Primitive subtyping $\wedge$ implicit polymorphism $\models$ object-orientation

Extended abstract

François Bourdoncle

Centre de Mathématiques Appliquées,
Ecole des Mines de Paris,
60, boulevard Saint-Michel,
F-75014 Paris, France

bourdoncle@cma.ensmp.fr

Stephan Merz

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

merz@informatik.tu-muenchen.de

1 Overview

We present a new predicative and decidable type system called $\mathrm{ML}_{\leq}$ that adds primitive subtyping to the ML type system [17]. Our goal is to devise a type discipline for a strongly typed, higher-order object-oriented language, building on well-understood concepts from type systems for functional languages, rather than on special calculi [1] or resorting to "ad-hoc" polymorphism [19, 23], second-order systems [6, 21], or recursive types [4]. Instead, our system is based on the following ideas. We insist on a separation of interface and implementation. Interfaces define types and method signatures, where subtyping provides for a hierarchical, user-extensible modelling. Implementation modules may define data types (in the form of tagged records), which form the leaves of the type hierarchy, as well as method implementations for specific argument types. We do not model the inheritance of implementations (which we believe to be a purely syntactic notion) in the type system. We think of methods as sets of functions, similar in spirit to languages like CLOS [7] and Cecil [5], that dispatch on the run-time types of all their arguments.

We enrich the original Hindley-Milner type system [12, 17] with *polymorphic constrained types* of the form $\forall\vartheta\colon \kappa.\ \theta$ where ϑ is a set of variables, κ is a constraint, and θ is a (quantifier-free) monotype. Before we define these notions in detail in section 2, we would like to illustrate the use of our type system at the hand of the classical example of points in the two-dimensional plane, expressed in a fictitious language with full type annotations (see figure 1). We do not study type inference in this paper, although we will come back to this issue in the conclusion.

The interface *Point* declares a zero-ary *type constructor class* Point as well as a type constructor point. Type constructors are used to form complex type expressions; we identify a zero-ary type constructor such as point with the type $\mathsf{point}[\,]$. Type constructor classes partition the universe of type constructors into disjoint subuniverses. In particular, we consider two types incomparable if their outermost type constructors belong to different classes. In OO jargon, one would call the type point an "abstract class", because the interface does not define an implementation for points. The interface *Point* specifies three methods *dist*, *move*, and *leq* that operate on objects of type point. Method *dist* has type $\mathsf{point} \to \mathsf{real}$, that is, it expects an argument of type point and returns a real number[1]. As we shall see, this type is equivalent to the polymorphic constrained type $\forall\alpha\colon \alpha \leq \mathsf{point}.\ \alpha \to \mathsf{real}$, that is, method *dist* may be applied to any subtype of type point, as it should be when object hierarchies are exten-

[1] In this example, we assume predefined types real and bool.

```
interface Point is
  class Point[];
  type point: Point;
  meth dist: point → real;
  meth move: ∀α: α ≤ point. α → α;
  meth leq: (point, point) → bool
end Point;
module Point is
  meth leq(p: point, q: point) =
    (dist p) ≤ (dist q)
end Point;
```

```
module CPoint is
  open Point;
  data cpt[]: Point is
     x, y: real end;
  order cpt ⊏ point;
  meth dist(p: cpt) =
    sqrt((cpt.x p)^2 + (cpt.y p)^2);
  meth move(p: cpt) =
    cpt (1 + (cpt.x p))
        (1 + (cpt.y p));
  end CPoint;
```

```
module PPoint is
  open Point;
  data ppt[]: Point is
     d, a: real end;
  order ppt ⊏ point;
  meth dist(p: ppt) = (ppt.d p);

  meth move(p: ppt) =
    ppt (1 + (ppt.d p))
        (ppt.a p);
  end PPoint;
```

Figure 1: Points in the plane

sible.

