an expression variable x, then rule *Var* and the well-formedness of $(\Delta;\Gamma)$ prove the existence of a well-formed minimal type $\widehat{\tau}$ such that $\widehat{e}$ is well-typed and has type $\widehat{\tau}$. Finally, if $(\Delta;\Gamma')$ is a subcontext of $(\Delta;\Gamma)$, then by definition, the type $\widehat{\tau}'$ of x in Γ' is a subtype of $\widehat{\tau}$ w.r.t. Δ.

If $\widehat{e}$ is a data constructor expression $\rho(d_C; \vartheta_C; x_1, \ldots, x_n)$, then rule *Rec* proves that $\widehat{e}$ is well-typed and has type $\widehat{\tau} = \forall\,\emptyset.\ d_C[\vartheta_C]$, which is well-formed since $\widehat{e}$ is well-formed w.r.t. $(\Delta; \Gamma)$. Finally, since $\widehat{\tau}$ is independent from Γ, $\widehat{e}$ has type $\widehat{\tau}$ w.r.t. any subcontext of $(\Delta; \Gamma)$.

If $\widehat{e}$ is a data extractor, then $\widehat{e}$ is well-typed since its constant type is well-formed w.r.t. Δ thanks to the hypothesis on the freshness of ϑ_C.

If $\widehat{e}$ is a method expression $\mathsf{meth}\ \{\vartheta \mid \kappa\}\,(x\colon \theta)\colon \theta' \Rightarrow [\pi_1 \Rightarrow e_1; \ldots; \pi_n \Rightarrow e_n]$ where $\pi_i = \exists \vartheta_i.\ \theta_i$. Let $\delta = \exists \vartheta\colon \kappa.\ \theta$ and $\Delta_i = (\upsilon, \vartheta, \vartheta_i)\colon (\kappa \wedge \kappa_i \wedge \upsilon \leq \theta \wedge \upsilon \leq \theta_i)$. If $\pi_1, \ldots, \pi_n$ is not a partition of δ w.r.t. Δ, or if Δ_i is not well-formed w.r.t. Δ, which is decidable for a given type structure, then $\widehat{e}$ is ill-typed. If not, then thanks to lemma 46, the typing context $\Delta[\Delta_i]; \Gamma[x\colon \forall\,\emptyset.\ \upsilon]$ is well-formed, and the well-typedness of e_i is decidable by induction. If any of the e_i is ill-typed, then so is the method. If every e_i is well-typed and has minimal type $\widehat{\tau}_i$, then either every $\widehat{\tau}_i$ is a subtype of $\forall\,\emptyset.\ \theta'$, and the method has minimal type $\forall \vartheta\colon \kappa.\ \theta \to \theta'$, or else the method is ill-typed. Assuming that the method is well-typed, let $(\Delta; \Gamma')$ be a subcontext of $(\Delta; \Gamma)$. By lemma 46, $\Delta[\Delta_i]; \Gamma'[x\colon \forall\,\emptyset.\ \upsilon]$ is thus a subcontext of $\Delta[\Delta_i]; \Gamma[x\colon \forall\,\emptyset.\ \upsilon]$. Therefore, we conclude by induction that the minimal type of each e_i w.r.t. this subcontext is a subtype of $\widehat{\tau}_i$, and thus a subtype of $\forall\,\emptyset.\ \theta'$, which proves that the method is well-typed w.r.t. the subcontext and has the same minimal type $\forall \vartheta\colon \kappa.\ \theta \to \theta'$.

Let us assume that $\widehat{e}$ is a let expression ($\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ e_0\ \mathsf{end}$). By induction, the well-typedness of e_1 is decidable. If e_1 is ill-typed, then so is $\widehat{e}$. If e_1 is well-typed, then it has a minimal type $\widehat{\tau}_1$, the typing context $\Delta; \Gamma[x_1\colon \widehat{\tau}_1]$ is well-formed and the well-typedness of e_0 in this context is decidable. If e_0 is ill-typed, which is decidable, then the monotonicity of minimal typing shows that e_0 cannot be well-typed in any typing context $\Delta; \Gamma[x_1\colon \tau_1]$ where τ_1 is a supertype of $\widehat{\tau}_1$ w.r.t. Δ, which proves that $\widehat{e}$ is ill-typed. If e_0 is well-typed and has minimal type $\widehat{\tau}_0$, then the monotonicity of minimal typing shows that for any supertype τ_1 of $\widehat{\tau}_1$ such that e_0 is well-typed w.r.t. $\Gamma[x_1\colon \tau_1]$ and has type τ_0, then $\widehat{\tau}_0$ is a subtype of τ_0, which shows that $\widehat{\tau}_0$ is a minimal type for the let expression. Finally, assuming that the let-expression is well-typed, let $(\Delta; \Gamma')$ be a subcontext of $(\Delta; \Gamma)$. Then, by induction, the monotonicity of minimal typing implies that the minimal type $\widehat{\tau}'_1$ of e_1 w.r.t. this subcontext is a subtype of $\widehat{\tau}_1$, and the minimal type $\widehat{\tau}'_0$ of e_0 w.r.t. $\Delta; \Gamma'[x_1 : \widehat{\tau}'_1]$ is a subtype of $\widehat{\tau}_0$ w.r.t. Δ.

