Nordic Journal of Computing 1(1994), 3–37.

A LAMBDA CALCULUS OF OBJECTS AND METHOD SPECIALIZATION

KATHLEEN FISHER [*]
Computer Science Department
Stanford University
Stanford, California 94305-4055, U.S.A.
kfisher@cs.stanford.edu

FURIO HONSELL [†]
Dipartimento di Informatica
Universitá di Udine
via Zanon, 6
33100 Udine, ITALY
honsell@dimi.uniud.it

JOHN C. MITCHELL [‡]
Computer Science Department
Stanford University
Stanford, California 94305-4055, U.S.A.
jcm@cs.stanford.edu

Abstract. This paper presents an untyped lambda calculus, extended with object primitives that reflect 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. Type soundness is proved using operational semantics and examples illustrating the expressiveness of the pure calculus are presented.

CR Classification: F.3.1, D.3.3, F.4.1

1. Introduction

There are several forms of object-oriented languages. One of the major lines of difference is between class-based and delegation-based languages. In class-based languages such as Smalltalk [9] and C^{++} [8], each object is created by a class and inheritance is determined by the class. In delegation-based languages such as Self [14, 7], an object may be created from another object, inheriting properties from the original. In this paper, we use an untyped lambda calculus of objects with a functional form of delegation as a tool for studying typing issues in object-oriented programming languages. Our

[*] Supported in part by an NSF Graduate Fellowship, an NSF PYI Award and NSF CCR-9303099.
[†] Supported in part by a CNR grant, and Mitchell's funding at Stanford spring 1992.
[‡] Supported in part by an NSF PYI Award, with matching funds from AT&T, Digital Equipment Corporation, the Powell Foundation and Xerox Corporation; and by NSF CCR-9303099 and the Wallace F. and Lucille M. Davis Faculty Scholarship.

Received May 1994. Accepted May 1994.

main interests lie in (i) understanding how the functionality of a method may change as it is inherited, intuitively due to reinterpretation of the special symbol *self* (or *this* in C^{++}), and (ii) setting the stage for equational reasoning about method bodies, apart from the particular context in which they first occur. The second goal seems a necessary precurser to understanding the effects of optimization or program transformations on method bodies or inheritance hierarchies. The main reason for using a delegation-based language in this study is its relative simplicity, when compared with class-based languages.

In our calculus, the main operations on objects are to send a message `m` to an object `e`, written $\mathtt{e} \Leftarrow \mathtt{m}$, and two forms of method definition. If expression `e` denotes an object without method `m`, then $\langle \mathtt{e} \longleftrightarrow\!\!\!\!+ \ \mathtt{m} = \mathtt{e}' \rangle$ denotes an object obtained from `e` by adding the method body $\mathtt{e}'$ for `m`. When $\langle \mathtt{e} \longleftrightarrow\!\!\!\!+ \ \mathtt{m} = \mathtt{e}' \rangle$ is sent the message `m`, the result is obtained by applying $\mathtt{e}'$ to $\langle \mathtt{e} \longleftrightarrow\!\!\!\!+ \ \mathtt{m} = \mathtt{e}' \rangle$. This form of "self-application" allows us to model the special symbol *self* of object-oriented languages directly by lambda abstraction. Intuitively, the method body $\mathtt{e}'$ must be a function, and the first actual parameter of $\mathtt{e}'$ will always be the "object itself." To reinforce this intuition, we often write method bodies in the form $\lambda \mathtt{self}.(\ldots)$. The final method operation on objects is to replace one method body by another. This provides a functional form of update. As in the language Self, we do not distinguish instance variables from methods, since this does not seem essential. The untyped lambda calculus we use bears a strong resemblance to the T object system [13, 2] (although it was originally developed without prior knowledge of T) and the untyped part of the calculus used in [1] to model a fragment of Modula 3 [3, 4].

The main goal of this paper is to develop a type system that allows methods to be specialized appropriately as they are inherited. Briefly, suppose `p` is a point object with `x` and `y` methods returning the integer x- and y-coordinates of `p`, and a `move` method with type $\mathtt{int} \times \mathtt{int} \rightarrow \mathtt{point}$. Method `move` has this type because if we send the message `move` to `p`, we obtain a function which given distances to move in the x and y directions, returns a point identical to `p`, but with updated x- and y-coordinates. If we create a colored point `cp` from `p` by the object-extension operation, then `cp` inherits the `x`, `y` and `move` methods from `p`. In an untyped object-oriented language such as Smalltalk, the inherited `move` method will change the position of a colored point, leaving the color unchanged. Therefore, in a typed language, we want the `move` method of `cp` to have the "specialized" type $\mathtt{int} \times \mathtt{int} \rightarrow \mathtt{colored\_point}$. If the inherited method had its original type $\mathtt{int} \times \mathtt{int} \rightarrow \mathtt{point}$, then whenever we moved a colored point, we would obtain an ordinary point without color, making the inherited `move` function largely useless. While an imperative version of `move` would bypass this difficulty by returning type `unit` (as it is called in ML, or `void` in C^{++}), the same issue arises when we attempt to define a function that must return a value. The construct `like Current` in Eiffel [10], analyzed in [6], illustrates the value of specializing the type of a method in an imperative language.

While C^{++} does not include such a construct, the widespread use of C^{++} is not counter-evidence to the usefulness of method specialization. In fact, it appears to be common for novice C^{++} programmers to attempt to specialize the types of methods in derived classes. More experienced C^{++} programmers appear to use "down casts" to approximate the effects described in this paper.