Method *move* similarly expects some type below point, but returns a result of the argument type. In some object-oriented languages, a similar type can be specified using type specifiers such as like self. Note that recursive types are necessary to give a satisfactory typing for the method *move* in the "methods-as-records" paradigm. We shall see in section 2 that the type of *move* is a strict subtype of the type point → point in the system $\mathrm{ML}_{\leq}$. Moreover, we can also give precise typings for methods that expect several arguments, because our methods dispatch on all their arguments. Finally, the method *leq* compares two point objects—even if they belong to different subtypes of type point. In fact, the type given to *leq* is equivalent to

$$\forall\alpha,\beta\colon \alpha \leq \mathsf{point} \wedge \beta \leq \mathsf{point}.\ (\alpha,\beta) \rightarrow \mathsf{bool}$$

Along with the interface, we also define an implementation module *Point* that provides a default implementation of the method *leq* based on the method *dist* whose existence is specified in the interface.

Modules *CPoint* and *PPoint* provide two different implementations of the interface *Point*, one using Cartesian and the other polar coordinates. The implementations first define zeroary data type constructors cpt and ppt below type constructor point. Data types not only define tagged records, which can be instantiated at run-time (as in ML), but also type constructors (as opposed to ML). The only relationship that exists between the data types cpt and ppt is the fact that they share a common supertype. In fact, we require data type constructors to be minimal elements in the hierarchy of type constructors. For notational conciseness, we assume that the definition of a data type implicitly defines corresponding constructor and accessor functions for the record components.

The modules *CPoint* and *PPoint* go on to provide implementations of the methods *dist* and *move* specified in the interface. Notice that we need not give new implementations for method *leq*, because its implementation in module *Point* is effectively defined on the entire domain of *leq*. Dynamic binding of methods ensures that the correct implementations of *dist* are used whenever *leq* is evaluated.

The requirement that data type constructors are minimal ensures that methods like *move* are well-typed. For assume we were allowed to define a subconstructor scpt ⊏ cpt of cpt in some further module. Because we insist that subtypes should be acceptable whenever their supertypes are, we should be able to pass an object of type scpt to the function *move* defined in module *CPoint*—which would then return an object of type cpt, violating the type specification of method *move* in the interface *Point*.

Finally, we would like to emphasize that methods are first-order in our system. Specifically, they can be passed as function arguments. For instance, suppose that we have a polymorphic sorting method *sort* of type

$$sort\colon \forall\alpha.\ \mathsf{list}[\alpha] \rightarrow ((\alpha,\alpha) \rightarrow \mathsf{bool}) \rightarrow \mathsf{list}[\alpha]$$

that expects a list and a comparison function and returns a list of the same element type. It is then perfectly acceptable to apply *sort* to a list of points. The capability of defining polymorphic higher-order functions has proven to be a powerful tool in functional programming; we inherit it for free from the ML type system, which is a special case of our type system. In contrast to ML, we allow methods to have different implementations for different argument types.

A complete exposition, including all proofs that had to be omitted due to space restrictions, can be found in [3].

2 Constrained types

A *type constructor class* $C = (N, T, D, \sqsubseteq, \partial)$ is given by a name N, a finite and non-empty set T of *type constructors*, a set $D \subseteq T$ of *data type constructors*, a partial order $\sqsubseteq$ on T such that the data type constructors are minimal with respect to $\sqsubseteq$, and a *variance* ∂, given as a tuple of elements of the set $\{\oplus, \ominus, \otimes\}$ that denote, respectively, positive variance, negative variance, and non-variance. The arity of a class is the length of its variance. A *type structure* is a finite set $\mathcal{T}$ of type constructor classes with distinct names and pairwise disjoint sets of type constructors. 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 $\rightarrow$. The set $\mathcal{G}_{\mathcal{T}}$ of *ground monotypes over* $\mathcal{T}$ 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 over $\mathcal{T}$. The *standard ordering* $\leq$ is the smallest relation on $\mathcal{G}_{\mathcal{T}}$ such that $t_C \sqsubseteq_C t'_C$ and $\Theta_C \leq_C \Theta'_C$ imply $t_C[\Theta_C] \leq t'_C[\Theta'_C]$, where the relation $\leq_C$ on monotype lists is defined as the componentwise ordering induced by the variance of C. For example, if C has the variance $(\oplus, \ominus, \otimes)$, then $(\theta_1, \theta_2, \theta_3) \leq_C (\theta'_1, \theta'_2, \theta'_3)$ holds if and only if $\theta_1 \leq \theta'_1$, $\theta'_2 \leq \theta_2$, and $\theta_3 = \theta'_3$ all hold, where we take the latter identity to mean that both $\theta_3 \leq \theta'_3$ and $\theta'_3 \leq \theta_3$ hold. The definition of the standard ordering ensures that $\leq$ is strictly structural: in particular, the outermost type constructors of the monotypes θ and θ' are of the same class whenever $\theta \leq \theta'$ holds.

