A Delegation-based Object Calculus with Subtyping

Kathleen Fisher* and John C. Mitchell**

Computer Science Department, Stanford University, Stanford, CA 94305
{kfisher,mitchell}@cs.stanford.edu

Abstract. This paper presents an untyped object calculus that reflects the capabilities of so-called delegation-based object-oriented languages. A type inference system allows static detection of errors, such as *message not understood,* while at the same time allowing the type of an inherited method to be specialized to the type of the inheriting object. The main advance over previous work is the provision for subtyping in the presence of delegation primatives. This is achieved by distinguishing a prototype, whose methods may be extended or replaced, from an object, which only responds to messages for which it already has methods. An advantage of this approach is that we have full subtyping without restricting the "run-time" use of inheritance. Type soundness is proved using operational semantics and an analysis of typing derivations.

1 Introduction

There are two basic forms of object-oriented languages, the most common of which is class-based. In these languages, which include C++ [Str86, ES90], Smalltalk [GR83] and Eiffel [Mey92], an object is created by a class, which is a construct that creates objects that share a common representation but which may have different specific data values. In typed class-based languages, it is common for all objects created by a given class to have the same type. An alternate approach, advanced in the language Self [US91, CU89], allows objects to be created directly or by cloning (copying) existing objects. In class-based languages, inheritance is a mechanism for defining one class from another. In delegation-based languages, inheritance is a mechanism for reusing part of one object to define another.

Following our previous work [Mit90, FHM94], the object calculus presented in this paper is delegation-based. Initially, we regarded the delegation-based approach as a technically simpler method for analyzing the same underlying language concepts. This appeared correct for the study of method specialization carried out in [Mit90, FHM94], but in [FM94] we observed that there appeared to be a fundamental trade-off between delegation and subtyping. Specifically,

* Supported in part by an NSF Graduate Fellowship, NSF Grant CCR-9303099, and a Fannie and John Hertz Foundation Fellowship.

** Supported in part by NSF Grant CCR-9303099 and the TRW Foundation.

if an object may be extended with new methods, then it is important to know at compile-time that certain methods have *not* been defined already. This requirement conflicts with the usual motivation for subtyping, which is to allow code to operate uniformly over all objects having some minimum set of required methods. (Similar and related observations appear in [AC94]; see Section 3.1.)

In this paper, we present one way of resolving this conflict. Intuitively, the main idea is to distinguish between objects that may be extended with additional methods (or have existing methods redefined) and those that cannot. This distinction is achieved by giving different uses of objects different types. In other words, an object may be created, and then have new methods added or existing methods redefined. At this point, only trivial subtyping may be used because the type system must keep track of exactly the set of methods associated with the object. However, such an object may be "converted" to a different kind of object, whose methods can no longer be altered. This conversion is done by changing the type of the object to a form which has the expected subtyping properties. In this way, we allow both delegation and subtyping, at the cost of some increase in the complexity of the type system. An alternate way of extending our previous calculus with subtyping, developed in [BL95], is described in Section 6.

The two forms of object type used in this paper highlight the distinction between two interfaces: the inheritance-interface and the client-interface of an object. This distinction is essentially familiar from object-oriented languages and databases, but often not explicitly mentioned in language documentation. If we write a C++ class such as *stack,* then there are really two separate ways of using this class. The first is by calling the constructor of the class to create *stack* objects and then calling their member functions. The second use of the *stack* class is by defining a derived class. (This "reuse" of implementation is traditionally called inheritance.) One way to see that these are very different uses of a class is to notice that they induce very different notions of class equivalence. If we just want the objects constructed by a class to behave in the same way, then we can perform a number of optimizations and program transformations. However, some transformations that preserve the behavior of constructed objects would observably change the behavior of derived classes. A simple example occurs when a class has two mutually recursive member functions, say f and g. If we replace these functions by two equivalent non-recursive functions, we do not change the behavior of constructed objects. However, this transformation may radically alter the behavior of a derived class if both are virtual functions and one is redefined in the derived class. An innovation of our object calculus, when compared with other recent work such as [AC94, Bru93, FHM94, PT94] (summarized and compared in [FM94]), is the type distinction between these two uses of a class.

The distinction between inheritance and client interfaces leads us to distinguish *prototypes* and *objects.* Intuitively, a prototype is a collection of methods that may be used to implement one or more objects. The operations on prototypes are (i) sending a message (which results in the invocation of a method), (ii) adding a new method (method addition), and (iii) redefining an existing method without changing its type (method override). The type of a prototype is called a *pro type.* For the reasons described above, only trivial subtyping exists between *pro* types.

Intuitively, an object is created by "packaging" a prototype, an operation that does nothing except "seal" the prototype so that no methods can be added or redefined. In our calculus, the "sealing" operation only changes the type from a *pro* type to an object, or *obj type*. Although the only operations allowed on objects from the "outside" are message sends, the methods within an object can override the methods of their host object. This ability to override methods internally permits the existence of methods such as a `setX` method in a point object which replace the point's `x` method with a new one storing a new location. Preventing external method override and method extension by sealing prototypes is sufficient, however, to make interesting subtyping sound for objects. The internal redefinitions do not cause an unsoundness in the type system because they are type-checked with precise knowledge of the *pro* type of the expression to which they were added.

In addition to subtyping, the distinction between prototypes and objects provides some useful insight into the two uses of classes mentioned above. Inheritance on classes is modeled in our system by operations on prototypes. Creating an object from a class is modeled by "sealing" a prototype to an object. The distinction between *pro* and *obj* types clearly illustrates the difference between a derived class's interface to a class and a client program's view of the objects created from that class. In particular, inheritance depends on the presence *and* absence of the methods in a prototype, whereas a client is only concerned with the presence of the methods it uses.

For simplicity, we work with a functional model of objects. Recent studies such as [AC95, Bv93, Pie93] confirm that imperative objects may be treated in essentially the same framework. In addition, our objects are collections of methods; we do not distinguish "member functions" from "member data" since data may be represented by functions that always return the same result. As in [FHM94], our object calculus provides method specialization. Intuitively, method specialization means that the types of methods in classes may be specialized when new classes are created from them via inheritance. Further discussion of method specialization may be found in [FHM94, FM94].

2 Untyped calculus of objects

This calculus is described in full detail in [FHM94]. We briefly summarize it here for presentational completeness.

2.1 The Objects

The untyped object calculus is the result of adding four new object-related syntactic forms to untyped lambda calculus:

$\langle\rangle$	the empty object
$\mathtt{e} \Leftarrow \mathtt{m}$	send message m to object e
$\langle \mathtt{e}_1 \leftarrow\!\!+\, \mathtt{m}{=}\mathtt{e}_2 \rangle$	extend object $\mathtt{e}_1$ with new method m having body $\mathtt{e}_2$
$\langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_2 \rangle$	replace $\mathtt{e}_1$'s method body for m by $\mathtt{e}_2$

We consider $\langle \mathtt{e}_1 \leftarrow\!\!+\, \mathtt{m}{=}\mathtt{e}_2 \rangle$ meaningful only if $\mathtt{e}_1$ denotes an object that does not have an m method, and $\langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_2 \rangle$ meaningful only if $\mathtt{e}_1$ denotes an object that already has an m method. These conditions will be enforced by the type system. If a method is new, then no other method in the object could have referred to it, so it may have any type. On the other hand, if a method is being replaced, then we must be careful not to violate any typing assumptions in other methods that refer to it.