The phenomenon we are concerned with is called "method specialization" in [12], which describes a precursor to the calculus used here. The earlier paper describes method specialization and explains its usefulness, but only presents a tentative type system by extending the already complicated record calculus of [5]. In addition, no analysis of the type system is given. The current paper presents a calculus of objects alone, without recourse to record calculi (although we owe a substantial debt to previous studies of record calculi), simplifies the typing rules substantially, and proves type soundness. In addition, we show that equality of typable terms is undecidable by showing how to encode numerals as objects and how to define all partial recursive function. Objects play an essential role here, since the function part of our calculus is simply typed, and therefore only sufficient to express total recursive functions. A preliminary version of this work appeared in [11].

2. Untyped objects and inheritance by delegation

2.1 Untyped calculus of objects

We extend the untyped lambda calculus with four object-related syntactic forms,

$$\mathtt{e} \;::=\; \mathtt{x} \mid \mathtt{c} \mid \lambda\mathtt{x}.\,\mathtt{e} \mid \mathtt{e}_1\mathtt{e}_2 \mid \langle\rangle \mid \mathtt{e} \Leftarrow \mathtt{m} \mid \langle \mathtt{e}_1 \leftarrow\!\!+\, \mathtt{m}{=}\mathtt{e}_2 \rangle \mid \langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_2 \rangle$$

In this grammar, x may be any variable, c is a constant symbol (such as a "built-in" function), λx. e is a lambda abstraction (function expression) and $\mathtt{e}_1\,\mathtt{e}_2$ is function application. The object forms are described in tabular form for easy reference:

$\langle\rangle$	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. The reason for distinguishing extension from method replacement is that these two operations will have different typing rules. 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. If we were not concerned with static typing, then we could use a single operation that adds a method to an object, replacing any existing method with the same name.

2.2 Examples of objects, inheritance, and method specialization

To provide some intuition for this calculus, we give a few short examples. The first shows how records may be encoded as objects, while the second and third illustrate method specialization. The latter examples may be regarded as the motivating examples for this paper; rather than try to define method specialization in general, we attempt to convey the essential properties by the examples of points and colored points given below.

To simplify notation, we write $\langle \mathtt{m}_1 = \mathtt{e}_1, \ldots, \mathtt{m}_k = \mathtt{e}_k \rangle$ for $\langle \ldots \langle\langle\rangle \longleftarrow\!\!\!+ \mathtt{m}_1 = \mathtt{e}_1\rangle \ldots \longleftarrow\!\!\!+ \mathtt{m}_k = \mathtt{e}_k\rangle$, where $\mathtt{m}_1, \ldots, \mathtt{m}_k$ are distinct method names. We illustrate the computational behavior of objects in this section using a simplified evaluation rule that reflects the operational semantics defined precisely below,

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

This allows us to evaluate a message send by retrieving the appropriate method body from the object and applying it to the entire object itself. Note that the relation $\xrightarrow{eval}$ represents one evaluation step, not full evaluation of an expression.

Record with two components. The first example is a form of "point" object that has constant x, y-coordinates:

$$\mathtt{r} \overset{def}{=} \langle \mathtt{x} = \lambda \mathtt{self}.\,3,\ \mathtt{y} = \lambda \mathtt{self}.\,2 \rangle$$

If we send the message `x` to `r`, we may calculate the result by

$$\mathtt{r} \Leftarrow \mathtt{x} \quad \xrightarrow{eval} \quad (\lambda \mathtt{self}.\,3)\,\mathtt{r} \quad \xrightarrow{eval} \quad 3$$

where the second evaluation step is ordinary β-reduction from lambda calculus. This example may be generalized to show how any record may be represented as an object whose methods are constant functions. In particular, we may represent the record $\langle \mathtt{l}_1 = \mathtt{e}_1, \ldots, \mathtt{l}_k = \mathtt{e}_k \rangle$ by the object $\langle \mathtt{m}_1 = \mathtt{K}\mathtt{e}_1, \ldots, \mathtt{m}_k = \mathtt{K}\mathtt{e}_k \rangle$, where $\mathtt{K} = \lambda \mathtt{x}.\, \lambda \mathtt{self}.\, \mathtt{x}$.

One-dimensional point with `move` *function.* A more interesting object, which we will refer to again, is the following "point" object with an x-coordinate and `move` method. We could easily give a similar two-dimensional point with x- and y-coordinates, but the one-dimensional case illustrates the same ideas more simply.

$$\begin{array}{lll} \mathtt{p} & \overset{def}{=} & \langle \quad \mathtt{x} = \lambda \mathtt{self}.\,3, \\ & & \qquad \mathtt{move} = \lambda \mathtt{self}.\,\lambda \mathtt{dx}.\ \langle \mathtt{self} \longleftarrow \mathtt{x} = \lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} \rangle \\ & & \rangle \end{array}$$

The `move` method, when applied to the object itself and a displacement `dx`, replaces the `x` method by one returning a coordinate incremented by `dx`. This is illustrated in the following example calculation, where we send the message `move` with parameter 2 to the object `p`:

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

Using a sound rule for object equality,

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

we may reach the conclusion

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

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

Inherit `move` *from point to colored point.* Our third introductory example shows how `x` and `move` are inherited when a colored point is defined from `p` by adding a `color` method.

$$\mathtt{cp} \overset{def}{=} \langle \mathtt{p} \leftarrow\!\!+\ \mathtt{color} = \lambda\mathtt{self}.\,\mathtt{red}\rangle$$

If we send the `move` message to `cp` with the same parameter as above, we may calculate the resulting object in exactly the same way as before:

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

with the final conclusion that

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

The important feature of this computation is that the `color` of the resulting colored point is the same as the original one. While `move` was defined originally for points, which only have an x-coordinate, the method body performs the correct computation when the method is inherited by a more complicated object with additional methods.

In many cases, it is also useful to redefine an inherited method to exhibit more specialized behavior. This may be accomplished in our calculus by a combination of inheritance and method redefinition. For example, if we want an object to change to a darker color when moved, we could first define colored points from points as above, obtaining a colored point with the right type of `move` method. Then, `move` could be redefined (without changing its type) to have the right behavior.