Let us assume that $\widehat{e}$ is a recursive let-expression ($\mathsf{letrec}\ x_1\colon \tau_1 = e_1; \ldots; x_n\colon \tau_n = e_n\ \mathsf{in}\ e_0\ \mathsf{end}$). If any type τ_i is ill-formed w.r.t. Δ, which is decidable, then the typing context $\Delta; \Gamma[x_1\colon \tau_1, \ldots, x_n\colon \tau_n]$, is ill-formed and $\widehat{e}$ is ill-typed. If $\Delta; \Gamma[x_1\colon \tau_1, \ldots, x_n\colon \tau_n]$ is well-formed, then since each e_i, $i \in [1, n]$, is a strict subexpression of $\widehat{e}$, its well-typedness w.r.t. $\Delta; \Gamma[x_1\colon \tau_1, \ldots, x_n\colon \tau_n]$ is decidable. If any of the e_i is ill typed, then so is $\widehat{e}$. If they are all well-typed and have minimal type $\widehat{\tau}_i$, then either $\widehat{\tau}_i$ is a subtype of τ_i for every $i \in [1, n]$ (which is decidable) and $\widehat{e}$ has minimal type $\widehat{\tau}_0$, or else $\widehat{e}$ is ill-typed. Finally, assuming that $\widehat{e}$ is well-typed, let $(\Delta; \Gamma')$ be a subcontext of $(\Delta; \Gamma)$. Then $\Delta; \Gamma'[x_1\colon \tau_1, \ldots, x_n\colon \tau_n]$ is obviously a subcontext of $\Delta; \Gamma[x_1\colon \tau_1, \ldots, x_n\colon \tau_n]$ and by induction, each e_i has a minimal type $\widehat{\tau}'_i$ w.r.t. this subcontext, which is a subtype of $\widehat{\tau}_i$ and thus a subtype of τ_i w.r.t. Δ, which proves that $\widehat{e}$ has type $\widehat{\tau}'_0$ which is a subtype of $\widehat{\tau}_0$ w.r.t. Δ.

If $\widehat{e}$ is an application $(e\ e_2)$, then e_1 and e_2 being strict subexpressions of $\widehat{e}$, the well-typedness of e_1 and e_2 is decidable. If e_1 or e_2 is ill-typed, then obviously $\widehat{e}$ is also ill-typed. So let us assume that both e_1 and e_2 are well-typed, and let $\widehat{\tau}_1$ and $\widehat{\tau}_2$ denote their minimal types, the determination of which is decidable. If $\widehat{\tau}_2$ belongs to $dom_\Delta(\widehat{\tau}_1)$ w.r.t. Δ, which is decidable, then $(e_1\ e_2)$ is well-typed and has type $app_\Delta(\widehat{\tau}_1, \widehat{\tau}_2)$, and theorem 41 proves that this type is both well-formed and minimal. Now, suppose that $\widehat{\tau}_2$ does not belong to $dom_\Delta(\widehat{\tau}_1)$ w.r.t. Δ, and let τ_1 and τ_2 be two other possible types of e_1 and e_2 such that τ_2 belongs to $dom_\Delta(\tau_1)$ w.r.t. Δ. Theorem 41 shows that $app_\Delta(\widehat{\tau}_1, \widehat{\tau}_2)$ is well-formed, which trivially implies that $\widehat{\tau}_1$ belongs to the domain of $\widehat{\tau}_2$, which is absurd. Consequently, $\widehat{e}$ is either decidably ill-typed or has minimal type $app_\Delta(\widehat{\tau}_1, \widehat{\tau}_2)$. Finally, assuming that $\widehat{e}$ is well-typed, let $(\Delta; \Gamma')$ be a subcontext of $(\Delta; \Gamma)$. Then by induction, e and e_2 have minimal types $\widehat{\tau}'_1$ and $\widehat{\tau}'_2$ which are subtypes of $\widehat{\tau}_1$ and $\widehat{\tau}_2$ and theorem 41 shows once again that $\widehat{e}$ has minimal type $app_\Delta(\widehat{\tau}'_1, \widehat{\tau}'_2)$ which is a subtype of $app_\Delta(\widehat{\tau}_1, \widehat{\tau}_2)$.

Let us assume that $\widehat{e}$ is a lambda-expression $(\mathsf{fun}\ \{\vartheta \mid \kappa\}\ (x\colon \theta) \Rightarrow e)$. If $(\vartheta\colon \kappa)$ is not well-formed w.r.t. Δ, which is decidable, then $\widehat{e}$ is ill-typed. So let us assume that $(\vartheta\colon \kappa)$ is well-formed w.r.t. Δ, that is, $(\vartheta \mathrel{\#} \widehat{\vartheta})$, κ is $(\vartheta, \widehat{\vartheta})$-closed, and $\widehat{\kappa}$ implies κ for all $\widehat{\vartheta}$. Since $\widehat{e}$ is well-formed by assumption, θ is $(\vartheta, \widehat{\vartheta})$-closed, hence $\widehat{\kappa}$ implies $\kappa \wedge \theta \leq \theta$ and therefore we have

$$\forall \widehat{\vartheta}.\ \widehat{\kappa} \models \kappa \wedge v \leq \theta \qquad (*)$$