To simplify notation, we write $\langle \mathtt{m}_1 = \mathtt{e}_1, \ldots, \mathtt{m}_k = \mathtt{e}_k \rangle$ for $\langle \ldots \langle \langle\rangle \leftarrow\!\!+\, \mathtt{m}_1 = \mathtt{e}_1 \rangle \ldots \leftarrow\!\!+\, \mathtt{m}_k = \mathtt{e}_k \rangle$, where $\mathtt{m}_1, \ldots, \mathtt{m}_k$ are distinct method names. Using this abbreviation, we may code a one-dimensional point object with field x and method move as follows:

$$\begin{aligned} \mathtt{p} \stackrel{def}{=} \langle\ & \mathtt{x} = \lambda \mathtt{self}.\, 3, \\ & \mathtt{move} = \lambda \mathtt{self}.\, \lambda \mathtt{dx}. \langle \mathtt{self} \leftarrow \mathtt{x} = \lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} \rangle \rangle \end{aligned}$$

We refer to x as a field because it does not use the self variable.

2.2 Message Send

Sending a message $\mathtt{m}_i$ to an object containing such a method is modeled by extracting the corresponding method body $\mathtt{e}_i$ from the object and applying it to the object itself:

$$\langle \mathtt{m}_1 = \mathtt{e}_1, \ldots, \mathtt{m}_k = \mathtt{e}_k \rangle \Leftarrow \mathtt{m}_i \quad \stackrel{eval}{\longrightarrow} \quad \mathtt{e}_i\ \langle \mathtt{m}_1 = \mathtt{e}_1, \ldots, \mathtt{m}_k = \mathtt{e}_k \rangle$$

For example, sending the message move with a displacement of 2 to p, we get:

$$\begin{aligned} \mathtt{p} \Leftarrow \mathtt{move}\, 2 &= (\lambda \mathtt{self}.\, \lambda \mathtt{dx}. \langle\ \ldots\ \rangle)\ \mathtt{p}\ \ 2 \\ &= \langle \mathtt{p} \leftarrow \mathtt{x} = \lambda \mathtt{s}.(\mathtt{p} \Leftarrow \mathtt{x}) + 2 \rangle \\ &= \langle \mathtt{p} \leftarrow \mathtt{x} = \lambda \mathtt{s}.\, 3 + 2 \rangle \\ &= \langle \mathtt{p} \leftarrow \mathtt{x} = \lambda \mathtt{self}.\, 5 \rangle \end{aligned}$$

Using a sound rule for object equality,

$$\langle \langle \mathtt{m}_1{=}\mathtt{e}_1, \ldots, \mathtt{m}_k{=}\mathtt{e}_k \rangle \leftarrow \mathtt{m}_i{=}\mathtt{e}'_i \rangle \quad = \quad \langle \mathtt{m}_1{=}\mathtt{e}_1, \ldots, \mathtt{m}_i{=}\mathtt{e}'_i, \ldots, \mathtt{m}_k{=}\mathtt{e}_k \rangle$$

we may reach the conclusion

$$\begin{aligned} \mathtt{p} \Leftarrow \mathtt{move}\, 2 = \langle\ & \mathtt{x} = \lambda \mathtt{self}.\, 5, \\ & \mathtt{move} = \lambda \mathtt{self}.\, \lambda \mathtt{dx}. \langle \ldots \rangle \rangle \end{aligned}$$

showing that the result of sending a move message with an integer parameter is an object identical to p, but with an updated x-coordinate.

2.3 Inheritance

The inheritance mechanism of this calculus is very simple, although its typing is somewhat complex. To define colored points from points, we simply need to add a color method to our point object $\mathtt{p}$.

$$\mathtt{cp} \stackrel{def}{=} \langle p \longleftarrow\!\!\!+\ \mathtt{myColor} = \lambda\mathtt{self.\,blue}\rangle$$

If we send the $\mathtt{move}$ message to $\mathtt{cp}$ with the same parameter, we may calculate the resulting object in exactly the same way as before:

$$\begin{aligned}\mathtt{cp} \Leftarrow \mathtt{move}\,2 &= (\lambda\mathtt{self.}\,\lambda\mathtt{dx.}\langle\ \ldots\ \rangle)\ \mathtt{cp}\ \ 2\\ &= \langle \mathtt{cp} \leftarrow \mathtt{x} = \lambda\mathtt{s.}(\mathtt{cp} \Leftarrow \mathtt{x}) + 2\rangle\\ &= \ldots\\ &= \langle \mathtt{cp} \leftarrow \mathtt{x} = \lambda\mathtt{self.}\,5\rangle\end{aligned}$$

with the final conclusion that

$$\begin{aligned}\mathtt{cp} \Leftarrow \mathtt{move}\,2 = \langle\ & \mathtt{x} = \lambda\mathtt{self.}\,5,\\ & \mathtt{move} = \lambda\mathtt{self.}\,\lambda\mathtt{dx.}\langle\ldots\rangle,\\ & \mathtt{myColor} = \lambda\mathtt{self.\,blue}\\ \rangle &\end{aligned}$$

The important feature of this computation is that the color of the resulting colored point is the same as the original one. While $\mathtt{move}$ was defined originally for points, the method body performs the correct computation when the method is inherited by an object with additional methods. Hence we have method specialization in the calculus.

2.4 Operational semantics

In defining the operational semantics for our calculus, we must give rules for extracting and applying the appropriate method of an object. A natural way to approach this would be to use a permutation rule

$$\langle\langle \mathtt{e}_1 \leftarrow\!\!\ominus\ \mathtt{n}{=}\mathtt{e}_2\rangle \leftarrow\!\!\ominus\ \mathtt{m}{=}\mathtt{e}_3\rangle = \langle\langle \mathtt{e}_1 \leftarrow\!\!\ominus\ \mathtt{m}{=}\mathtt{e}_3\rangle \leftarrow\!\!\ominus\ \mathtt{n}{=}\mathtt{e}_2\rangle$$

where $\mathtt{m}$ and $\mathtt{n}$ are distinct and each occurrence of $\leftarrow\!\!\ominus$ may be either $\longleftarrow\!\!\!+$ or $\leftarrow$. This equational rule would let us treat objects as sets of methods, rather than ordered sequences. However, this rule would cause typing complications, since our typing rules only allow us to type object expressions when methods are added in an appropriate order. In particular, if we permute the methods of the object expression

$$\begin{aligned}&\langle\langle\langle\rangle \longleftarrow\!\!\!+\ \mathtt{x} = \lambda\mathtt{self.}\,3\rangle\\ &\quad\ \longleftarrow\!\!\!+\ \mathtt{x\_succ} = \lambda\mathtt{self.}(\mathtt{self} \Leftarrow \mathtt{x}) + 1\rangle\end{aligned}$$

then the subexpression

$$\langle\langle\rangle \longleftarrow\!\!\!+\ \mathtt{x\_succ} = \lambda\mathtt{self.}(\mathtt{self} \Leftarrow \mathtt{x}) + 1\rangle$$

is not well-typed. Therefore, the entire expression cannot be typed.

We circumvent the problem of method order by using a more complicated "standard form" for object expressions, namely,

$$\langle\langle\langle \mathtt{m}_1 = \mathtt{e}_1, \ldots \mathtt{m}_k = \mathtt{e}_k \rangle \leftarrow \mathtt{m}_1 = \mathtt{e}'_1 \rangle \ldots \leftarrow \mathtt{m}_k = \mathtt{e}'_k \rangle$$

where each method is defined exactly once, using some arbitrary method body that does not contribute to the observable behavior of the object, and then redefined exactly once by giving the desired method body. Even if the two definitions of a method are the same, this form is useful since it allows us to permute the list of method redefinitions arbitrarily. More formally, in addition to the $\xrightarrow{eval}$ relation that allows us to evaluate message sends and function applications, the operational semantics includes a subsidiary "bookkeeping" relation $\xrightarrow{book}$, which allows each object to be transformed into the "standard form" indicated above. The relation $\xrightarrow{book}$ is the congruence closure of the first four clauses listed in Table 1. These rules also allow the method redefinitions to be permuted arbitrarily. An important property of $\xrightarrow{book}$, proved in Section 5, is that if $\mathtt{e} \xrightarrow{book} \mathtt{e}'$, then any type for $\mathtt{e}$ is also derivable for $\mathtt{e}'$. This property would fail if we had the more general permutation rule discussed above.