Mutually recursive methods. As a technical simplification, our system is formulated so that methods are added to an object one at a time. This leads us to formulate our typing rules in a manner that makes it difficult to write object expressions with mutually recursive functions. More specifically, the static type system will only allow a method body to be added if all the other methods it refers to are already available from the object. For example, we cannot type the object expression

$$\langle\langle\rangle \longleftarrow\!\!+\ \texttt{x\_plus1} = \lambda\texttt{self}.(\texttt{self} \Leftarrow \texttt{x}) + \texttt{1}\rangle$$

which has a method referring to `x` but does not have an `x` method. The reason this object expression is not typable is that if we send it the message `x_plus1`, the object will then send the message `x` to itself. But since the object does not have an `x` method, this is an error; it is precisely the error we aim to prevent with our type system. On the other hand, we may type the object expression

$$\langle\langle\langle\rangle \longleftarrow\!\!+\ x = \lambda\texttt{self}.\,3\rangle \longleftarrow\!\!+\ \texttt{x\_plus1} = \lambda\texttt{self}.(\texttt{self} \Leftarrow \texttt{x}) + \texttt{1}\rangle,$$

which is formed by first extending the empty object with an `x` method, then the `x_plus1` method that refers to `x`.

The typing restriction that no method may refer to a method that the object does not have is inconvenient if we wish to add mutually-recursive methods `m` and `n` to some object. However, there is a standard idiom for adding mutually recursive methods. Specifically, we first extend the object by giving some method body for `m` that has the correct type but does not depend on `n`. Then, the object may be extended with the desired method body for `n`, referring to `m`. Finally, we replace the "dummy" method body for `m` with the desired method body referring to `n`. While this is a programming inconvenience, it is not a limitation in expressiveness. It therefore does not seem serious enough to merit complicating the typing rules in a way that alleviates the difficulty. In any "real" programming language based on our illustrative object calculus, we would expect there to be convenient syntactic sugar for simultaneously adding several, possibly mutually recursive, methods to an object.

2.3 Operational semantics

In defining the operational semantics of our calculus, we must give rules for extracting and applying the appropriate method of an object. A natural way to approach this is 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 would let us treat objects as sets of methods, rather than ordered sequences. However, this equational 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

$$\langle\langle\langle\rangle \longleftarrow\!\!+ \mathtt{x} = \lambda\mathtt{self}.\,3\rangle \longleftarrow\!\!+ \mathtt{x\_plus1} = \lambda\mathtt{self}.(\mathtt{self} \Leftarrow \mathtt{x}) + 1\rangle$$

then the subexpression

$$\langle\langle\rangle \longleftarrow\!\!+ \mathtt{x\_plus1} = \lambda\mathtt{self}.(\mathtt{self} \Leftarrow \mathtt{x}) + 1\rangle$$

is not well-typed, as described in the previous section. Therefore, the entire expression cannot be typed.

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

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

where each method is defined exactly once, using some arbitrary method body that does not contribute to the observable behavior of the object, and 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 object and function expressions, 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 I. These rules also allow the method redefinitions to be permuted arbitrarily. An important property of $\xrightarrow{book}$, proved in Section 4, is that if $\mathtt{e} \xrightarrow{book} \mathtt{e}'$, then any type for $\mathtt{e}$ is also derivable for $\mathtt{e}'$. This would fail if we had the more general permutation rule discussed above.

The evaluation relation is the congruence closure of the union of $\xrightarrow{book}$ and the two evaluation clauses, (β) and $(\Leftarrow)$, at the bottom of Table I. In other words, $\mathtt{e} \xrightarrow{eval} \mathtt{e}'$ if we may obtain $\mathtt{e}'$ from $\mathtt{e}$ by applying a bookkeeping or basic evaluation step to one subterm.

3. Static type system

3.1 Class types and message send

The type of an object will be called a *class type.* This is in part a misuse of the word "class," since classes in object-oriented languages generally determine the representations as well as the interfaces, or types, of objects.

(*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)	$(\lambda \mathtt{x}.\, \mathtt{e}_1)\mathtt{e}_2$	$\xrightarrow{eval}$	$[\mathtt{e}_2/\mathtt{x}]\mathtt{e}_1$
$(\Leftarrow)$	$\langle \mathtt{e}_1 \leftarrow\!\ominus\, \mathtt{m}{=}\mathtt{e}_2\rangle \Leftarrow \mathtt{m}$	$\xrightarrow{eval}$	$\mathtt{e}_2\langle \mathtt{e}_1 \leftarrow\!\ominus\, \mathtt{m}{=}\mathtt{e}_2\rangle$ where $\leftarrow\!\ominus$ may either $\leftarrow\!\!+$ or $\leftarrow$.

TABLE I: Bookkeeping and evaluation rules.

However, "class" seems the best standard term from object-oriented programming that suggests the type of an object.

The type defined by the type expression

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

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

The typing rule for message send has the form

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

where the substitution for t in τ reflects the recursive nature of class types. This rule may be used to give the point $\mathtt{p}$ with $\mathtt{x}$ and $\mathtt{move}$ methods considered in Section 2.2 type

$$\mathbf{class}\, t.\langle\!\langle\, \mathtt{x} : \mathit{int},\ \mathtt{move} : \mathit{int} \to t \rangle\!\rangle,$$

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

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 extend $\mathtt{p}$ with a color to obtain $\mathtt{cp}$, also given in Section 2.2, we obtain an object with type

$$\mathbf{class}\, t.\langle\!\langle\, \mathtt{x} : \mathit{int},\ \mathtt{move} : \mathit{int} \to t,\ \mathtt{color} : \mathit{colors} \rangle\!\rangle$$

The important change to notice here is that although the syntactic type of $\mathtt{move}$ is still $\mathit{int} \to t$, 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 is the effect that we have called method specialization – the type of a method may change when the method is inherited. For this kind of reinterpretation of type variables to be sound, the typing rule for object extension must insure that every possible type for a new method will be correct. This is done through a form of implicit higher-order polymorphism.