C-constructors	ϕ_C	$::=$	$t_C \mid v_C$
Monotypes	θ	$::=$	$\phi_C[\Theta_C] \mid v$
Constraints	κ	$::=$	$\phi_C \sqsubseteq \phi_C \mid \theta \leq \theta \mid \kappa \wedge \kappa$

Figure 2: Monotypes and constraints

In programming terms, a type structure is determined by class, type, and ordering declarations in modules. Our goal is to model object-orientation; we must therefore provide for the extension of type structures. Adding new type constructor classes is never a problem (assuming that there are no name clashes), because type constructors of different classes are completely unrelated to each other. When new type constructors are added to existing classes, one has to be careful to preserve the ordering on existing type constructors. We thus define *admissible extensions* of type structures much as conservative extensions are defined for models of algebraic specifications [3].

Ground monotypes are not enough to model polymorphism. We therefore assume given countable and pairwise disjoint sets of *type variables* $v, v', \ldots$, and, for each class C, *type constructor variables* $v_C, v'_C, \ldots$. In informal exposition, we do not distinguish between type and type constructor variables, but assume a universal set of variables $\alpha, \beta, \gamma, \ldots$. We define arbitrary (not necessarily ground) *monotypes* to be either a single type variable v, or recursively be built from type constructors, type constructor variables, and monotypes. The formation rules are formally given in figure 2, where we let Θ_C denote a list of monotypes whose length agrees with the arity of class C.

Figure 2 defines a *constraint* κ to be a conjunction of inequations $\phi_C \sqsubseteq \phi'_C$ between type constructors (or type constructor variables) and $\theta \leq \theta'$ between monotypes. We identify a constraint with the set of its conjuncts and use *true* to denote the trivial constraint $\mathsf{unit} \sqsubseteq \mathsf{unit}$.

A *type* τ is of the form $\forall \vartheta \colon \kappa.\, \theta$ where ϑ is a variable set, κ is a constraint, and θ is a monotype. Constrained types provide a flexible and expressive notation for types. For example, a type like $\forall \alpha \colon \mathsf{int} \leq \alpha \wedge \alpha \leq \mathsf{real}.\ (\alpha, \alpha) \rightarrow \alpha$

may be a very precise typing of the subtraction method; it subsumes both types $(\mathsf{int}, \mathsf{int}) \to \mathsf{int}$ and $(\mathsf{real}, \mathsf{real}) \to \mathsf{real}$, neither of which is a subtype of the other. However, many syntactically different constrained types can have the same meaning. For example, we would expect to identify the types $\forall\alpha\colon \mathsf{cpt} \le \alpha \wedge \alpha \le \mathsf{cpt}.\ \alpha$ and $\forall\emptyset\colon \mathit{true}.\ \mathsf{cpt}$ (which we also write as $\forall\,\emptyset.\ \mathsf{cpt}$). It is therefore important to formally identify equivalent types in a type system. We propose to do this by defining a subtyping relation between types. In most predicative type systems, subtyping is implicitly defined in terms of instantiations: a polytype is "below" all its ground instances. In particular, two ML types are equivalent if and only if they are equal up to renaming of bound variables.