(*switch ext ov*)	$\langle\langle \mathtt{e}_1 \leftarrow \mathtt{n}=\mathtt{e}_2 \rangle \leftarrow\!\!+\, \mathtt{m}=\mathtt{e}_3 \rangle$	$\xrightarrow{book} \langle\langle \mathtt{e}_1 \leftarrow\!\!+\, \mathtt{m}=\mathtt{e}_3 \rangle \leftarrow \mathtt{n}=\mathtt{e}_2 \rangle$
(*perm ov ov*)	$\langle\langle \mathtt{e}_1 \leftarrow \mathtt{m}=\mathtt{e}_2 \rangle \leftarrow \mathtt{n}=\mathtt{e}_3 \rangle$	$\xrightarrow{book} \langle\langle \mathtt{e}_1 \leftarrow \mathtt{n}=\mathtt{e}_3 \rangle \leftarrow \mathtt{m}=\mathtt{e}_2 \rangle$
(*add ov*)	$\langle \mathtt{e}_1 \leftarrow\!\!+\, \mathtt{m}=\mathtt{e}_3 \rangle$	$\xrightarrow{book} \langle\langle \mathtt{e}_1 \leftarrow\!\!+\, \mathtt{m}=\mathtt{e}_3 \rangle \leftarrow \mathtt{m}=\mathtt{e}_3 \rangle$
(*cancel ov ov*)	$\langle\langle \mathtt{e}_1 \leftarrow \mathtt{m}=\mathtt{e}_2 \rangle \leftarrow \mathtt{m}=\mathtt{e}_3 \rangle$	$\xrightarrow{book} \langle \mathtt{e}_1 \leftarrow \mathtt{m}=\mathtt{e}_3 \rangle$
(β)	$(\lambda \mathtt{x}.\, \mathtt{e}_1)\mathtt{e}_2$	$\xrightarrow{eval} [\mathtt{e}_2/\mathtt{x}]\mathtt{e}_1$
($\Leftarrow$)	$\langle \mathtt{e}_1 \leftarrow\!\!\circ\, \mathtt{m}=\mathtt{e}_2 \rangle \Leftarrow \mathtt{m}$	$\xrightarrow{eval} \mathtt{e}_2\langle \mathtt{e}_1 \leftarrow\!\!\circ\, \mathtt{m}=\mathtt{e}_2 \rangle$ where $\leftarrow\!\!\circ$ may either $\leftarrow\!\!+$ or $\leftarrow$.

Table 1. Bookkeeping and evaluation rules.

3 Static Type System

3.1 Subtyping and delegation

In general, there are two forms of subtyping that are applicable to object types: *width* and *depth* subtyping. If one object type provides all of the methods of another, with the same types, then every object of the first type should be safely substitutable in any context expecting an object with the second type. We call this form of subtyping *width* subtyping. It is also generally possible for one object type to restrict the type of one or more of its methods to a subtype. Again, any object with the more restricted type should be safely substitutable in any context expecting an object with the less restricted type. We call this form of subtyping *depth* subtyping.

As explained in [FM94], pure-delegation based languages, such as that described in [FHM94], do not permit either form of object subtyping. Consider the intuitive definition of a subtype: $\mathtt{A}$ is a subtype of $\mathtt{B}$ if we may use an object of type $\mathtt{A}$ in any context expecting an object of type $\mathtt{B}$. If objects of type $\mathtt{A}$ are to be used as $\mathtt{B}$'s, then $\mathtt{A}$-objects must have all of the methods of $\mathtt{B}$-objects. Because method addition is a legal operation on objects in delegation-based languages, objects with extra methods cannot be used in some contexts where an object with fewer methods may. As an example, a colored point object cannot be used in a context that will add color, but a point object can. For $\mathtt{A}$ to be a subtype of $\mathtt{B}$ then, $\mathtt{A}$'s must contain exactly the same methods as $\mathtt{B}$'s. Hence we do not get width subtyping.

We do not get depth subtyping either, as depth subtyping is unsound in the presence of method override. The following example, given in [AC94] illustrate the problem. Consider the object types $\mathtt{A}$ and $\mathtt{B}$:

$$\mathtt{B} \stackrel{def}{=} \mathbf{pro}\,\mathtt{t}.\langle\!\langle \mathtt{x} : \mathtt{Int},\ \mathtt{y} : \mathtt{Real} \rangle\!\rangle$$
$$\mathtt{A} \stackrel{def}{=} \mathbf{pro}\,\mathtt{t}.\langle\!\langle \mathtt{x} : \mathtt{PosInt},\ \mathtt{y} : \mathtt{Real} \rangle\!\rangle$$

If we allow deep record/object subtyping, $\mathtt{A} <: \mathtt{B}$ since $\mathtt{PosInt} <: \mathtt{Int}$. Now consider an object $\mathtt{a}$ defined as follows:

$$\mathtt{a} \stackrel{def}{=} \langle \mathtt{x} = \mathtt{1},\ \mathtt{y} = \mathtt{ln}(\mathtt{self} \Leftarrow \mathtt{x}) \rangle$$

We can see that $\mathtt{a}$ has type A. By the subsumption rule, we may consider $\mathtt{a}$ to have type B. With this typing, the expression $\langle \mathtt{a} \leftarrow \mathtt{x} = -\mathtt{1} \rangle$ is legal. But then sending the message $\mathtt{y}$ to $\mathtt{a}$ produces a run-time type error.

In the remainder of this section, we give a type system for the object calculus described in Section 2 that is an extension of the one given in [FHM94]. The major change to the earlier system is the addition of *obj* types, which permit subtyping in both width and depth. In both the earlier system and the one we describe here, expressions with *pro* type (called *class* type in [FHM94]) may have new methods added via method addition ($\leftarrow\!\!+$) and old methods replaced via method override($\leftarrow$). As described above, only trivial subtyping is sound for such types. In the system presented here, we may decide that we are no longer interested in modifying the methods of an expression with *pro* type. At this point, we may "seal" the expression by converting it to a value with an *obj* type. Since $\leftarrow$ and $\leftarrow\!\!+$ are not valid operations on *obj* types, width and depth subtyping are sound for these types. We call expressions given *pro* type *prototypes* because they may either be sent messages or modified to create new kinds of prototypes. Expressions with *obj* type are called *objects* since such expressions may be sent messages or subtyped, but not extended or modified. In the following, we will use the meta-variable *probj* to denote either *obj* or *pro*.

3.2 The Type System

The type defined by the type expression

$$\mathbf{probj}\,\mathtt{t}.\langle\!\langle \mathtt{m}_1 : \tau_1, \ldots, \mathtt{m}_k : \tau_k \rangle\!\rangle$$

is a type $\mathtt{t}$ with the property that when we send the message $\mathtt{m_i}$ for $1 \leq \mathtt{i} \leq \mathtt{k}$ to any element $\mathtt{x}$ of this type, the result has type τ_i. A significant aspect of this type is that the bound type variable $\mathtt{t}$ may appear in the types $\tau_1, \ldots, \tau_k$. Thus, when we say $\mathtt{x} \Leftarrow \mathtt{m_i}$ will have type τ_i, we mean type τ_i with any free occurrences of $\mathtt{t}$ in τ_i referring to the type $\mathbf{probj}\,\mathtt{t}.\langle\!\langle \mathtt{m}_1 : \tau_1, \ldots, \mathtt{m}_k : \tau_k \rangle\!\rangle$ itself. Thus, $\mathbf{probj}\,\mathtt{t}.\langle\!\langle \ldots \rangle\!\rangle$ is a special form of recursively-defined type.