because v is fresh. Using lemma 46, it follows that the typing context $\Delta[\vartheta, v\colon \kappa \wedge v \leq \theta]; \Gamma[x\colon \forall \emptyset.\ v]$ is well-formed. Since e is a strict subexpression of $\widehat{e}$, the well-typedness of e w.r.t. $\Delta[\vartheta, v\colon \kappa \wedge v \leq \theta]; \Gamma[x\colon \forall \emptyset.\ v]$ is decidable by induction hypothesis. If e is ill-typed, then so is $\widehat{e}$. So let us assume that e is well-typed w.r.t. $\Delta[\vartheta, v\colon \kappa \wedge v \leq \theta]; \Gamma[x\colon \forall \emptyset.\ v]$, and let $\tau' = \forall \vartheta'\colon \kappa'.\ \theta'$ denote its minimal type, which can be determined by induction hypothesis. Let $\widehat{\tau}$ denote the type $(\forall \vartheta, \vartheta', v\colon \kappa \wedge \kappa' \wedge v \leq \theta.\ v \to \theta')$. Since τ' is well-formed w.r.t. $\Delta[\vartheta, v\colon \kappa \wedge v \leq \theta]$, we have that $\vartheta' \mathrel{\#} (\widehat{\vartheta}, \vartheta, v)$, that κ' and θ' are $(\widehat{\vartheta}, \vartheta, v, \vartheta')$-closed, and $\widehat{\kappa} \wedge \kappa \wedge v \leq \theta$ implies κ' for all $(\widehat{\vartheta}, \vartheta, v)$. Thus, by corollary 5, $\widehat{\kappa} \wedge \kappa \wedge v \leq \theta$ implies $\kappa \wedge v \leq \theta \wedge \kappa'$ for all $(\widehat{\vartheta}, \vartheta, v)$, and by lemma 1, $\widehat{\kappa} \wedge \kappa \wedge v \leq \theta$ implies $\kappa \wedge v \leq \theta \wedge \kappa'$ for all $\widehat{\vartheta}$. By $(*)$ and corollary 5, $\widehat{\kappa}$ implies $\widehat{\kappa} \wedge \kappa \wedge v \leq \theta$ for all $\widehat{\vartheta}$. By rule *Trans*, we conclude that $\widehat{\kappa}$ implies $\kappa \wedge v \leq \theta \wedge \kappa'$ for all $\widehat{\vartheta}$, which proves that $\widehat{\tau}$ is well-formed. Now, let $\tau'' = \forall \vartheta''\colon \kappa''.\ \theta''$ be any supertype of τ' w.r.t. $\Delta[\vartheta, v\colon \kappa \wedge v \leq \theta]$, and let $\widehat{u}$, u and u' be three fresh and distinct type variables and u'' be a fresh Arrow-constructor variable. By hypothesis, we have

$$\forall \widehat{\vartheta}, \vartheta, v, u'.\ \widehat{\kappa} \wedge \kappa \wedge v \leq \theta \wedge \kappa'' \wedge \theta'' \leq u' \models \kappa' \wedge \theta' \leq u' \qquad (**)$$

therefore

$$\begin{array}{lll}
\forall \widehat{\vartheta}, \widehat{u}. & \widehat{\kappa} \wedge \kappa \wedge v \leq \theta \wedge \kappa'' \wedge (v \to \theta'') \leq \widehat{u} & (\mathit{VElim}) \\
\models & \widehat{\kappa} \wedge \kappa \wedge v \leq \theta \wedge \kappa'' \wedge (v \to \theta'') \leq \widehat{u} \wedge \widehat{u} = u''[u, u'] & (\mathit{Trans}, \mathit{MElim}) \\
\models & (\widehat{\kappa} \wedge \kappa \wedge v \leq \theta \wedge \kappa'' \wedge \theta'' \leq u') \wedge (u \leq v \wedge \widehat{u} = u''[u, u'] \wedge (\to) \sqsubseteq u'') &
\end{array}$$

From $(**)$, using

$$(\kappa' \wedge \theta' \leq u') \mathrel{\#} (u \leq v \wedge \widehat{u} = u''[u, u'] \wedge (\to) \sqsubseteq u'')\ [\widehat{u}, u', \vartheta, \widehat{\vartheta}, v]$$

and the fact that $\widehat{u}$ does not occur in $(**)$, lemma 2 and lemma 3 imply that

$$\begin{array}{lll}\forall\vartheta,\widehat{\vartheta},v,\widehat{u},u'. & (\widehat{\kappa}\wedge\kappa\wedge v\leq\theta\wedge\kappa''\wedge\theta''\leq u')\wedge(u\leq v\wedge\widehat{u}=u''[u,u']\wedge(\rightarrow)\sqsubseteq u'') & \\ \models & (\kappa'\wedge\theta'\leq u')\wedge(u\leq v\wedge\widehat{u}=u''[u,u']\wedge(\rightarrow)\sqsubseteq u'') & (\mathit{MIntro},\mathit{Trans}) \\ \models & \kappa'\wedge v\rightarrow\theta'\leq\widehat{u} & \end{array}$$

Now, lemma 1 yields

$$\begin{array}{ll}\forall\widehat{\vartheta},\widehat{u}. & (\widehat{\kappa}\wedge\kappa\wedge v\leq\theta\wedge\kappa''\wedge\theta''\leq u')\wedge(u\leq v\wedge\widehat{u}=u''[u,u']\wedge(\rightarrow)\sqsubseteq u'') \\ \models & \kappa'\wedge v\rightarrow\theta'\leq\widehat{u}\end{array}$$

and by transitivity we obtain

$$\begin{array}{ll}\forall\widehat{\vartheta},\widehat{u}. & \widehat{\kappa}\wedge\kappa\wedge v\leq\theta\wedge\kappa''\wedge(v\rightarrow\theta'')\leq\widehat{u} \\ \models & \kappa'\wedge v\rightarrow\theta'\leq\widehat{u}\end{array}$$

which shows that $\widehat{\tau}$ is a subtype of $(\forall\vartheta,\vartheta'',v\colon\kappa\wedge\kappa''\wedge v\leq\theta.\ v\rightarrow\theta'')$ w.r.t. $\widehat{\vartheta}\colon\widehat{\kappa}$ and is thus minimal. Finally, assuming that $\widehat{e}$ is well-typed, let $(\Delta;\Gamma')$ be a subcontext of $(\Delta;\Gamma)$. By lemma 46, $\Delta[\vartheta,v\colon\kappa\wedge v\leq\theta];\Gamma'[x\colon\forall\emptyset.\ v]$ is thus a subcontext of $\Delta[\vartheta,v\colon\kappa\wedge v\leq\theta];\Gamma[x\colon\forall\emptyset.\ v]$, and by induction hypothesis, the minimal type of e' w.r.t. this subcontext is a subtype of τ'. Using the same monotonicity proof as above, we thus have that the minimal type of $\widehat{e}$ w.r.t. this subcontext is a subtype of $\widehat{\tau}$ w.r.t. Δ, which ends the proof.
■