For constrained types, we want a similar subtyping relation, which should moreover be compatible with the standard ordering on ground monotypes. For example, since $\tau \equiv \forall\alpha\colon \mathsf{cpt} \le \alpha \wedge \alpha \le \mathsf{cpt}.\ \alpha$ and $\forall\,\emptyset.\ \mathsf{cpt}$ should be equivalent, and $\forall\,\emptyset.\ \mathsf{cpt}$ should obviously be a subtype of $\forall\,\emptyset.\ \mathsf{point}$, τ itself should be a subtype of $\forall\,\emptyset.\ \mathsf{point}$, although point does not satisfy τ's constraint. We find it helpful to intuitively think of a type $\forall\vartheta\colon \kappa.\ \theta$ as the *upper ideal closure* (w.r.t. the standard ordering on ground monotypes) of the set of ground instances of θ that satisfy the constraint κ. Hence, the type τ defined above would denote the set $\{\mathsf{cpt}, \mathsf{point}\}$ in the type structure defined by the program of figure 1, and so would the type $\forall\,\emptyset.\ \mathsf{cpt}$. Taking the upper closure of the ground solutions ensures that subtypes may be substituted for their supertypes. For example, we want method *leq* to be applicable to objects of type $\forall\,\emptyset.\ \mathsf{cpt}$.

Unfortunately, in general this intuition is not quite correct, because it does not account for possible extensions of the current type structure in object-oriented programming. For example, consider the types $\forall\,\emptyset.\ \mathsf{point}$ and $\forall\alpha\colon \alpha \le \mathsf{point}.\ \alpha$ in the type structure defined by the interface *Point* of figure 1 alone. Although both types have the same solutions, namely just point, in this type structure, they should not be considered equivalent. In general, the denotation of the latter type will be larger than the denotation of the former one.

Formally, assuming that ϑ_1 and ϑ_2 are disjoint, we say that $\tau_1 \equiv \forall\vartheta_1\colon \kappa_1.\ \theta_1$ is a subtype of $\tau_2 \equiv \forall\vartheta_2\colon \kappa_2.\ \theta_2$ if for every admissible extension $\mathcal{T}^*$ of the given type structure $\mathcal{T}$ and every ground substitution σ_2 such that $\kappa_2[\sigma_2]$ holds in $\mathcal{T}^*$, there exists a ground substitution σ_1 that agrees with σ_2 on the variables that are free in τ_2 such that $(\kappa_1 \wedge \theta_1 \le \theta_2)[\sigma_1]$ holds in $\mathcal{T}^*$. The requirement that this condition holds for any admissible extension ensures that well-typed programs remain well-typed when the type structure is extended.

In order to study the ordering on types, we define a more general notion of *constraint implication*. We will show below how the ordering on types may be defined in terms of constraint implication. We say that a constraint κ implies a constraint κ' for all ϑ, written $\forall\vartheta.\ \kappa \models \kappa'$ if for every admissible extension $\mathcal{T}^*$ of type structure $\mathcal{T}$ and every solution σ_1 of κ in $\mathcal{T}^*$ there exists a solution σ_2 of κ' in $\mathcal{T}^*$ that agrees with σ_1 on the variables in ϑ. We propose to axiomatize the implication of constraints by the rules of figure 3 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, d_C denotes a data type constructor of class C and we let $\theta_1 \simeq \theta_2$ denote either of $\theta_1 \le \theta_2$ or $\theta_2 \le \theta_1$. $\mathcal{S}(\vartheta)$ denotes the set of *ϑ-substitutions*, which map type variables to monotypes and C-constructor variables to type constructors in T_C, and which are the identity on variables in ϑ. Finally, we write $\kappa\{\kappa'\}$ to denote that κ' is a subconstraint of κ, and $\theta = \theta'$ for the constraint $\theta \le \theta' \wedge \theta' \le \theta$.

In particular, we say that a constraint κ is *well-formed* if $\forall\emptyset.\ \mathit{true} \models \kappa$ holds, and that two constraints κ_1 and κ_2 are *ϑ-equivalent* if they imply each other for all ϑ.