The typing rule for message send has the form

$$\frac{\mathtt{e} : \mathbf{probj}\,\mathtt{t}.\langle\!\langle \ldots\, \mathtt{m} : \tau \rangle\!\rangle}{\mathtt{e} \Leftarrow \mathtt{m} : [\mathbf{probj}\,\mathtt{t}.\langle\!\langle \ldots\, \mathtt{m} : \tau \rangle\!\rangle / \mathtt{t}]\tau}$$

where the substitution for $\mathtt{t}$ in τ reflects the recursive nature of object types. This rule may be used in its *pro* form to give the point `p` with `x` and `move` methods type

$$\mathbf{pro}\,\mathtt{t}.\langle\!\langle\, \mathtt{x} : \mathtt{Int}, \mathtt{move} : \mathtt{Int} \to \mathtt{t} \rangle\!\rangle$$

since $\mathtt{p} \Leftarrow \mathtt{x}$ returns an integer and $\mathtt{p} \Leftarrow \mathtt{move}\ \mathtt{n}$ has the same type as `p`.

A subtle but very important aspect of the type system is that when an object is extended with an additional method, the syntactic type of each method does not change. For example, when we extended `p` with a color to obtain `cp`, we obtain an object with type

$$\mathbf{pro}\,\mathtt{t}.\langle\!\langle\, \mathtt{x} : \mathtt{Int}, \mathtt{move} : \mathtt{Int} \to \mathtt{t}, \mathtt{myColor} : \mathtt{Color} \rangle\!\rangle$$

The important change to notice here is that although the syntactic type of `move` is still $\mathtt{Int} \to \mathtt{t}$, the meaning of the variable `t` has changed. Instead of referring to the type of `p` as it did originally, it now refers to the type of `cp`. This behavior is the method specialization discussed briefly in Section 1. For this kind of reinterpretation of type variables to be sound, the typing rules for object extension and object override must insure that every possible type for a new method will be correct. This check is done through a form of implicit higher-order polymorphism, which is explained more fully in [FHM94].

3.3 Adding subtyping

To insure that subtyping is sound for *obj* expressions, we disable object extension and object override on them, permitting them only to receive messages. Since the methods handling such messages are defined when the expression has a *pro* type, however, it is possible for these methods to redefine themselves or other methods of the host expression. (The polymorphism requirement for method bodies prevents them from adding new methods.) These redefinitions do not cause an unsoundness in the type system, however, because they were type-checked with precise knowledge of the *pro* type of the expression to which they were added. The unsoundness discussed above arises when the type of an expression is promoted via subsumption and then a method is replaced by another method with higher type. This scenario cannot occur in our system because once we seal an expression to a potentially imprecise *obj* type, methods cannot be overridden from the outside, where only the imprecise sealed type is available. Internal methods, type-checked with precise *pro* type information about the expression to which they were added, are type-safe.

Since sealing a prototype `e` requires no change to `e`, we would have liked to be able to use the following rule in conjunction with a subsumption rule to "seal" prototypes:

$$(\textit{seal-unsound}) \qquad \frac{\Gamma \vdash \mathbf{obj}\, t \,\textbf{.}\, R : T}{\Gamma \vdash \mathbf{pro}\, t \,\textbf{.}\, R <: \mathbf{obj}\, t \,\textbf{.}\, R}$$

where the judgement $\Gamma \vdash \mathbf{obj}\, t \,\textbf{.}\, R : T$ denotes that type $\mathbf{obj}\, t \,\textbf{.}\, R$ is well formed. Unfortunately, this rule is unsound when the variable t appears contravariantly (or invariantly) in R. Hence we need a more complicated rule that has the same form as the rule for deriving subtyping judgements on *obj* types. Because of this similarity, we may combine the sealing rule and the *obj* subtyping rule to produce:

$$(<:\ \textit{object}) \quad \frac{\begin{array}{c}\Gamma \vdash \mathbf{probj}\, s_1 \,\textbf{.}\, \langle\!\langle \overrightarrow{m} : \overrightarrow{\tau},\ \overrightarrow{n} : \overrightarrow{\sigma} \rangle\!\rangle : T \\ \Gamma \vdash \mathbf{obj}\, s_2 \,\textbf{.}\, \langle\!\langle \overrightarrow{m} : \overrightarrow{\tau}' \rangle\!\rangle : T \\ \Gamma, s_1 <: s_2 \vdash \tau_i <: \tau_i' \\ s_1 \notin FV(\tau_i'),\quad s_2 \notin FV(\tau_i)\end{array}}{\Gamma \vdash \mathbf{probj}\, s_1 \,\textbf{.}\, \langle\!\langle \overrightarrow{m} : \overrightarrow{\tau},\ \overrightarrow{n} : \overrightarrow{\sigma} \rangle\!\rangle <: \mathbf{obj}\, s_2 \,\textbf{.}\, \langle\!\langle \overrightarrow{m} : \overrightarrow{\tau}' \rangle\!\rangle}$$

The meta-variable *probj* is *pro* when this rule is used to seal prototypes. When used simply for subtyping between object types, *probj* is *obj*. Notice that this rule gives us subtyping in both width and depth for *obj* types. This rule has the somewhat unfortunate consequence that we cannot seal prototypes to *obj* types containing methods that are contravariant (or invariant) in the bound type variable. However, this appears to be a fundamental trade-off and limitation.

The other rules in the type system are either routine or the same as those in [FHM94]. A complete list of the typing rules appears in Appendix A.

3.4 Rows, types, and kinds

More formally, the type expressions are given by:

Types
$$\tau \ ::= \ t \,|\, s \,|\, \tau_1 \rightarrow \tau_2 \,|\, \mathbf{pro}\, t\,\mathbf{.}\, R \,|\, \mathbf{obj}\, t\,\mathbf{.}\, R$$

Rows
$$R \ ::= \ r \,|\, \langle\!\langle\rangle\!\rangle \,|\, \langle\!\langle R \,|\, m : \tau \rangle\!\rangle \,|\, \lambda t.\, R \,|\, R\tau$$

Kinds
$$kind \ ::= \ T \,|\, \kappa$$
$$\kappa \ ::= \ T^n \rightarrow [m_1, \ldots, m_k]$$

The row expressions, which are essentially lists of method names and associated types, appear as subexpressions of type expressions. Types and rows are distinguished by kinds. As a notational simplification, we write $[m_1, \ldots, m_k]$ for $T^0 \rightarrow [m_1, \ldots, m_k]$. Intuitively, the elements of kind $[m_1, \ldots, m_k]$ are rows that do *not* include method names $m_1, \ldots, m_k$. The reason we must know statically that some method does not appear is to guarantee that methods are not multiply defined. Kinds of the form $T^n \rightarrow [m_1, \ldots, m_k]$, for $n \geq 1$, are used to infer a form of higher-order polymorphism of method bodies.

The environments, or contexts, of the system list term, type, and row variables.
$$\Gamma \ ::= \ \epsilon \,|\, \Gamma, x : \tau \,|\, \Gamma, t : T \,|\, \Gamma, r : \kappa \,|\, \Gamma, s_1 <: s_2$$
Note that contexts are ordered lists, not sets.