Proof of lemma 49

Let $\Delta=\widehat{\vartheta}\colon\widehat{\kappa}$ be a well-formed context, $\tau=\forall\vartheta\colon\kappa.\ \theta$ be a well-formed type w.r.t. the trivial context such that $(\vartheta\ \#\ \widehat{\vartheta})$, and $\tau'=\forall\vartheta'\colon\kappa'.\ \theta'$ be a well-formed type w.r.t. Δ. Without loss of generality, we assume that $\vartheta'\ \#\ \vartheta\ \#\ \widehat{\vartheta}$.

(1) Type $\tau[\Delta]$ is obviously closed, and it is well-formed since if *true* implies κ and *true* implies $\widehat{\kappa}$, then corollary 4 shows that *true* implies $\widehat{\kappa}\wedge\kappa$. Now, if v is a fresh type variable, then $\forall v.\ \kappa\wedge\theta\leq v\models\kappa\wedge\theta\leq v$ and $\forall v.\ \mathit{true}\models\widehat{\kappa}$ (using lemma 2). Consequently, lemma 3 shows that $\forall v.\ \kappa\wedge\theta\leq v\models\kappa\wedge\widehat{\kappa}\wedge\theta\leq v$. Since, trivially, $\forall v.\ \kappa\wedge\widehat{\kappa}\wedge\theta\leq v\models\kappa\wedge\theta\leq v$, we conclude that τ and $\tau[\Delta]$ are equivalent w.r.t. *True*.

(2) Since $(\vartheta'\colon\kappa')$ is well-formed w.r.t. Δ, $\tau'[\Delta]$ is trivially closed and well-formed.

(3) Let us now assume that τ is a subtype of τ' w.r.t. Δ. For any fresh type variable $\widehat{v}$, we have

$$\forall\widehat{v},\widehat{\vartheta}.\ \widehat{\kappa}\wedge\kappa'\wedge\theta'\leq\widehat{v}\models\kappa\wedge\theta\leq\widehat{v}$$

and by lemma 1

$$\forall\widehat{v}.\ (\widehat{\kappa}\wedge\kappa')\wedge\theta'\leq\widehat{v}\models\kappa\wedge\theta\leq\widehat{v}$$

which proves that τ is a subtype of $\tau'[\Delta]$ w.r.t. *True*. Conversely, assuming that

$$\forall\widehat{v}.\ \widehat{\kappa}\wedge\kappa'\wedge\theta'\leq\widehat{v}\models\kappa\wedge\theta\leq\widehat{v}$$

we remark that the right-hand side of the implication does not contain variables in $\widehat{\vartheta}$. Therefore, lemma 2 gives us

$$\forall \widehat{v}, \widehat{\vartheta}.\ \widehat{\kappa} \wedge \kappa' \wedge \theta' \leq \widehat{v} \models \kappa \wedge \theta \leq \widehat{v}$$

which proves that τ is a subtype of τ' w.r.t. Δ.

■

Proof of theorem 50

Let $\widehat{\Omega}$ be a well-formed run-time environment, $\widehat{\Delta}$ be a well-formed constraint context, and $\widehat{e}$ be a well-typed expression with type $\widehat{\tau}$ w.r.t. $(\widehat{\Delta}; \widehat{\Omega})$. We prove the theorem by cases by showing that the abstract machine always remains well-formed while evaluating $\widehat{e}$ and that if the evaluation of $\widehat{e}$ terminates, it returns a run-time value with a closed run-time type which is a subtype of the closure of $\widehat{\tau}$ w.r.t. $\widehat{\Delta}$.

If $\widehat{e}$ is an expression variable, then rule *VarVal* shows that the evaluation terminates and returns a well-formed run-time value ω with run-time type τ such that $\omega : \tau$ is a subterm of $\widehat{\Omega}$. But since $\widehat{e}$ is well-typed w.r.t. $(\widehat{\Delta}; \widehat{\Omega})$ by hypothesis, rule *Var* shows that $\widehat{\tau} = \tau$, and τ being a fresh closed run-time type and $\widehat{\Delta}$ being a well-formed context, it is easy to see that $\widehat{\tau}[\widehat{\Delta}]$ is well-formed and is equivalent to τ, and thus that τ is a subtype of $\widehat{\tau}[\widehat{\Delta}]$ w.r.t. *True*.

If $\widehat{e}$ is an abstraction, then rule *ClsVal* shows that the evaluation terminates and returns run-time value $\mathsf{cls}(\widehat{\Delta}, \widehat{e})$ with well-formed and closed type $\widehat{\tau}[\widehat{\Delta}]$. Finally, rules *Fun* and *Meth* show that $\widehat{\tau}$, and, hence, $\widehat{\tau}[\widehat{\Delta}]$, are run-time types and rule *ClsType* shows that $\mathsf{cls}(\widehat{\Delta}, \widehat{e})$ is well-formed and has type $\widehat{\tau}[\widehat{\Delta}]$ w.r.t. $\widehat{\Omega}$.