The following theorem shows that the axiomatization of figure 3 captures our intentions:

Theorem 1 *Let ϑ be a variable set, and κ_1 and κ_2 be two well-formed constraints. Then $\forall\vartheta.\ \kappa_1 \models \kappa_2$ is derivable 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 $\mathcal{T}^*$.*

For computational purposes, it is helpful to observe that rules *VElim* and *MElim* can be

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

Figure 3: Constraint implication

used to reduce any well-formed constraint to a constraint $\kappa \equiv \kappa_S \wedge \kappa_B$ in "normal form" where the "substitution part" κ_S contains only subconstraints of the form $v = \theta$ for type variables v in ϑ and monotypes θ (for a given variable set ϑ), and the "base part" κ_B contains only subconstraints of the form $v \leq v'$ or $\phi_C \sqsubseteq \phi'_C$ that do not contain complex monotypes. We show in [3] how this normal form helps to prove that constraint implication is decidable.

Theorem 2 *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.*

It follows from the results of [22] that deciding well-formedness of constraints (and therefore also constraint implication) is PSPACE-hard. In particular, conversion to the normal form described above may in general entail an exponential growth in the size of the constraint. Nevertheless, in [3] we give an algorithm (inspired by an incomplete algorithm given by Fuh and Mishra in [9]) that we expect to be polynomial for "real-world" programs.

We can now define the ordering on types in terms of constraint implication. In practice, we work relative to a *constraint context* $\Delta = (\vartheta : \kappa)$. The use of the constraint context will become clear when we explain type checking in section 3. It is used to store type information for symbols defined in the context of the current declaration; intuitively, it asserts the existence of variables ϑ that satisfy constraint κ. We say that a type $\tau_1 \equiv \forall\vartheta_1 : \kappa_1.\ \theta_1$ is *well-formed* w.r.t. Δ if ϑ and ϑ_1 are disjoint, all variables that appear in κ_1 or θ_1 are from $\vartheta \cup \vartheta_1$, and $\forall\vartheta.\ \kappa \models \kappa_1$ holds. For two types $\tau_1 \equiv \forall\vartheta_1 : \kappa_1.\ \theta_1$ and $\tau_2 \equiv \forall\vartheta_2 : \kappa_2.\ \theta_2$ that are well-formed w.r.t. Δ, we say that τ_1 is a subtype of τ_2 w.r.t. Δ if

$$\forall\vartheta, \vartheta_2.\ \kappa \wedge \kappa_2 \models \kappa_1 \wedge \theta_1 \leq \theta_2$$

holds, assuming that ϑ_1 and ϑ_2 are disjoint. This definition of type ordering generalizes the instance relation between typing statements defined by Mitchell [18], which requires equality on the right-hand side.

As an example, let us prove that the type $\forall\alpha : \alpha \leq \mathsf{point}.\ \alpha \to \alpha$ of the method *move* of figure 1 is a subtype of $\forall\,\emptyset.\ \mathsf{point} \to \mathsf{point}$ in the empty context. We have to prove

$$\forall\emptyset.\ \mathit{true} \models \alpha \leq \mathsf{point} \wedge \alpha \to \alpha \leq \mathsf{point} \to \mathsf{point}$$

which follows by rule *VIntro* from

$$\begin{aligned}\forall\emptyset.\ \mathit{true} \models\ & \mathsf{point} \leq \mathsf{point} \\ & \wedge\ \mathsf{point} \to \mathsf{point} \leq \mathsf{point} \to \mathsf{point}\end{aligned}$$

On the other hand, $\forall\,\emptyset.\ \mathsf{point} \to \mathsf{point}$ is not a subtype of $\forall\alpha : \alpha \leq \mathsf{point}.\ \alpha \to \alpha$, so the latter type is a strict subtype of the former one. For otherwise, we would have to show

$$\forall\alpha.\ \alpha \leq \mathsf{point} \models \mathsf{point} \to \mathsf{point} \leq \alpha \to \alpha$$

which (by rule *MElim*) amounts to proving

$$\forall\alpha.\ \alpha \le \mathsf{point} \models \alpha = \mathsf{point}$$

which is obviously not derivable.

In the report [3] we prove that the definition of type ordering ensures that, as in ML, any well-formed type is a subtype of all its ground instances, and that the type ordering is compatible with the standard order on ground monotypes. Moreover, we obtain a lattice-like structure on types, as shown by the following theorem. This result is important to ensure minimal typing of, for instance, conditional expressions.