Another subtle aspect of the type system is that objects which behave identically when we send either any sequence of messages may have different types. This is because adding and redefining methods are also considered operations on objects. A simple example is given by the following two objects.

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

$$\begin{array}{lcll} \mathtt{q} & \overset{def}{=} & \langle & \mathtt{x} = \lambda\mathtt{self.}\,3, \\ & & & \mathtt{move} = \lambda\mathtt{self.}\,\lambda\mathtt{dx.}\ \ (\mathtt{p} \Leftarrow \mathtt{move}\ \mathtt{dx}) \\ & & \rangle & \end{array}$$

It is not hard to see that p and q return the same results for any sequence of message sends. (Either we send x, which clearly gives the same result for p or q, or we send the message move. But since $\mathtt{q} \Leftarrow \mathtt{move}$ uses $\mathtt{p} \Leftarrow \mathtt{move}$, any sequence of subsequent messages will produce identical results.) However, p and q are *not* equivalent if we extend them with additional methods. The reason is that the first move method will preserve any additional methods added by object extension, but the second will not. This distinction shows up in the type system, where we can give p the first type below and q the second, but it is unsound to give q the first type.

$$\begin{array}{lcl} \mathit{point} & = & \mathbf{class}\, t\,\mathbf{.}\langle\!\langle\, \mathtt{x} : \mathit{int},\, \mathtt{move} : \mathit{int} \rightarrow t \rangle\!\rangle \\[1ex] \mathit{q\_point} & = & \mathbf{class}\, t\,\mathbf{.}\langle\!\langle\, \mathtt{x} : \mathit{int},\, \mathtt{move} : \mathit{int} \rightarrow \mathit{point} \rangle\!\rangle \end{array}$$

A similar situation arises in Smalltalk, for example, where it is possible to have two classes that generate equivalent objects, but behave differently when we inherit from them. In adapting our type system to Smalltalk, we might expect to distinguish two such classes by type. The reason is that we wish the type of a Smalltalk class to not only give information about the behavior of objects, but also about the types of methods when they are inherited by other classes.

3.2 Types, rows, and kinds

The type expressions include type variables, function types, and class types. It would not change the system in any substantial way to add type constants, but we will not need them in this paper.

Types

$$\tau \ ::= \ t \,|\, \tau_1 \to \tau_2 \,|\, \mathbf{class}\, t.R$$

Rows

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

Kinds

$$\begin{aligned} kind \ &::= \ T \,|\, \kappa \\ \kappa \ &::= \ T^n \to [m_1, \ldots, m_k] \end{aligned}$$

The row expressions appear as subexpressions of type expressions, with rows and types distinguished by kinds. As a notational simplification, we write $[m_1, \ldots, m_k]$ for $T^0 \to [m_1, \ldots, m_k]$. 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 $T^n \to [m_1, \ldots, m_k]$, 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.

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

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

3.3 Typing rules

For the most part, the formulation of the type system is routine. The most novel rules, which are discussed below, are those associated with objects. The complete type system appears in Appendix A.

The empty object $\langle\rangle$ has the object type $\mathbf{class}\, t.\langle\!\langle\rangle\!\rangle$, as specified in the rule:

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

The empty object has no methods and therefore cannot respond to any messages. However, this object can be extended with methods to obtain objects of other types.

The typing rule (*meth app*) for sending a message to an object has the form described in the last section. However, in the more precise rule below, the row occurring in the class type of the object may have any form as long as the method name being used occurs explicitly. Type equality allows the order of method names to be permuted, as described in Appendix A. Consequently, there is no loss of generality in assuming `m` is the last method listed in the type.

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

The most subtle and complicated rule of the system is the (*obj ext*) typing rule for adding methods to objects. In this rule we assume $\mathtt{e}_1$ is an object of some class type and that $\mathtt{e}_1$ does not include a method `n`, to be added. The final assumption is a typing for $\mathtt{e}_2$, the expression to be used as the method body for `n`. The first thing to notice about the typing for $\mathtt{e}_2$ is that it contains a row variable r, which is implicitly universally quantified. Because of this quantification, $\mathtt{e}_2$ will have the indicated type for any substitution of row expression R for r, provided R has the correct kind. (See Lemma 4.4 below.) This is essential, since it implies that $\mathtt{e}_2$ will have the required functionality for any possible future extension of $\langle \mathtt{e}_1 \longleftarrow\!\!\!+ \; \mathtt{n} = \mathtt{e}_2 \rangle$. The second important property of the typing for $\mathtt{e}_2$ is that the type has the form $t \to \tau$, with a class type substituted for t. While t is hidden in the class type of $\langle \mathtt{e}_1 \longleftarrow\!\!\!+ \; \mathtt{n} = \mathtt{e}_2 \rangle$, it is necessary in the hypothesis since sending the message `n` to $\langle \mathtt{e}_1 \longleftarrow\!\!\!+ \; \mathtt{n} = \mathtt{e}_2 \rangle$ will result in the application of $\mathtt{e}_2$ to this object. To simplify notation, we will use $\vec{m} : \vec{\tau}$ as an abbreviation for $m_1 : \tau_1, \ldots, m_k : \tau_k$.

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

The rule for redefining, or overriding, a method body has the same form, but is slightly simpler.