If $\widehat{e}$ is a projection d^i_C, then rules *PrjVal* and *Prj* show that $\widehat{\tau} = (\forall \vartheta_C.\ d^i_C\langle\vartheta_C\rangle \to d_C[\vartheta_C])$, which is a well-formed and closed run-time type. Moreover, the assumption on the freeness of ϑ_C shows that $\widehat{\tau}[\widehat{\Delta}]$ is well-formed and is equivalent to $\widehat{\tau}$. Consequently, $\widehat{e}$ evaluates to $\mathsf{prj}(d_C, i)$ with closed run-time type $\widehat{\tau}[\widehat{\Delta}]$ and rule *PrjType* shows that $\mathsf{prj}(d_C, i)$ is well-formed w.r.t. $\widehat{\Omega}$.

If $\widehat{e}$ is a record expression of the form $\rho(d_C; \emptyset)$, then rule *Rec* shows that $\widehat{\tau} = \forall\, \emptyset.\ d_C[\,]$ and rule *RecVal* shows that the evaluation of $\widehat{e}$ terminates and returns run-time value $\mathsf{rec}(d_C)$ with closed and well-formed run-time type $\widehat{\tau}[\widehat{\Delta}]$. Rule *RecType* shows that this run-time value is well-formed w.r.t. $\widehat{\Omega}$.

If $\widehat{e}$ is a record expression of the form $\rho(d_C; \vartheta_C; x_1, \ldots, x_n)$, $n \geq 1$, then the hypothesis on the possible occurrences of such terms shows that $\widehat{\Omega}$ necessarily contains n bindings $x_1 = \omega_1 : (\forall \vartheta_1 : \kappa_1.\ \theta_1)$, $\ldots$, $x_n = \omega_n : (\forall \vartheta_n : \kappa_n.\ \theta_n)$ and that $\widehat{\Delta}$, which results from n applications of rule *FunApp* started in the trivial context, is of the form

$$(\kappa_1 \wedge \theta_1 \leq d^1_C\langle\vartheta_C\rangle) \wedge \ldots \wedge (\kappa_n \wedge \theta_n \leq d^n_C\langle\vartheta_C\rangle)$$

Now, by rule *Rec*, the type τ of the record expression w.r.t. $\widehat{\Delta}; \widehat{\Omega}$ is equal to $\forall\, \emptyset.\ d_C[\vartheta_C]$. Consequently, $\tau[\widehat{\Delta}]$, which is well-formed, is precisely $(d_C\ \tau_1 \ldots \tau_n)$, which shows that the evaluation of $\widehat{e}$ terminates and returns run-time value $\mathsf{rec}(d_C; \omega_1 : \tau_1, \ldots, \omega_n : \tau_n)$ which is well-formed and has type $\tau[\widehat{\Delta}]$ w.r.t. $\widehat{\Omega}$.

If $\widehat{e}$ is a let-expression ($\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ e_0\ \mathsf{end}$), let $\widehat{\Gamma}$ be the typing context associated to $\widehat{\Omega}$. Rule *Let* shows that $\widehat{\Delta}; \widehat{\Gamma} \vdash e_1 : \widehat{\tau}_1$ and $\widehat{\Delta}; \widehat{\Gamma}[x_1 : \widehat{\tau}_1] \vdash e_0 : \widehat{\tau}_0$. Let us assume that $\widehat{\tau}_1$ and

$\widehat{\tau}_0$ are minimal. We thus have $\widehat{\Delta}; \widehat{\Omega} \vdash e_1 : \widehat{\tau}_1$, so starting the evaluation of e_1 in context $\widehat{\Delta}$ yields a well-formed abstract machine. Let us assume that the evaluation terminates and returns a run-time value ω_1 with type τ_1 w.r.t. Ω_1. By hypothesis, τ_1 is a subtype of $\widehat{\tau}_1[\widehat{\Delta}]$ w.r.t. *True*. Since τ_1 is fresh, we deduce that τ_1 is a subtype of $\widehat{\tau}_1$ w.r.t. $\widehat{\Delta}$. Consequently, theorem 48 shows that e_0 is well-typed w.r.t. $\widehat{\Gamma}[x_1 : \tau_1]$ and has minimal type $\widehat{\tau}'_0$ which is a subtype of $\widehat{\tau}_0$ w.r.t. $\widehat{\Delta}$. Therefore, since Ω_1 "extends" $\widehat{\Omega}$, starting the evaluation of e_0 in context $\widehat{\Delta}$ and run-time environment $\Omega_1[x_1 = \omega_1 : \tau_1]$ yields a well-formed abstract machine. Let us assume that the evaluation terminates and returns a well-formed run-time value ω_0 with type τ_0. By hypothesis, τ_0 is a subtype of $\widehat{\tau}'_0[\widehat{\Delta}]$ w.r.t. $\widehat{\Delta}$. Since τ_0 is a fresh and closed run-time type, we deduce that τ_0 is a subtype of $\widehat{\tau}'_0$ w.r.t. $\widehat{\Delta}$ and is thus a subtype of $\widehat{\tau}_0$ w.r.t. $\widehat{\Delta}$. Since $\widehat{\tau} = \widehat{\tau}_0$, we conclude that τ_0 is a subtype of $\widehat{\tau}[\widehat{\Delta}]$ w.r.t. *True*.