Theorem 3 *Assume that $\tau_1 \equiv \forall\vartheta_1 \colon \kappa_1.\ \theta_1$ and $\tau_2 \equiv \forall\vartheta_2 \colon \kappa_2.\ \theta_2$ are well-formed types that have a common supertype. Then τ_1 and τ_2 have a least upper bound $(\tau_1 \vee \tau_2)$ which is equivalent to*

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

if ϑ_1 and ϑ_2 are disjoint and v is a fresh type variable.

Of particular interest are functional types of the form $\forall\vartheta \colon \kappa.\ \theta \to \theta'$. We show in [3] that it is possible to identify functional types with monotonic type transformers (w.r.t. the subtype relation) that provide an abstract representation of the input-output relation of a function. We also show that one can define the domain $dom_\Delta(\tau)$ of a functional type, written as $\exists\vartheta \colon \kappa.\ \theta$, with a theory dual to that of types. (We use existential quantification to emphasize the duality of types and domains, our notation should not be confused with existential types as proposed for example in [16].) We say that a type $\tau \equiv \forall\vartheta_1 \colon \kappa_1.\ \theta_1$ belongs to a domain $\delta \equiv \exists\vartheta_2 \colon \kappa_2.\ \theta_2$ in context $\vartheta \colon \kappa$ if for every admissible extension there exist instances of θ_1 and θ_2 that satisfy κ_1 and κ_2, respectively, such that θ_1 is below θ_2. Again, we use constraint implication for the formal definition of domain membership, and say that type τ belongs to domain δ in context $\vartheta \colon \kappa$ if

$$\forall\vartheta.\ \kappa \models \kappa_1 \wedge \kappa_2 \wedge \theta_1 \le \theta_2$$

holds, assuming that ϑ_1 and ϑ_2 are disjoint. This definition ensures that every subtype of τ belongs to δ if τ does, and that τ belongs to every superdomain of δ. Hence, subtypes may be substituted for supertypes as function arguments, as required by the notion of substitutivity in object-oriented languages.

A functional type $\tau \equiv \forall\vartheta \colon \kappa.\ \theta \to \theta'$ can be interpreted as a type transformer if we define the application $app_\Delta(\tau, \tau_1)$ for type $\tau_1 \equiv \forall\vartheta_1 \colon \kappa_1.\ \theta_1$ as

$$app_\Delta(\tau, \tau_1) = \forall\vartheta, \vartheta_1 \colon \kappa \wedge \kappa_1 \wedge \theta_1 \le \theta.\ \theta'$$

again assuming that ϑ and ϑ_1 are disjoint. For example,

$$\begin{aligned} & app_\Delta(\forall\alpha \colon \alpha \le \mathsf{point}.\ \alpha \to \alpha, \forall\emptyset.\ \mathsf{cpt}) \\ = \quad & \forall\alpha \colon \alpha \le \mathsf{point} \wedge \mathsf{cpt} \le \alpha.\ \alpha \end{aligned}$$

which is equivalent to $\forall\emptyset.\ \mathsf{cpt}$ w.r.t. the type ordering. The definition of the operator app_Δ ensures that it is covariant in both arguments, which is at the heart of a proof of the subject reduction theorem for the language defined in the following section.

3 Type-checking

We consider an explicitly typed functional language based on the type system introduced in section 2 and give rules to type-check expressions of this language. A program (interface or module) consists of a set of declarations of type constructor classes and types, together with the implementation of every declared data type. These declarations serve to define the type structure $\mathcal{T}$; they may be mutually recursive. We will not consider them in detail, but see figure 1 for an example. The second part of a program is an expression e which must type check w.r.t. $\mathcal{T}$. Figure 4 introduces the constructs of the language together with the corresponding typing rules. A typing context is a pair $(\Delta; \Gamma)$ where Δ is a constraint context and Γ is a list of bindings of the form $x : \tau$ for expression variables. An expression of the form $\rho(d_C; \vartheta_C; x_1, \ldots, x_n)$ is a record expression; it can (essentially) only appear in the implementation of a data constructor

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

As in section 1, we assume that the declaration of any data constructor also defines corresponding extractors $d_C.i$.