The judgement forms are standard:

$\Gamma \vdash *$	well-formed context
$\Gamma \vdash e : \tau$	term has type
$\Gamma \vdash \tau : T$	well-formed type
$\Gamma \vdash R : \kappa$	row has kind
$\Gamma \vdash \tau_1 <: \tau_2$	τ_1 subtype of τ_2

4 Examples

To give some intuition for this calculus, we present two examples, neither of which is typable in the system of [FHM94]. Other examples, demonstrating the expressivity of the system, appear in [FHM94]. The first example below illustrates the use of subtyping for code reuse, while the second demonstrates its use for encapsulation.

4.1 Subtyping for code reuse

Consider the function
$$\mathtt{average} \stackrel{def}{=} \lambda \mathtt{p}_1.\, \lambda \mathtt{p}_2.((\mathtt{p}_1 \Leftarrow \mathtt{x}) + (\mathtt{p}_2 \Leftarrow \mathtt{x}))/2$$

We may give this expression type:

$$\texttt{average} \;:\; \mathbf{obj}\,t.\langle\!\langle \texttt{x} : \texttt{int} \rangle\!\rangle \rightarrow \mathbf{obj}\,t.\langle\!\langle \texttt{x} : \texttt{int} \rangle\!\rangle \rightarrow \texttt{int}$$

(assuming integer division). Now consider the following two expressions:

$$\begin{aligned} \texttt{p} &\stackrel{def}{=} \langle\, \texttt{x} = \lambda\texttt{self}.\,3, \\ &\qquad \texttt{move} = \lambda\texttt{self}.\,\lambda\texttt{dx}.\,\langle \texttt{self} \leftarrow \texttt{x} = \lambda\texttt{s}.(\texttt{self} \Leftarrow \texttt{x}) + \texttt{dx} \rangle \rangle \\ \texttt{cp} &\stackrel{def}{=} \langle\, \texttt{p} \leftarrow\!\!+\, \texttt{c} = \lambda\texttt{self}.\texttt{red} \;\rangle \end{aligned}$$

As in the system in [FHM94], we may give p and cp the following types:

$$\begin{aligned} \texttt{p} &: \mathbf{pro}\,t.\langle\!\langle\, \texttt{x} : \texttt{int}, \texttt{move} : \texttt{int} \rightarrow \texttt{t} \rangle\!\rangle \\ \texttt{cp} &: \mathbf{pro}\,t.\langle\!\langle\, \texttt{x} : \texttt{int}, \texttt{move} : \texttt{int} \rightarrow \texttt{t}, \texttt{c} : \texttt{colors} \rangle\!\rangle \end{aligned}$$

In the new system, we may seal these expressions and give them both the following common supertype:

$$\texttt{p}, \texttt{cp} \;:\; \mathbf{obj}\,t.\langle\!\langle \texttt{x} : \texttt{int} \rangle\!\rangle$$

Hence we may give the expression `average p cp` type `int`, allowing us to use the function `average` to calculate the average of colored points and points interchangeably.

4.2 Subtyping for encapsulation

Once we change the type of an expression from *pro* type to *obj* type, the methods of that prototype become "read-only", including those methods that are effectively data fields. Hence if the designers of a prototype wish its users to be able to change the values of some of its methods after it is sealed, they must provide methods to change those values. This restriction provides a mechanism whereby prototype designers can guarantee invariants for the objects created from their prototypes.

Suppose, for example, we were interested in colored points whose color was guaranteed to be either blue or green. Then consider the expression:

$$\begin{aligned} \texttt{cp}_2 \stackrel{def}{=} \langle\, &\texttt{x} = \ldots, \\ &\texttt{move} = \ldots, \\ &\texttt{c} = \lambda\texttt{self}.\texttt{blue}, \\ &\texttt{makeBlue} = \lambda\texttt{self}.\langle \texttt{self} \leftarrow \texttt{c} = \texttt{blue} \rangle, \\ &\texttt{makeGreen} = \lambda\texttt{self}.\langle \texttt{self} \leftarrow \texttt{c} = \texttt{green} \rangle, \\ \rangle& \end{aligned}$$

As in the original system, we may give this expression type:

$$\texttt{cp}_2 : \mathbf{pro}\,t.\langle\!\langle\, \texttt{x} : \texttt{int}, \texttt{move} : \texttt{int} \rightarrow \texttt{t}, \texttt{c} : \texttt{colors}, \texttt{makeBlue} : \texttt{t}, \texttt{makeGreen} : \texttt{t} \rangle\!\rangle$$

With this type, the user may override $\mathtt{cp}_2$'s `c` method to give $\mathtt{cp}_2$ any color. However, we may prevent this undesired change by sealing $\mathtt{cp}_2$. We may use the rule ($<:$ *object*) and subsumption to give $\mathtt{cp}_2$ the type:

$$\mathtt{cp}_2 : \mathbf{obj}\,\mathtt{t}.\langle\!\langle\, \mathtt{x} : \mathtt{int},\, \mathtt{move} : \mathtt{int} \to \mathtt{t},\, \mathtt{c} : \mathtt{colors},\, \mathtt{makeBlue} : \mathtt{t},\, \mathtt{makeGreen} : \mathtt{t} \,\rangle\!\rangle$$

With this type, users may no longer set the color of $\mathtt{cp}_2$ by overridding its value for `c`. Now they may only change its color by sending the messages `makeBlue` and `makeGreen`, which guarantees that the object only takes on legal colors.

5 Soundness

We prove the soundness of our type system with respect to the operational semantics given in Section 2.4. We first prove that evaluation preserves type; this property is traditionally called subject reduction. We then show that no typable expression evaluates to *error* using the evaluation rules given in [FHM94]. This fact guarantees that we have no message-not-understood errors for expressions with either *pro* or *obj* types.

The proof begins with two lemmas about substitution for row and type variables. We then prove a normal form lemma that allows us to restrict our attention to derivations of a certain form, simplifying later proofs. Lemmas 4 and 5 give us subject reduction for the bookkeeping rules and β-reduction, respectively. Lemma 6 implies subject reduction for ($\Leftarrow$)-reduction on expressions with *pro* type, while Lemmas 7 and 8 give us subject reduction for ($\Leftarrow$)-reduction on expressions with *obj* type.

The following two substitution lemmas are used in conjunction with row variables to specialize *pro* types to contain additional methods.

Lemma 1. *If* $\Gamma,\, u_2 : V_2,\, \Gamma' \vdash U_1 : V_1$ *and* $\Gamma \vdash U_2 : V_2$ *are both derivable, then so is* $\Gamma, [U_2/u_2]\Gamma' \vdash [U_2/u_2]U_1 : V_1$*, where* $U_1 : V_1$ *and* $U_2 : V_2$ *are either both* $\tau : T$ *or both* $R : \kappa$.

Lemma 2. *If* $\Gamma,\, r : T^n \to [m_1, \ldots, m_k],\, \Gamma' \vdash e : \tau$ *and* $\Gamma \vdash R : T^n \to [m_1, \ldots, m_k]$ *are both derivable, then so is* $\Gamma,\, [R/r]\Gamma' \vdash e : [R/r]\tau$.

The type and row equality rules introduce many non-essential judgement derivations, which unnecessarily complicate derivation analysis. We therefore restrict our attention to $\vdash_N$-derivations, which we define as those derivations in which the only appearance of a type or row equality rule is as (*row*β) immediately following an occurrence of (*row fn app*). The τnf of a type or row expression is its normal form with respect to ordinary β-reduction. The τnf of a term expression `e` is just `e`. Since we are only interested in types and rows in τnf, the following lemma shows we can find a $\vdash_N$-derivation for any judgement of interest.

Lemma 3. *If* $\Gamma \vdash A$ *is derivable, then so is* $\tau nf(\Gamma) \vdash_N \tau nf(A)$.

The proof of this lemma is by induction on the derivation of $\Gamma \vdash A$. Occurrences of equality rules may be eliminated in the $\vdash_N$-derivation because two row or type expressions related via β-reduction must have the same τnf.