If $\widehat{e}$ is a recursive let-expression ($\mathsf{letrec}\ x_1 : \tau_1 = f_1; \ldots; x_n : \tau_n = f_n\ \mathsf{in}\ e_0\ \mathsf{end}$), let $\widehat{\Gamma}$ be the typing context associated to $\widehat{\Omega}$. Rule *LetRec* shows that each f_i is well-typed w.r.t. $\widehat{\Delta}$ and that its minimal type $\widehat{\tau}_i$ is a subtype of τ_i w.r.t. $\widehat{\Delta}$. Moreover, e_0 has minimal type $\widehat{\tau}_0$ w.r.t. $\widehat{\Delta}; \widehat{\Gamma}[x_1 : \tau_1, \ldots, x_n : \tau_n]$. Consequently, $\widehat{\tau}_i[\widehat{\Delta}]$ is a subtype of $\tau_i[\widehat{\Delta}]$ w.r.t. *True*, and rule *ClsType* shows that $\mathsf{cls}(\widehat{\Delta}, f_i)$ is well-formed and has type $\tau_i[\widehat{\Delta}]$ w.r.t. $\widehat{\Omega}$. Run-time environment $\Omega[x_1 = \mathsf{cls}(\widehat{\Delta}, f_1) : \tau_1[\widehat{\Delta}], \ldots, x_n = \mathsf{cls}(\widehat{\Delta}, f_n) : \tau_n[\widehat{\Delta}]]$ is thus well-formed. Now, up to an α-substitution of its variables, each closed run-time type $\tau_i[\widehat{\Delta}]$ is a subtype of τ_i w.r.t. $\widehat{\Delta}$. Therefore, $\widehat{\Delta}; \widehat{\Gamma}[x_1 : \tau_1[\widehat{\Delta}], \ldots, x_n : \tau_n[\widehat{\Delta}]]$ is a subcontext of $\widehat{\Delta}; \widehat{\Gamma}[x_1 : \tau_1, \ldots, x_n : \tau_n]$, and theorem 48 shows that e_0 is well-typed w.r.t. this subcontext and has minimal type $\widehat{\tau}'_0$ which is a subtype of $\widehat{\tau}_0 = \widehat{\tau}$ w.r.t. $\widehat{\Delta}$. Consequently, starting the evaluation of e_0 in environment $\widehat{\Omega}[x_1 = \mathsf{cls}(\widehat{\Delta}, f_1) : \tau_1[\widehat{\Delta}], \ldots, x_n = \mathsf{cls}(\widehat{\Delta}, f_n) : \tau_n[\widehat{\Delta}]]$ yields a well-formed abstract machine. Let us assume that this evaluation terminates and returns a run-time value ω_0 with closed run-time type τ_0. By hypothesis, we deduce that τ_0 is a subtype of $\widehat{\tau}'_0[\widehat{\Delta}]$ and thus that τ_0 is a subtype of $\widehat{\tau}[\widehat{\Delta}]$ w.r.t. $\widehat{\Delta}$.

If $\widehat{e}$ is an application $(\bar{e}\ \bar{e}\,')$, let $\widehat{\Gamma}$ be the typing context associated to $\widehat{\Omega}$. Rule *App* shows that

$$\begin{cases} \widehat{\Delta}; \widehat{\Gamma} \vdash \bar{e} : \bar{\tau} \\ \widehat{\Delta}; \widehat{\Gamma} \vdash \bar{e}\,' : \bar{\tau}\,' \\ \widehat{\Delta} \vdash \bar{\tau}\,' \in \mathit{dom}_{\widehat{\Delta}}(\bar{\tau}) \\ \widehat{\tau} = \mathit{app}_{\widehat{\Delta}}(\bar{\tau}, \bar{\tau}\,') \end{cases}$$

and starting the evaluation of $\bar{e}$ in $(\widehat{\Delta}; \widehat{\Omega})$ yields a well-formed abstract machine. Let us assume that $\bar{\tau}$ and $\bar{\tau}\,'$ are minimal and that this evaluation terminates and returns a run-time value ω with closed run-time type τ w.r.t. Ω. Similarly, starting the evaluation of $\bar{e}\,'$ in $(\widehat{\Delta}; \Omega)$ yields a well-formed abstract machine. Let us assume that this evaluation also terminates and returns a run-time value ω' with closed run-time type τ' w.r.t. Ω'. By hypothesis, we deduce that τ is a subtype of $\bar{\tau}[\widehat{\Delta}]$ w.r.t. *True* and that τ' is a subtype of $\bar{\tau}\,'[\widehat{\Delta}]$ w.r.t. *True*. Since τ and τ' are fresh, we conclude that τ is a subtype of $\bar{\tau}$ w.r.t. $\widehat{\Delta}$ and that τ' is a subtype of $\bar{\tau}\,'$ w.r.t. $\widehat{\Delta}$. But $\bar{\tau}$ being prefunctional, theorems 40 and 39 show that τ' belongs to the domain of τ w.r.t. $\widehat{\Delta}$. However, since τ and τ' are closed, and $\widehat{\Delta}$ is well-formed, we conclude that τ' belongs to the domain of τ w.r.t. *True*, which shows that τ is a well-formed functional type. Looking at the rules of figure 6.1, it is clear that ω can only be of the form $\mathsf{prj}(d_C, i)$ or $\mathsf{cls}(\Delta, f)$. There are three subcases to consider.