3.4 Example typing derivations

We claimed earlier that we should be able to prove the following two typings:

$$\begin{array}{lcl} \mathtt{p} & : & \mathbf{class}\, t.\langle\!\langle\, \mathtt{x} : \mathit{int},\, \mathtt{move} : \mathit{int} \to t \rangle\!\rangle \\ \mathtt{q} & : & \mathbf{class}\, t.\langle\!\langle\, \mathtt{x} : \mathit{int},\, \mathtt{move} : \mathit{int} \to \mathit{point} \rangle\!\rangle, \end{array}$$

and argued that it is unsound to give `q` the first type. To illustrate the use of the typing rules, we prove the first of these in Table II. The second is proved similarly. In addition, we can prove that the object expression

$$\mathtt{cp} \;=\; \langle \mathtt{p} \longleftarrow\!\!\!+ \; \mathtt{color} = \lambda \mathtt{self.red} \rangle$$

Contexts

$$\begin{array}{lcl} \Gamma_1 & = & r : T \rightarrow [\mathtt{x}, \mathtt{move}], \\ & & \mathtt{self} : \mathbf{class}\, t.\langle\!\langle rt \mid \mathtt{x} : \mathit{int}, \mathtt{move} : \mathit{int} \rightarrow t \rangle\!\rangle, \\ & & \mathtt{dx} : \mathit{int} \\ \Gamma_2 & = & \Gamma_1,\ r' : T \rightarrow [\mathtt{x}, \mathtt{move}], \\ & & \mathtt{p} : \mathbf{class}\, t.\langle\!\langle r't \mid \mathtt{x} : \mathit{int}, \mathtt{move} : \mathit{int} \rightarrow t \rangle\!\rangle \end{array}$$

Derivation (assuming $\epsilon \vdash \langle \mathtt{x} {=} \lambda \mathtt{self}.\, 3 \rangle : \mathbf{class}\, t.\langle\!\langle \mathtt{x} : \mathit{int} \rangle\!\rangle$)

$$\begin{array}{rcl} \Gamma_2 & \vdash & (\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} : \mathit{int} \\ \Gamma_2 - \mathtt{p} & \vdash & \lambda \mathtt{p}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} : \mathbf{class}\, t.\langle\!\langle r't \mid \mathtt{x} : \mathit{int}, \mathtt{move} : \mathit{int} \rightarrow t \rangle\!\rangle \rightarrow \mathit{int} \\ \Gamma_1 & \vdash & \langle \mathtt{self} \leftarrow \mathtt{x} {=} \lambda \mathtt{p}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} \rangle \\ & & : \mathbf{class}\, t.\langle\!\langle rt \mid \mathtt{x} : \mathit{int}, \mathtt{move} : \mathit{int} \rightarrow t \rangle\!\rangle \\ \Gamma_1 - \mathtt{dx} - \mathtt{self} & \vdash & \lambda \mathtt{self}.\, \lambda \mathtt{dx}.\langle \mathtt{self} \leftarrow \mathtt{x} {=} \lambda \mathtt{p}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} \rangle \\ & & : [\mathbf{class}\, t.\langle\!\langle rt \mid \mathtt{x} : \mathit{int}, \mathtt{move} : \mathit{int} \rightarrow t \rangle\!\rangle / t](t \rightarrow \mathit{int} \rightarrow t) \\ \epsilon & \vdash & \langle \langle \mathtt{x} {=} \lambda \mathtt{self}.\, 3 \rangle \\ & & \leftarrow\!\!+\ \mathtt{move} {=} \lambda \mathtt{self}.\, \lambda \mathtt{dx}.\langle \mathtt{self} \leftarrow \mathtt{x} {=} \lambda \mathtt{p}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} \rangle \rangle \\ & & : \mathbf{class}\, t.\langle\!\langle \mathtt{x} : \mathit{int}, \mathtt{move} : \mathit{int} \rightarrow t \rangle\!\rangle \end{array}$$

Table II: Example typing derivation.

has type

$$\mathbf{class}\, t.\langle\!\langle\, \mathtt{x} : \mathit{int}, \mathtt{move} : \mathit{int} \rightarrow t, \mathtt{color} : \mathit{colors} \rangle\!\rangle$$

by similar means. These examples are intended to demonstrate that the type system captures the desired form of method specialization.

4. Soundness of the Type System

We prove the soundness of the type system in several steps. The first is subject reduction, i.e., if e has type τ and $e \overset{eval}{\twoheadrightarrow} e'$, then e' also has type τ, where the symbol $\overset{eval}{\twoheadrightarrow}$ denotes the transitive closure of the $\xrightarrow{eval}$ relation. We then define an evaluation strategy *eval* that reduces expressions of the object calculus to values, a special "value" *error*, or is undefined if the expression in question fails to terminate. We then use subject reduction to show that if we may derive a type for a closed expression e, then $\mathit{eval}(e) \neq \mathit{error}$. Since the evaluator returns *error* if it fails to find a required method, this fact justfies the claim that our type system prevents "message not understood" errors.

The subject reduction proof begins with a collection of lemmas about substitution for row and type variables. These lemmas allow us to specialize polymorphic object types to include additional methods. In Section 4.2, we introduce a normal form for our typing derivations to permit us to restrict our attention to derivations where the only occurrences of equality rules are as $(row\ \beta)$ immediately following an occurrence of $(row\ fn\ app)$. This restriction greatly simplifies later proofs. We then prove a series of technical lemmas in Section 4.3. The first of these allows us to treat contexts, which are lists, more like sets. The remaining lemmas in the section provide tools for constructing row expressions to generalize polymorphic object types. The lemmas in Section 4.4 then show that the ($\xrightarrow{book}$), (β), and ($\Leftarrow$) components of the ($\xrightarrow{eval}$) relation preserve expression types. The subject reduction theorem then follows. In Section 4.5, we define the *eval* function and show that typable closed expressions do not evaluate to *error*, a fact which guarantees that we do not get dynamic message-not-understood errors.

4.1 Substitution Lemmas

The first two lemmas are used to prove that if we may derive a type or row judgement containing a type or row variable, then substituting a type or row expression of the appropriate kind produces a derivable judgement. Lemma 4.1 is the desired property, except it has the additional hypothesis that we may derive the judgement $\Gamma,\ [U_2/u_2]\Gamma' \vdash *$. This extra assumption is required for the (*projection*), (*weakening*), and (*empty row*) cases. Lemma 4.2 then uses Lemma 4.1 to show that the additional hypothesis is in fact unnecessary.