From this point on, we will only concern ourselves with contexts and type or row expressions that are in τnf. This limitation is not severe, since any term that has a type has a type in τnf. Future analyses of derivations will consider only $\vdash_N$-derivations, since its restriction on equality rules greatly simplifies the proofs.

The next three lemmas show that the various components of the $\xrightarrow{eval}$ relation preserve expression types. Lemma 4 is the first of these, showing that the $(\xrightarrow{book})$ relation has the necessary property.

Lemma 4. *If* $\Gamma \vdash \mathtt{e} : \tau$ *is derivable, and* $\mathtt{e} \xrightarrow{book} \mathtt{e}'$, *then* $\Gamma \vdash \mathtt{e}' : \tau$ *is derivable.*

The proof of Lemma 4 consists of two parts: the first shows that a derivation from $\Gamma \vdash \mathtt{e} : \tau$ can only depend on the form of τ, not on the form of $\mathtt{e}$. More formally, if $\Gamma \vdash \mathtt{C}[\mathtt{e}] : \tau$ is derived from $\Gamma' \vdash \mathtt{e} : \sigma$ and $\Gamma' \vdash \mathtt{e}' : \sigma$ is also derivable, then so is $\Gamma \vdash \mathtt{C}[\mathtt{e}'] : \tau$. This fact is easily seen by an inspection of the typing rules. The second part shows that if $\Gamma \vdash \mathtt{e} : \tau$ is derivable, and $\mathtt{e} \xrightarrow{book} \mathtt{e}'$ by $\mathtt{e}$ matching the left-hand side of one of the $(\xrightarrow{book})$ axioms, then $\Gamma \vdash \mathtt{e}' : \tau$ is also derivable. This fact follows from a case analysis of the four $(\xrightarrow{book})$ axioms. Row variables and Lemma 2 are essential for the (*switch ext ov*) case.

The fact that (β)-reduction preserves expression types is an immediate consequence of the following lemma:

Lemma 5. *If* $\Gamma, \mathtt{x} : \tau_1, \Gamma' \vdash \mathtt{e_2} : \tau_2$ *and* $\Gamma \vdash \mathtt{e_1} : \tau_1$ *are both derivable, then so is* $\Gamma, \Gamma' \vdash [\mathtt{e_1}/\mathtt{x}]\mathtt{e_2} : \tau_2$.

Lemma 5 is proved by induction on the derivation of $\Gamma, \mathtt{x} : \tau_1, \Gamma' \vdash \mathtt{e_2} : \tau_2$.

The following three lemmas are useful in showing that $(\Leftarrow)$-reduction preserves expression types.

Lemma 6. *If* $\Gamma \vdash \langle\langle \mathtt{e} \leftarrow\!\!\circ\, \mathtt{m_1}{=}\mathtt{e_1} \rangle \ldots \leftarrow\!\!\circ\, \mathtt{m_k}{=}\mathtt{e_k} \rangle : \mathbf{pro}\,t\,.\langle\!\langle R \,|\, m_1 {:} \tau_1, \ldots, m_k {:} \tau_k \rangle\!\rangle$ *is derivable, where* $m_1, \ldots, m_k$ *are distinct and are precisely the method names that occur consecutively to the right of* $\mathtt{e}$, *then*
$\Gamma \vdash \mathtt{e_i} : [\mathbf{pro}\,t\,.\langle\!\langle R \,|\, m_1 : \tau_1, \ldots, m_k : \tau_k \rangle\!\rangle / t](t \rightarrow \tau_i)$. *is also derivable.*

The proof of this lemma also has two parts. Part A gives a derivable type for each object obtained by stripping off some number of the $\leftarrow\!\!\circ$ operations. The appropriate derivation is obtained by induction on the number of $\leftarrow\!\!\circ$ operations removed. Part B gives the required type for each of the $\mathtt{e_i}$'s. Row variables and Lemma 2 are critical to its proof, which is an analysis of the derivation of the type of $\langle\langle \mathtt{e} \leftarrow\!\!\circ\, \mathtt{m_1}{=}\mathtt{e_1} \rangle \ldots \leftarrow\!\!\circ\, \mathtt{m_i}{=}\mathtt{e_i} \rangle$, given by part A.

The key insight in the soundness of the subtyping of the calculus is that if we derive that an expression $ob \stackrel{def}{=} \langle\langle e \leftarrow\!\!\circ_1\, m_1{=}e_1 \rangle \ldots \leftarrow\!\!\circ_k\, m_k{=}e_k \rangle$ has *obj* type, then a series of lemmas about typing derivations show that we must have derived that *ob* had a related *pro* type. Furthermore, the methods of *ob* must have been typed checked under the assumption that *ob* had that *pro* type. Hence any $\leftarrow$ or $\leftarrow\!\!+$ operations in the methods of *ob* were type checked with full knowledge of *ob*'s most general form and are thus type-safe. Technically, we may state this insight as:

Lemma 7. *If we may derive $\Gamma \vdash ob : \tau$, then there exists some type $\mathbf{pro}\, t . \langle\!\langle R \mid \overrightarrow{m} : \overrightarrow{\tau} \rangle\!\rangle$ such that we may derive in no more steps $\Gamma \vdash ob : \mathbf{pro}\, t . \langle\!\langle R \mid \overrightarrow{m} : \overrightarrow{\tau} \rangle\!\rangle$ and also $\Gamma \vdash \mathbf{pro}\, t . \langle\!\langle R \mid \overrightarrow{m} : \overrightarrow{\tau} \rangle\!\rangle <: \tau$, where $\overrightarrow{m} : \overrightarrow{\tau}$ is shorthand for $m_1 : \tau_1, \ldots, m_k : \tau_k$.*

To prove subject reduction for the operational semantics, we need to show that each of the basic evaluation rules preserves type. For the ($\Leftarrow$)-reduction rule, this corresponds to showing that if we may derive $\Gamma \vdash \langle \mathtt{e_1} \leftarrow\!\!\circ\, \mathtt{m} = \mathtt{e_2} \rangle \Leftarrow \mathtt{m} : \tau$ then we may also derive $\Gamma \vdash \mathtt{e_2} \langle \mathtt{e_1} \leftarrow\!\!\circ\, \mathtt{m} = \mathtt{e_2} \rangle : \tau$. This fact is proved by induction on the derivation of $\Gamma \vdash \langle \mathtt{e_1} \leftarrow\!\!\circ\, \mathtt{m} = \mathtt{e_2} \rangle \Leftarrow \mathtt{m} : \tau$. The following lemma is important for the (*object meth app*) case of this proof:

Lemma 8. *If $\Gamma \vdash \mathbf{probj}\, s_1 . \langle\!\langle R_1 \mid m : \tau_1 \rangle\!\rangle <: \mathbf{probj}\, s_2 . \langle\!\langle R_2 \mid m : \tau_2 \rangle\!\rangle$, is derivable, then $\Gamma \vdash [\mathbf{probj}\, s_1 . \langle\!\langle R_1 \mid m : \tau_1 \rangle\!\rangle / s_1]\tau_1 <: [\mathbf{probj}\, s_2 . \langle\!\langle R_2 \mid m : \tau_2 \rangle\!\rangle / s_2]\tau_2$ is also derivable.*

Theorem 9 Subject Reduction. *If $\Gamma \vdash e : \tau$ is derivable, and $e \xrightarrow{eval} e'$, then $\Gamma \vdash e' : \tau$ is also derivable.*

The proof is similar in outline to that of Lemma 4; it reduces to showing that each of the basic evaluation steps preserves the type of the expression being reduced. The ($\xrightarrow{book}$) case follows from Lemma 4, the (β) case from Lemma 5 and the ($\Leftarrow$) case from Lemmas 6, 7, and 8.