If ω is the i-th projection $\mathsf{prj}(d_C, i)$, then by rules *Prj* and *PrjVal* we deduce that τ is of the form $\forall \vartheta_C.\ d_C[\vartheta_C] \to d_C^i\langle\vartheta_C\rangle$, where ϑ_C is fresh, which is a well-formed and closed run-time type. Therefore, τ' belongs to $\exists \vartheta_C.\ d_C[\vartheta_C]$ w.r.t. *True*. Since C cannot be the arrow class, for which no projection is defined, the rules of figure 6.1 show (1) that ω' can only be a record $\mathsf{rec}(d_C; \omega_1 \colon \tau_1, \ldots, \omega_n \colon \tau_n)$ with well-formed type $\tau' = (d_C\, \tau_1 \ldots \tau_n)$ w.r.t. Ω', and (2) that $\Omega' \vdash \omega_i \colon \tau_i$ for $i \in [1, n]$. Rule *PrjApp* can thus be applied and the evaluation of $\widehat{e}$ returns run-time value ω_i with closed run-time type τ_i w.r.t. Ω'. Moreover, by theorem 41, we know that $app(\tau, \tau')$ is a subtype of $\widehat{\tau} = app_{\widehat{\Delta}}(\bar{\tau}, \bar{\tau}\,')$ w.r.t. $\widehat{\Delta}$, and thus that $app(\tau, \tau')$ is a subtype of $\widehat{\tau}[\widehat{\Delta}]$ w.r.t. *True*. Consequently, theorem 42 shows that τ_i is a subtype of $\widehat{\tau}[\widehat{\Delta}]$ w.r.t. *True*.

If ω is a method $\mathsf{cls}(\Delta, \mathsf{meth}\ \{\vartheta \mid \kappa\}\, (x \colon \theta) \colon \theta'' \Rightarrow [\pi_1 \Rightarrow e_1; \ldots; \pi_n \Rightarrow e_n])$, with $\Delta = \tilde{\vartheta} \colon \tilde{\kappa}$, then rule *ClsType* shows that $\tilde{\tau} = \forall \tilde{\vartheta}, \vartheta \colon \tilde{\kappa} \wedge \kappa.\ \theta \to \theta''$ is a subtype of τ w.r.t. *True*, and also that $\pi_1, \ldots, \pi_n$ is a partition of $\delta = \exists \vartheta \colon \kappa.\ \theta$ w.r.t. Δ. Theorems 40 and 39 thus prove that τ' belongs to the domain $\tilde{\delta} = \delta[\Delta]$ of $\tilde{\tau}$ w.r.t. *True*, so that by theorem 45, we know that $i = disp_\Delta(\tau'; \delta[\Delta]; \pi_1, \ldots, \pi_n)$ is well-defined and that τ' belongs to $\delta[\Delta] \wedge \pi_i$, that is, assuming that π_i is of the form $\exists \vartheta_i.\ \theta_i$, τ' belongs to

$$\exists \tilde{\vartheta}, v, \vartheta, \vartheta_i \colon \tilde{\kappa} \wedge \kappa \wedge v \leq \theta, \theta_i.\ v$$

and by theorem 43, we also know that

$$app(res(\tilde{\tau}, \pi_i), \tau') \quad \doteq \quad app(\tilde{\tau}, \tau')$$

So let v be a fresh type variable and let Δ_i denote the context $\Delta[v, \vartheta, \vartheta_i \colon \kappa \wedge v \leq \theta, \theta_i]$ and Δ' denote the context $(\vartheta' \colon \kappa' \wedge \theta' \leq v)$. Rule *Meth* shows that Δ_i is well-formed and that e_i has type $\forall \emptyset.\ \theta''$ in typing context $\Delta_i; \Gamma[x \colon \forall \emptyset.\ v]$. Now, the fact that τ' belongs to $\tilde{\delta} \wedge \pi_i$ implies that $\Delta_i \wedge \Delta'$ is well-formed, and by lemma 47, we deduce that e_i has type $\forall \emptyset.\ \theta''$ w.r.t. typing context $\Delta_i \wedge \Delta'; \Gamma[x \colon \forall \emptyset.\ v]$. Finally, it is easy to see that, up to an α-substitution of its variables, τ' is a subtype of $(\forall \emptyset.\ v)$ w.r.t. $\Delta_i \wedge \Delta'$ and thus, by theorem 48, we conclude that e_i is well-typed and has type $\forall \emptyset.\ \theta''$ w.r.t. typing context $\Delta_i \wedge \Delta'; \Gamma[x \colon \tau']$. Consequently, starting the evaluation of e_i in environment $\Delta_i \wedge \Delta'; \Omega'[x = \omega' \colon \tau']$ yields a well-formed abstract machine. Assuming that this evaluation terminates and returns a run-time value ω'' with closed run-time type τ'', we know by hypothesis that τ'' is a subtype of the closure of $\forall \emptyset.\ \theta''$ w.r.t. $\Delta_i \wedge \Delta'$, which happens to be equivalent to $app(res(\tilde{\tau}, \pi_i), \tau')$ and is thus equivalent to $app(\tilde{\tau}, \tau')$ which, by theorem 41, is a subtype of $app(\tau, \tau')$ and is thus a subtype of $app(\bar{\tau}[\widehat{\Delta}], \bar{\tau}\,'[\widehat{\Delta}])$, which finally shows that τ'' is a subtype of $app(\bar{\tau}, \bar{\tau}\,')[\widehat{\Delta}] \doteq \widehat{\tau}[\widehat{\Delta}]$ w.r.t. *True*.