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

$$\Delta; \Gamma \vdash \rho(d_C; \vartheta_C; x_1, \ldots, x_n)\colon (\forall \emptyset.\, d_C[\vartheta_C]) \quad [\mathit{Rec}] \qquad \Delta; \Gamma \vdash d_C.i\colon (\forall \vartheta_C.\, d_C[\vartheta_C] \to d_C^i\langle \vartheta_C\rangle) \quad [\mathit{Prj}]$$

$$\frac{\Delta; \Gamma \vdash e\colon \tau \qquad \Delta \vdash \tau \leq \tau'}{\Delta; \Gamma \vdash e\colon \tau'}\,[\mathit{Sub}] \qquad \frac{\Delta; \Gamma \vdash e_1\colon \tau_1 \qquad \Delta; \Gamma[x_1\colon \tau_1] \vdash e_0\colon \tau_0}{\Delta; \Gamma \vdash (\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ e_0\ \mathsf{end})\colon \tau_0}\,[\mathit{Let}]$$

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

$$\frac{\Delta; \Gamma \vdash e\colon \tau \qquad \Delta; \Gamma \vdash e'\colon \tau' \qquad \Delta \vdash \tau' \in \mathit{dom}_\Delta(\tau)}{\Delta; \Gamma \vdash (e\ e')\colon \mathit{app}_\Delta(\tau, \tau')}\,[\mathit{App}]$$

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

$$\frac{\begin{array}{c}\delta = \exists \vartheta\colon \kappa.\, \theta \qquad \pi_i = \exists \vartheta_i.\, \theta_i \qquad \pi_1, \ldots, \pi_n \text{ is a partition of } \delta \text{ w.r.t. } \Delta \\ \Delta[v, \vartheta, \vartheta_i\colon \kappa \wedge v \leq \theta, \theta_i]; \Gamma[x\colon \forall \emptyset.\, v] \vdash e_i\colon (\forall \emptyset.\, \theta') \quad (1 \leq i \leq n,\ v\ \mathit{fresh})\end{array}}{\Delta; \Gamma \vdash (\mathsf{meth}\ \{\vartheta \mid \kappa\}\,(x\colon \theta)\colon \theta' \Rightarrow [\pi_1 \Rightarrow e_1; \ldots; \pi_n \Rightarrow e_n])\colon (\forall \vartheta\colon \kappa.\, \theta \to \theta')}\,[\mathit{Meth}]$$

Figure 4: Typing rules

The subsumption rule *Sub* and the application rule *App* are reminiscent of rules in $F_\leq$; standard ML-like type systems leave subtyping implicit in the non-deterministic search for instantiations and do not define the notion of application of a (functional) type to another one. The difference is that type application is approximated in $ML_\leq$ but exact in $F_\leq$; exactness may be lost if type variables are shared between record fields in data type implementations [3].

The rules *Let* and *LetRec* are standard. In the rule *Fun* for function declarations, the function body e is type-checked in a context where the parameter x is known to have type $\forall \emptyset.\, \theta$ for some instantiation of ϑ such that κ holds. Note that only the domain, but not the result type of a function needs to be given explicitly. In fact, this is the reason for keeping function declarations in the language, which could otherwise be subsumed by the following rule for method declarations.

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 *pattern* π_i is a special kind of domain, either $\exists v.\, v$ or $\exists \vartheta_C.\, d_C[\vartheta_C]$ for a type constructor d_C. We think of a method as a set of functions, one for each pattern π_i, whose type is a subtype of the method type, restricted to the domain π_i. In contrast to ML patterns, which may be complex, the present definition of $ML_\leq$ patterns allows for dynamic dispatch according to the outermost type constructor only. Rule *Meth* defines the type of method m as the explicitly given type $\forall \vartheta\colon \kappa.\, \theta \to \theta'$, provided that two conditions are met. First, the set of patterns must be a *partition* of the method's domain δ. This condition ensures the absence of "method not understood" errors at run-time, as well as the existence of a most specific pattern for every type in the domain of the method (see [3] for a precise definition of partition). In the presence of a module system as in figure 1, this condition must be checked at link time. Second, the body e_i of each alternative must have type $\forall \emptyset.\, \theta'$ in the context where x is known to have both type θ and type θ_i.