Theorem 10 Type Soundness. *If the judgement $\epsilon \vdash e : \tau$ is derivable, then $eval(e) \neq error$, where the function eval is as in [FHM94].*

6 Related work

The authors of [BL95] approach the trade-off between delegation and subtyping in a very different way. Their system, which is an extension of [FHM94], permits a limited form of width subtyping on expressions with (what we call) *pro* type. In particular, they consider $\mathbf{pro}\, t . R$ a subtype of $\mathbf{pro}\, t . R'$ so long as R contains all the methods of R' (with matching types) and none of the "forgotten" methods are referred to in the methods listed in R'. This second condition guarantees that if we forget about the presence of some method `m` via subsumption and then re-add `m` with a potentially unrelated type, we cannot violate any typing assumptions the other methods of the object may have made about `m`. This guarantee is essential to the soundness of this approach to adding subtyping; however, its inability to "forget" methods referred to by "remembered" methods is somewhat unfortunate. In particular, it cannot type examples such as that given in Section 4.2, where desired invariants about an object are maintained by restricting access to certain methods via subsumption.

7 Conclusion

In this paper, we present a type system for an object calculus that extends previous work in [Mit90, FHM94]. The main additions to our previous object calculus are the distinction between prototypes and objects and the addition of subtyping. The distinction between prototypes, which allow method addition and override, and objects, which only receive messages, is essential for subtyping, since previous studies show an incompatibility between object extension, method override, and non-trivial subtyping [FM94]. We have devised a type soundness proof for our calculus of prototypes, objects, and subtyping based on a straightforward operational semantics.

References

[AC94] M. Abadi and L. Cardelli. A theory of primitive objects: untyped and first-order systems. In *Proc. Theor. Aspects of Computer Software*, pages 296–320. Springer-Verlag LNCS 789, 1994.

[AC95] M. Abadi and L. Cardelli. An imperative object calculus: Basic typing and soundness. In *SIPL '95 - Proc. Second ACM SIGPLAN Workshop on State in Programming Languages.* Technical Report UIUCDCS-R-95-1900, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[BL95] V. Bono and L. Liquori. A subtyping for the fisher-honsell-mitchell lambda calculus of objects. Berlin, June 1995. Springer LNCS 933. To appear.

[Bru93] K. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *Proc 20th ACM Symp. Principles of Programming Languages*, pages 285–298, 1993.

[Bv93] K. Bruce and R. van Gent. TOIL: a new type-safe object-oriented imperative language. Manuscript, 1993.

[CU89] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pages 146–160, 1989.

[ES90] M. Ellis and B. Stroustrop. *The Annotated* C^{++} *Reference Manual.* Addison-Wesley, 1990.

[FHM94] K. Fisher, F. Honsell, and J.C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. Computing (*formerly *BIT)*, 1:3–37, 1994. Preliminary version appeared in *Proc. IEEE Symp. on Logic in Computer Science,* 1993, 26–38.

[FM94] K. Fisher and J.C. Mitchell. Notes on typed object-oriented programming. In *Proc. Theoretical Aspects of Computer Software*, pages 844–885. Springer LNCS 789, 1994.

[GR83] A. Goldberg and D. Robson. *Smalltalk–80: The language and its implementation.* Addison Wesley, 1983.

[Mey92] B. Meyer. *Eiffel: The Language.* Prentice-Hall, 1992.

[Mit90] J.C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 109–124, January 1990.

[Pie93] Benjamin C. Pierce. Mutable objects. Draft report; available electronically, June 1993.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, 1994.

[Str86] B. Stroustrop. *The C^{++} Programming Language.* Addison-Wesley, 1986.

[US91] D. Ungar and R.B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation*, 4(3):187–206, 1991. Preliminary version appeared in *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications,* 1987, 227-241.

A Type System

Types

$$\tau \ ::= \ t \,|\, s \,|\, \tau_1 \rightarrow \tau_2 \,|\, \mathbf{pro}\, t \,.\, R \,|\, \mathbf{obj}\, t \,.\, R$$

Rows

$$R \ ::= \ r \,|\, \langle\!\langle\rangle\!\rangle \,|\, \langle\!\langle R \,|\, m : \tau \rangle\!\rangle \,|\, \lambda t.\, R \,|\, R\tau$$

Kinds

$$kind \ ::= \ T \,|\, \kappa$$

$$\kappa \ ::= \ T^n \rightarrow [m_1, \ldots, m_k]$$

Contexts:

$$\Gamma \ ::= \ \epsilon \,|\, \Gamma, x : \tau \,|\, \Gamma, t : T \,|\, \Gamma, r : \kappa \,|\, \Gamma, s_1 <: s_2$$

The judgement forms are standard:

$\Gamma \vdash *$	well-formed context
$\Gamma \vdash e : \tau$	term has type
$\Gamma \vdash \tau : T$	well-formed type
$\Gamma \vdash \tau_1 <: \tau_2$	τ_1 subtype of τ_2
$\Gamma \vdash R : \kappa$	row has kind

A.1 Typing rules

General Rules

(*start*)
$$\frac{}{\epsilon \vdash *}$$

(*projection*)
$$\frac{\begin{array}{c}\Gamma \vdash * \\ u : v \in \Gamma\end{array}}{\Gamma \vdash u : v}$$

(*weakening*)
$$\frac{\begin{array}{c}\Gamma \vdash A \\ \Gamma, \Gamma' \vdash *\end{array}}{\Gamma, \Gamma' \vdash A}$$

Rules for type expressions

(*type var*)
$$\frac{\begin{array}{c}\Gamma \vdash * \\ t \notin dom(\Gamma)\end{array}}{\Gamma, t : T \vdash *}$$

(*type arrow*)
$$\frac{\begin{array}{c}\Gamma \vdash \tau_1 : T \\ \Gamma \vdash \tau_2 : T\end{array}}{\Gamma \vdash \tau_1 \rightarrow \tau_2 : T}$$

(*type pro*)
$$\frac{\Gamma, t : T \vdash R : [m_1, \ldots, m_k]}{\Gamma \vdash \mathbf{pro}\, t \mathbf{.} R : T}$$

Type and Row Equality

Type or row expressions that differ only in names of bound variables or order of *label* : *type* pairs are considered identical. In other words, we consider α-conversion of type variables bound by λ, *pro*, or *obj* and applications of the principle

$$\langle\!\langle \langle\!\langle R \mid n : \tau_1 \rangle\!\rangle \mid m : \tau_2 \rangle\!\rangle = \langle\!\langle \langle\!\langle R \mid m : \tau_2 \rangle\!\rangle \mid n : \tau_1 \rangle\!\rangle$$

within type or row expressions to be conventions of syntax, rather than explicit rules of the system. Additional equations between types and rows arise as a result of β-reduction, written $\rightarrow_\beta$, or β-conversion, written $\leftrightarrow_\beta$.