If ω is a function $\mathsf{cls}(\Delta, \mathsf{fun}\ \{\vartheta \mid \kappa\}\, (x \colon \theta) \Rightarrow e)$ where the expression $\mathsf{fun}\ \{\vartheta \mid \kappa\}\, (x \colon \theta) \Rightarrow e$ has minimal type $\forall \vartheta, \vartheta'', v \colon \kappa \wedge \kappa'' \wedge v \leq \theta.\ v \to \theta''$ w.r.t. Δ (where v is a fresh type variable), then rule *ClsType* shows that type $\tilde{\tau} = \forall \tilde{\vartheta}, \vartheta, \vartheta'', v \colon \tilde{\kappa} \wedge \kappa \wedge \kappa'' \wedge v \leq \theta.\ v \to \theta''$ is a subtype of τ w.r.t. *True*. Theorems 40 and 39 thus prove that τ' belongs to the domain $\tilde{\delta}$ of $\tilde{\tau}$ w.r.t. *True*. So let Δ_0 denote the context $\Delta[v, \vartheta \colon \kappa \wedge v \leq \theta]$ and Δ' denote the context $(\vartheta' \colon \kappa' \wedge \theta' \leq v)$. Rule *Fun* and lemma 47 show that Δ_0 is well-formed and that e has type $\forall \vartheta'' \colon \kappa''.\ \theta''$ in typing context $\Delta_0; \Gamma[x \colon \forall \emptyset.\ v]$. Now, the fact that τ' belongs to $\tilde{\delta}$ implies in particular that $\Delta_0 \wedge \Delta'$ is well-formed, and by lemma 47, we deduce that e has type $\forall \vartheta'' \colon \kappa''.\ \theta''$ w.r.t. typing context $\Delta_0 \wedge \Delta'; \Gamma[x \colon \forall \emptyset.\ v]$. Finally, it is easy to see that, up to an α-substitution of its variables, τ' is a subtype of $\forall\emptyset.\ v$ w.r.t. $\Delta_0 \wedge \Delta'$ and thus, by theorem 48, we conclude that e is well-typed and has type $\forall\vartheta''\colon \kappa''.\ \theta''$ w.r.t. typing context $\Delta_0 \wedge \Delta'; \Gamma[x\colon \tau']$. Consequently, starting the evaluation of e in environment $\Delta_0 \wedge \Delta'; \Omega'[x = \omega'\colon \tau']$ yields a well-formed abstract machine. Assuming that this evaluation terminates and returns a run-time value ω'' with closed run-time type τ'', we know by hypothesis that τ'' is a subtype of the closure of $\forall\vartheta''\colon \kappa''.\ \theta''$ w.r.t. $\Delta_0 \wedge \Delta'$, which happens to be equivalent to $app(\tilde{\tau}, \tau')$ which, by theorem 41, is a subtype of $app(\tau, \tau')$ and is thus a subtype of $\hat{\tau}[\widehat{\Delta}]$ w.r.t. *True*. ■

Proof of theorem 51

Let σ be a solution, and F be such that $F(v) = \{\sigma(v)\}$ for every variable $v \in \vartheta$. Then since $\sigma(v_1) = v_1$ for every variable $v_1 \in \vartheta_1$, we have $F(v_1) \subseteq \Phi(F)(v_1)$. Now, let $v_2 \in \vartheta_2$, and v be any variable in ϑ such that $v \preceq v_2$. We know that since σ is a solution, $\sigma(v) \preceq \sigma(v_2)$, so $\sigma(v_2)$ belongs to $\bigcup_{v_1 \in F(v)} \uparrow v_1$, and $\sigma(v_2)$ belongs to $(\bigcap_{v \preceq v_2} \bigcup_{v_1 \in F(v)} \uparrow v_1)$. Similarly, we show that $\sigma(v_2)$ belongs to $(\bigcap_{v_2 \preceq v} \bigcup_{v_1 \in F(v)} \downarrow v_1)$, and, finally, we conclude that $F(\sigma_2) \subseteq \Phi(F)(\sigma_2)$, which proves that F is a pre-solution. ■

Proof of theorem 52

Let us assume that G is a maximum pre-solution such that $|G(v_2)| = 1$ for every $v_2 \in \vartheta_2$. Since the same property holds for $v_1 \in \vartheta_1$, we denote by $\sigma(v)$ the unique element of $G(v)$ for any $v \in \vartheta$. We thus have $\sigma(v_1) = v_1$ for every $v_1 \in \vartheta_1$, and $G(v) = \{\sigma(v)\}$ for every $v \in \vartheta$. Moreover, for every $v_2 \in \vartheta_2$, we have

$$\begin{cases} \sigma(v_2) \in \bigcap_{v \preceq v_2} \{v_1 \in \vartheta_1 \mid \sigma(v) \preceq v_1\} & (1) \\ \sigma(v_2) \in \bigcap_{v_2 \preceq v} \{v_1 \in \vartheta_1 \mid v_1 \preceq \sigma(v)\} & (2) \end{cases}$$

So let $v_1 \in \vartheta_1$ and $v_2 \in \vartheta_2$ be two distinct variables such that $v_1 \preceq v_2$. By property (1), choosing $v = v_1$, we deduce that $\sigma(v_2)$ belongs to the set $\{v_1' \in \vartheta_1 \mid \sigma(v_1) \preceq v_1'\}$, or else, $\sigma(v_1) \preceq \sigma(v_2)$. Similarly, assuming that $v_2 \preceq v_1$, we deduce that $\sigma(v_2) \preceq \sigma(v_1)$. Now, let v_2 and v_2' be two distinct variables in ϑ_2 such that $v_2 \preceq v_2'$. By rule (2), choosing $v = v_2'$, we deduce that $\sigma(v_2)$ belongs to the set $\{v_1' \in \vartheta_1 \mid v_1' \preceq \sigma(v_2')\}$, from which we conclude that $\sigma(v_2) \preceq \sigma(v_2')$, which finally shows that σ is a solution. ■