The following theorem shows that $ML_\leq$ has decidable minimal typing.

Theorem 4 *Let* $(\Delta; \Gamma)$ *be a well-formed typing context and* e *be a well-formed expression w.r.t.* $(\Delta; \Gamma)$. *It is decidable whether* e *is well-typed in* *the context* $(\Delta;\Gamma)$. *If* e *is well-typed, it has a minimal type and this minimal type can be effectively determined.*

4 Conclusion

Our interest in this paper has been to enhance the standard Hindley-Milner type system for an explicitly typed version of ML so that it can be used as a basis for a higher-order object-oriented language. We have also defined a small programming language and given typing rules in the system $\mathrm{ML}_{\leq}$. In the report [3], we define a strict operational semantics for this language and prove a subject reduction theorem. This semantics performs dynamic dispatch on run-time types, as opposed to ML-style pattern matching, which is performed on run-time values. We have not addressed type inference in this paper. Nevertheless, we conjecture that standard type inference techniques such as described in [2, 8, 9, 10, 13, 18, 20] can be used to infer the types of untyped $\mathrm{ML}_{\leq}$ programs except for method declarations. It is an open question whether or not methods have principal types in this context.

The report [3] discusses several extensions to the base language presented here. In particular, we propose a module system, show that multi-methods are more expressive than the single-argument methods we have presented in this paper, and discusses type classes. Although some of the uses of type classes can be achieved in the $\mathrm{ML}_{\leq}$ type system, type classes are still desirable since our notion of subtyping is strictly structural: type constructors of different classes are incomparable. We also discuss some syntactic sugar, for instance, to obtain a form of implementation inheritance.

$\mathrm{ML}_{\leq}$ has some similarity with $\mathrm{F}_{\leq}$. However, $\mathrm{F}_{\leq}$ is undecidable, impredicative, and relies on explicit polymorphism, that is, types are passed as arguments to functions. In contrast, $\mathrm{ML}_{\leq}$ is a decidable and predicative system for implicit polymorphism where run-time types are used for dynamic binding[2].

[2] The operational semantics of $\mathrm{ML}_{\leq}$ defined in [3] does not keep complete type information, but only tags run-time values with the outermost type constructor of their dynamic type, in order to perform dynamic dispatch.

$\mathrm{ML}_{\leq}$ has strong connections with many other systems derived from the Hindley-Milner type system, in particular, systems of overloaded functions [14, 15, 23]. As indicated above, we believe that type classes are orthogonal to our system. They cannot be used to obtain typing of mixed operations like *leq*, unless multi-parameter type classes are used, but then type-checking becomes undecidable in general. Kaes' type system [15] is probably closest in spirit to $\mathrm{ML}_{\leq}$; it offers even more expressive types, and provides type inference. On the other hand, Kaes does not address methods and dynamic dispatch, nor does he provide an operational semantics. The report [3] gives detailed comparisons with these and other type systems such as dynamics and record-based models, and indicates connections that deserve further study.

References

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

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

[3] F. Bourdoncle, S. Merz: On the integration of functional programming, class-based object-oriented programming, and multi-methods. Technical Report 26, Centre des Mathématiques Appliquées, Ecole des Mines de Paris, March 1996. Available on the WWW via `http://www.ensmp.fr/~bourdonc/`

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

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

[6] 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)

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

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

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

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

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

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

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

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

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

[16] J. Mitchell, G. Plotkin. Abstract types have existential type. ACM Transactions on Programming Languages and Systems 10 (3) (1988)

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

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

[19] 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

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

[21] B. C. Pierce, D. N. Turner. Simple type-theoretic foundations for object-oriented programming. Journal of Functional Programming 4 (2) (1994) 207–247

[22] J. Tiuryn. Subtype Inequalities. Proceedings of the Seventh Symposium on Logic in Computer Science (1992) 308–315

[23] 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