LEMMA 4.1. *If the judgement* $\Gamma,\ u_2 : V_2,\ \Gamma' \vdash U_1 : V_1$*, the judgement* $\Gamma \vdash U_2 : V_2$*, and the judgement* $\Gamma,\ [U_2/u_2]\Gamma' \vdash *$ *are all 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 of the form* $\tau : T$ *or* $R : \kappa$*.*

PROOF. The proof is by induction on the derivation of $\Gamma,\ u_2 : V_2,\ \Gamma' \vdash U_1 : V_1$. Most of the cases are either vacuous or follow immediately from the inductive hypothesis. The (*projection*) case requires a case analysis on whether or not u_2 is the projected variable. The (*row* β) and (*type* β) cases follow from the fact that if $U_1 \rightarrow_\beta U_2$, then $[U/u]U_1 \rightarrow_\beta [U/u]U_2$, where U_1 and U_2 are either both row expressions or both type expressions, U is either a row or type expression, and u is either a row or type variable, to match U. The (*weakening*) case requires the fact that $\Gamma \vdash *$ is derivable if $\Gamma \vdash A$ is. The (*class*) and (*row fn abs*) cases follow from the fact that bound variables may be consistently renamed as necessary. $\square$

LEMMA 4.2. *If* $\Gamma,\ u_2 : V_2,\ \Gamma' \vdash *$ *and* $\Gamma \vdash U_2 : V_2$ *are both derivable, then so is* $\Gamma, [U_2/u_2]\Gamma' \vdash *$*, where* $U_2 : V_2$ *is either* $\tau : T$ *or* $R : \kappa$*.*

PROOF. The proof of Lemma 4.2 is by induction on the length of Γ'. Lemma 4.1 is needed for the case where Γ' is extended via (*exp var*). $\square$

Using the fact that $\Gamma \vdash *$ is derivable if $\Gamma \vdash A$ is, we may combine Lemmas 4.1 and 4.2 to give the desired substitution property on type and kind derivations.

LEMMA 4.3. *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* $U_2 : V_2$ *may be either* $\tau : T$ *or* $R : \kappa$*.*

The following lemma is used in conjunction with Lemma 4.3 to specialize class types to contain additional methods.

LEMMA 4.4. *If* $\Gamma,\, r : T^n \rightarrow [\vec{m}],\, \Gamma' \vdash e : \tau$ *and* $\Gamma \vdash R : T^n \rightarrow [\vec{m}]$ *are both derivable, then so is* $\Gamma,\, [R/r]\Gamma' \vdash e : [R/r]\tau$*.*

PROOF. Lemma 4.4 follows by induction on the derivation of $\Gamma,\, r{:}T^n \rightarrow [\vec{m}]$, $\Gamma' \vdash e : \tau$. Lemma 4.2 is used in the (*projection*), (*weakening*), and (*empty object*) cases. Lemma 4.3 with U_2 as a row expression is needed for the (*obj ext*) case. □

4.2 Normal Form

The equality rules in the proof system introduce many non-essential judgement derivations, which unnecessarily complicate derivation analysis. We therefore restrict our attention to derivations in which the only appearance of an equality rule is as (*row* β) immediately following an occurrence of (*row fn app*). We will call these derivations $\vdash_N$-derivations. Although not all judgements derivable in the full system are derivable by $\vdash_N$-derivations, we will see below that all judgements whose row and type expressions are in a particular form, which we will call τnf (for type normal form), are derivable via $\vdash_N$-derivations. Since every expression that has a type at all will have a type in τnf, we may prove soundness using only $\vdash_N$-derivations.

The τnf of a type or row expression is its normal form with respect to β-reduction, applied to the row function application redexes within it. This notion is well-defined since the row and type portion of our calculus is strongly normalizing and confluent. We prove this fact by giving a translation function tr into $\lambda^{\rightarrow}(\Sigma)$, where $\lambda^{\rightarrow}(\Sigma)$ denotes the typed lambda calculus with function types over signature Σ, which is defined as:

Type Constants	:	$typ, meth$
Term Constants	:	$\mathtt{er} : meth$
		$\mathtt{ar} : typ \rightarrow typ \rightarrow typ$
		$\mathtt{cl} : (typ \rightarrow meth) \rightarrow typ$
		$\mathtt{br_m} : meth \rightarrow typ \rightarrow meth$
		for each method name m.

Let the variables of $\lambda^{\rightarrow}(\Sigma)$ contain all of the row and type variables of our calculus. Then define the translation tr by:

$$
\begin{aligned}
&tr(t) = t\\
&tr(\tau_1 \to \tau_2) = \mathtt{ar}\ tr(\tau_1)\ tr(\tau_2)\\
&tr(\mathbf{class}\, t\,\mathbf{.}R) = \mathtt{cl}\,(\lambda t : typ.tr(R))\\
&tr(r) = r\\
&tr(\langle\!\langle\rangle\!\rangle) = \mathtt{er}\\
&tr(\langle\!\langle R \,|\, m : \tau \rangle\!\rangle) = \mathtt{br_m}\ tr(R)\ tr(\tau)\\
&tr(\lambda t.R) = \lambda t : typ.tr(R)\\
&tr(R\tau) = tr(R)\ tr(\tau)
\end{aligned}
$$

We extend tr to the kinds and contexts of our system as follows:

$$
\begin{aligned}
&tr(T) = typ\\
&tr(T^n \to [\vec{m}]) = typ^n \to meth\\
\\
&tr(\epsilon) = \emptyset\\
&tr(\Gamma,\, x : \tau) = tr(\Gamma)\\
&tr(\Gamma,\, t : T) = tr(\Gamma) \cup \{tr(t) : tr(T)\}\\
&tr(\Gamma,\, r : \kappa) = tr(\Gamma) \cup \{tr(r) : tr(\kappa)\}
\end{aligned}
$$

Note that tr preserves both bound and free variables of expressions, i.e., $BV(U) = BV(tr(U))$ and $FV(U) = FV(tr(U))$ for all row and type expressions U. Furthermore, if $U_1 =_\alpha U_2$, then $tr(U_1) =_\alpha tr(U_2)$ under the same renaming of bound variables.