(*row* β)
$$\frac{\Gamma \vdash R : \kappa, \quad R \rightarrow_\beta R'}{\Gamma \vdash R' : \kappa}$$

(*type* β)
$$\frac{\Gamma \vdash \tau : T, \quad \tau \rightarrow_\beta \tau'}{\Gamma \vdash \tau' : T}$$

$(type\ eq)$
$$\frac{\Gamma \vdash e : \tau, \quad \tau \leftrightarrow_\beta \tau', \quad \Gamma \vdash \tau' : T}{\Gamma \vdash e : \tau'}$$

Rules for rows

$(empty\ row)$
$$\frac{\Gamma \vdash *}{\Gamma \vdash \langle\!\langle\rangle\!\rangle : [m_1, \ldots, m_k]}$$

$(row\ var)$
$$\frac{\begin{array}{c}\Gamma \vdash * \\ r \notin dom(\Gamma)\end{array}}{\Gamma,\, r : T^n \rightarrow [m_1, \ldots, m_k] \vdash *}$$

$(row\ label)$
$$\frac{\begin{array}{c}\Gamma \vdash R : T^n \rightarrow [m_1, \ldots, m_k] \\ \{n_1, \ldots, n_\ell\} \subseteq \{m_1, \ldots, m_k\}\end{array}}{\Gamma \vdash R : T^n \rightarrow [n_1, \ldots, n_\ell]}$$

$(row\ ext)$
$$\frac{\begin{array}{c}\Gamma \vdash R : [m, m_1, \ldots, m_k] \\ \Gamma \vdash \tau : T\end{array}}{\Gamma \vdash \langle\!\langle R \,|\, m : \tau \rangle\!\rangle : [m_1, \ldots, m_k]}$$

$(row\ fn\ abs)$
$$\frac{\Gamma, t : T \vdash R : T^n \rightarrow [m_1, \ldots, m_k]}{\Gamma \vdash \lambda t.\, R : T^{n+1} \rightarrow [m_1, \ldots, m_k]}$$

$(row\ fn\ app)$
$$\frac{\begin{array}{c}\Gamma \vdash R : T^{n+1} \rightarrow [m_1, \ldots, m_k] \\ \Gamma \vdash \tau : T\end{array}}{\Gamma \vdash R\tau : T^n \rightarrow [m_1, \ldots, m_k]}$$

Rules for assigning types to terms

$(exp\ var)$
$$\frac{\begin{array}{c}\Gamma \vdash \tau : T \\ x \notin dom(\Gamma)\end{array}}{\Gamma,\, x : \tau \vdash *}$$

$(exp\ abs)$
$$\frac{\Gamma\,,\, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \rightarrow \tau_2}$$

$(exp\ app)$
$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

$(empty\ object)$
$$\frac{\Gamma \vdash *}{\Gamma \vdash \langle\rangle : \mathbf{pro}\, t.\langle\!\langle\rangle\!\rangle}$$

$$(pro\ meth\ app) \qquad \frac{\Gamma \vdash e : \mathbf{pro}\, t.\langle\!\langle R \mid m : \tau \rangle\!\rangle}{\Gamma \vdash e \Leftarrow m : [\mathbf{pro}\, t.\langle\!\langle R \mid m : \tau \rangle\!\rangle / t]\tau}$$

$$(obj\ ext) \quad \frac{\begin{array}{l} \Gamma \vdash e_1 : \mathbf{pro}\, t.\langle\!\langle R \mid m_1 : \tau_1, \ldots, m_k : \tau_k \rangle\!\rangle \\ \Gamma, t : T \vdash R : [m_1, \ldots, m_k, n] \\ \Gamma, r : T \rightarrow [m_1, \ldots, m_k, n] \vdash \\ \quad e_2 : [\mathbf{pro}\, t.\langle\!\langle rt \mid \overrightarrow{m} : \overrightarrow{\tau}, n : \tau \rangle\!\rangle / t](t \rightarrow \tau) \qquad r \text{ not in } \tau \end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!+ n = e_2 \rangle : \mathbf{pro}\, t.\langle\!\langle R \mid \overrightarrow{m} : \overrightarrow{\tau}, n : \tau \rangle\!\rangle}$$

$$(obj\ over) \qquad \frac{\begin{array}{l} \Gamma \vdash e_1 : \mathbf{pro}\, t.\langle\!\langle R \mid m_1 : \tau_1, \ldots, m_k : \tau_k \rangle\!\rangle \\ \Gamma, r : T \rightarrow [m_1, \ldots, m_k] \vdash \\ \quad e_2 : [\mathbf{pro}\, t.\langle\!\langle rt \mid \overrightarrow{m} : \overrightarrow{\tau} \rangle\!\rangle / t](t \rightarrow \tau_i) \end{array}}{\Gamma \vdash \langle e_1 \leftarrow m_i = e_2 \rangle : \mathbf{pro}\, t.\langle\!\langle R \mid \overrightarrow{m} : \overrightarrow{\tau} \rangle\!\rangle}$$

Here $\overrightarrow{m} : \overrightarrow{\tau}$ is used as an abbreviation for $m_1 : \tau_1, \ldots, m_k : \tau_k$.

A.2 New Rules

$$(type\ object) \qquad \frac{\Gamma, t : T \vdash R : [m_1, \ldots, m_k]}{\Gamma \vdash \mathbf{obj}\, t.R : T}$$

$$(<:\ var) \qquad \frac{\begin{array}{c} \Gamma \vdash * \\ s_1, s_2 \notin dom(\Gamma) \\ s_1 \neq s_2 \end{array}}{\Gamma, s_1 <: s_2 \vdash *}$$

$$(<:\ proj) \qquad \frac{\begin{array}{c} \Gamma \vdash * \\ s_1 <: s_2 \in \Gamma \end{array}}{\Gamma \vdash s_1 <: s_2}$$

$$(<:\ var\ proj) \qquad \frac{\begin{array}{c} \Gamma \vdash * \\ s_1 <: s_2 \in \Gamma \end{array}}{\Gamma \vdash s_i : T}$$

$$(<:\ refl) \qquad \frac{\Gamma \vdash \tau : T}{\Gamma \vdash \tau <: \tau}$$

$$(<:\ trans) \qquad \frac{\Gamma \vdash \sigma <: \tau \qquad \Gamma \vdash \tau <: \rho}{\Gamma \vdash \sigma <: \rho}$$

$$(<: arrow) \qquad \frac{\Gamma \vdash \tau_1\ <:\ \sigma_1\ \Gamma \vdash \sigma_2 <: \tau_1}{\Gamma \vdash \sigma_1 \rightarrow \sigma_2 <: \tau_1 \rightarrow \tau_2}$$

$$(<:\ object) \qquad \frac{\begin{array}{c}\Gamma \vdash \mathbf{probj}\, s_1 \mathbf{.} \langle\!\langle \overrightarrow{m} : \overrightarrow{\tau},\ \overrightarrow{n} : \overrightarrow{\sigma} \rangle\!\rangle : T \\ \Gamma \vdash \mathbf{obj}\, s_2 \mathbf{.} \langle\!\langle \overrightarrow{m} : \overrightarrow{\tau}' \rangle\!\rangle : T \\ \Gamma, s_1 <: s_2 \vdash \tau_i <: \tau_i' \\ s_1 \notin FV(\tau_i'), \quad s_2 \notin FV(\tau_i)\end{array}}{\Gamma \vdash \mathbf{probj}\, s_1 \mathbf{.} \langle\!\langle \overrightarrow{m} : \overrightarrow{\tau},\ \overrightarrow{n} : \overrightarrow{\sigma} \rangle\!\rangle <: \mathbf{obj}\, s_2 \mathbf{.} \langle\!\langle \overrightarrow{m} : \overrightarrow{\tau}' \rangle\!\rangle}$$

$$(subsumption) \qquad \frac{\Gamma \vdash \sigma\ <:\ \tau \qquad \Gamma \vdash e\ :\ \sigma}{\Gamma \vdash e\ :\ \tau}$$

$$(object\ meth\ app) \qquad \frac{\Gamma \vdash e\ :\ \mathbf{obj}\, t \mathbf{.} \langle\!\langle R \mid m : \tau \rangle\!\rangle}{\Gamma \vdash e \Leftarrow m\ :\ [\mathbf{obj}\, t \mathbf{.} \langle\!\langle R \mid m : \tau \rangle\!\rangle / t]\tau}$$

This article was processed using the LaTeX macro package with LLNCS style