To show that strong normalization for our system follows from strong normalization for $\lambda^{\to}(\Sigma)$, we need to establish two properties of tr. We need to show that the translations of any two terms related via $\to_\beta$ in our system are related via $\to_\beta$ in $\lambda^{\to}(\Sigma)$ and that the translation of every well-kinded term in our system is a well-typed $\lambda^{\to}(\Sigma)$ term. Lemma 4.5 proves the first of these properties.

LEMMA 4.5. *If $U_1 \to_\beta U_2$, then $tr(U_1) \to_\beta tr(U_2)$, where U_1 and U_2 are either both row or both type expression in our system.*

PROOF. The proof of Lemma 4.5 is by induction on the structure of U_1. Each inductive case is a case analysis of the possible forms of U_2. The only case which does not follow routinely is when $U_1 = (\lambda t.R)\tau$ and $U_2 = [\tau/t]R$. This case follows from the subsidiary lemma that $[tr(\tau)/t]tr(U) = tr([\tau/t]U)$ for all type and row expressions U, which is proved by induction on the structure of U. $\square$

Lemma 4.6 establishes that tr produces typable $\lambda^{\to}(\Sigma)$ terms.

LEMMA 4.6. *If $\Gamma \vdash U : V$ is derivable in our system, then $tr(\Gamma) \rhd tr(U) : tr(V)$ is derivable in $\lambda^{\to}(\Sigma)$, where we use $\rhd$ to distinguish $\lambda^{\to}(\Sigma)$-derivations from derivations in our system.*

Proof. The proof of Lemma 4.6 is by induction on the derivation of $\Gamma \vdash U : V$. The cases for (*row* β) and (*type* β) require Lemma 4.5 and Subject Reduction for $\lambda^{\rightarrow}(\Sigma)$. □

Combining Lemmas 4.5 and 4.6 we get that the row and type portion of our calculus is strongly normalizing:

Lemma 4.7. (Strong Normalization for Row and Type Portion) *If $\Gamma \vdash U : V$ is derivable, where U is either a row or type expression, then there is no infinite sequence of $\rightarrow_\beta$ reductions from U.*

The following lemma is crucial to showing that confluence for $\lambda^{\rightarrow}(\Sigma)$ implies confluence for the row and type portion of our calculus.

Lemma 4.8. *If $tr(U) \rightarrow_\beta W$, then there is a unique expression U' such that $U \rightarrow_\beta U'$ and $tr(U') = W$, where U is either a row or type expression and W is an expression of $\lambda^{\rightarrow}(\Sigma)$.*

Proof. The proof of Lemma 4.8 is by induction on the structure of U. It is similar in outline to the proof of Lemma 4.5. □

The confluence of the row and type portion of our system now follows.

Lemma 4.9. (Confluence for Row and Type Portion) *If $\Gamma \vdash U_1 : V_1$ is derivable and $U_1 \twoheadrightarrow_\beta U_2$ and $U_1 \twoheadrightarrow_\beta U_3$, then there exists a U_4 such that $U_2 \twoheadrightarrow_\beta U_4$ and $U_3 \twoheadrightarrow_\beta U_4$.*

Since each row and type expression has a unique normal form, the τnf of row and type expressions is a well-defined notion. Because any term expression that has a type has a type in normal form, we may restrict our attention to types and rows in τnf. To this end, we need to extend the definition of τnf to term expressions and contexts. The τnf of a term expression e is just that expression e. The τnf of a context Γ is the context listing the τnf's of the elements of Γ. The following lemma then shows we can find a $\vdash_N$-derivation for any judgement in τnf.

Lemma 4.10. (Normal Form) *If $\Gamma \vdash A$ is derivable, then so is $\tau nf(\Gamma) \vdash_N \tau nf(A)$.*

Proof. 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. Most of the other cases follow immediately from the inductive hypothesis. We give the (*row fn app*) case in detail, since it is the most interesting.

If $\Gamma \vdash A$ is derived via (*row fn app*), then $\Gamma \vdash A$ must be of the form

$$\Gamma \vdash R\tau : T^n \rightarrow [\vec{m}].$$

By the hypotheses for $(row\ fn\ app)$, we must have previously derived

$$\Gamma \vdash R : T^{n+1} \rightarrow [\vec{m}]$$

and

$$\Gamma \vdash \tau : T.$$

By the inductive hypothesis, we know that

$$\tau nf(\Gamma) \underset{N}{\vdash} \tau nf(R : T^{n+1} \rightarrow [\vec{m}])$$

and

$$\tau nf(\Gamma) \underset{N}{\vdash} \tau nf(\tau : T)$$

are both derivable. Applying $(row\ fn\ app)$ to these two judgements produces

$$\tau nf(\Gamma) \underset{N}{\vdash} \tau nf(R) \tau nf(\tau) : T^n \rightarrow [\vec{m}].$$

There are two cases to consider: either $\tau nf(R)$ is a λ-abstraction or it is not.

Case 1: $\tau nf(R) = \lambda t.\tau nf(R')$**.** In this case, we have that

$$\tau nf(\Gamma) \underset{N}{\vdash} (\lambda t.\tau nf(R'))\tau nf(\tau) : T^n \rightarrow [\vec{m}].$$

Now

$$(\lambda t.\tau nf(R'))\tau nf(\tau) \underset{\beta}{\rightarrow} [\tau nf(\tau)/t]\tau nf(R'),$$

so applying $(row\ \beta)$ to the last judgement produces

$$\tau nf(\Gamma) \underset{N}{\vdash} [\tau nf(\tau)/t]\tau nf(R') : T^n \rightarrow [\vec{m}],$$

which is still a $\vdash_N$-derivation. Note that

$$[\tau nf(\tau)/t]\tau nf(R') = \tau nf(R\tau)$$

since substituting types in normal form for variables cannot introduce new β-reductions. (Substituting a row expression for a row variable could introduce a new β-reduction, but we do not allow abstraction over row variables so we need not worry about this case.) Thus the following judgement is derivable:

$$\tau nf(\Gamma) \underset{N}{\vdash} \tau nf(R\tau : T^n \rightarrow [\vec{m}]).$$

Case 2: $\tau nf(R)$ **is not a λ-abstraction.** In this case, the desired result follows immediately from the inductive hypothesis. □

From this point on, we will only concern ourselves with 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.

4.3 Technical Lemmas

The following lemma is useful because it allows contexts, which are in fact lists, to be treated more like sets.

LEMMA 4.11. *If* $\Gamma, \Gamma' \vdash A$ *and* $\Gamma, y:Z, \Gamma' \vdash *$ *are both derivable, then so is* $\Gamma, y:Z, \Gamma' \vdash A$*, where* $y:Z$ *can be* $r:\kappa$*,* $t:T$*, or* $x:\tau$*.*

PROOF. The proof of Lemma 4.11 is by induction on the derivation of $\Gamma, \Gamma' \vdash A$. The (*obj ext*) and (*obj over*) cases require a subsidiary lemma that permits the consistent renaming of the row variables in a judgement. This renaming is necessary to avoid naming-clashes. □

The remaining lemmas in this section are used to build well-formed row expressions that can be substituted for row variables in typing derivations. This ability is crucial to the proofs of Lemmas 4.15 and 4.17.

LEMMA 4.12. *If* $\Gamma \vdash_N \lambda t_1, \ldots, t_n.\langle\!\langle R \mid \vec{m} : \vec{\tau} \rangle\!\rangle : T^n \rightarrow [\vec{l}]$ *is derivable, then so are* $\Gamma, t_1:T, \ldots, t_n:T \vdash_N \tau_i : T$ *for each* τ_i *in* $\vec{\tau}$ *and* $\Gamma, t_1:T, \ldots, t_n:T \vdash_N R : [\vec{m}, \vec{l}]$*.*

PROOF. The proof of Lemma 4.12 is by induction on the derivation of $\Gamma \vdash_N \lambda t_1, \ldots, t_n.\langle\!\langle R \mid \vec{m} : \vec{\tau} \rangle\!\rangle : T^n \rightarrow [\vec{l}]$. Most of the cases follow routinely from the inductive hypothesis. The (*weakening*) case follows from repeated applications of Lemma 4.11. The (*row β*) case is worked out here, since it is the most difficult.

If $\Gamma \vdash_N \lambda t_1, \ldots, t_n.\langle\!\langle R \mid \vec{m} : \vec{\tau} \rangle\!\rangle : T^n \rightarrow [\vec{l}]$ is derived via (*row β*), then the previous step must have been (*row fn app*) by the definition of a $\vdash_N$-derivation. Furthermore, its hypotheses must have been of the form:

$$\Gamma \underset{N}{\vdash} \lambda t.\lambda t_1, \ldots, t_n.\langle\!\langle R^* \mid \vec{m} : \vec{\tau}^* \rangle\!\rangle : T^n \rightarrow [\vec{l}]$$

and

$$\Gamma \underset{N}{\vdash} \tau' : T$$

where

$$\lambda t_1, \ldots, t_n.\langle\!\langle R \mid \vec{m} : \vec{\tau} \rangle\!\rangle = [\tau'/t]\lambda t_1, \ldots, t_n.\langle\!\langle R^* \mid \vec{m} : \vec{\tau}^* \rangle\!\rangle.$$

Without loss of generality, $t_1, \ldots t_n \notin FV(\tau') \cup \{t\}$, so

$$\lambda t_1, \ldots, t_n.\langle\!\langle R \mid \vec{m} : \vec{\tau} \rangle\!\rangle = \lambda t_1, \ldots, t_n.\langle\!\langle [\tau'/t]R^* \mid \vec{m} : [\tau'/t]\vec{\tau}^* \rangle\!\rangle,$$

which implies that $\vec{\tau} = [\tau'/t]\vec{\tau}^*$ and $R = [\tau'/t]R^*$. By the inductive hypothesis, for each τ_i^* in $\vec{\tau}^*$

$$\Gamma, t:T, t_1:T, \ldots, t_n:T \underset{N}{\vdash} \tau_i^* : T$$

and

$$\Gamma, t:T, t_1:T, \ldots, t_n:T \underset{N}{\vdash} R^* : [\vec{m}, \vec{l}]$$

are derivable. By Lemma 4.3, for each τ_i^* in $\vec{\tau}^*$,

$$\Gamma, t_1 : T, \ldots, t_n : T \vdash [\tau'/t]\tau_i^* \, : \, T$$

and

$$\Gamma, t_1 : T, \ldots, t_n : T \vdash [\tau'/t]R^* \, : \, [\vec{m}, \vec{l}]$$

are derivable. Since $\vec{\tau} = [\tau'/t]\vec{\tau^*}$ and $R = [\tau'/t]R^*$, these are just:

$$\Gamma, t_1 : T, \ldots, t_n : T \vdash \tau_i \, : \, T$$

and

$$\Gamma, t_1 : T, \ldots, t_n : T \vdash R \, : \, [\vec{m}, \vec{l}].$$

The original derivation is in τnf, so Γ, $\vec{\tau}$, and R are all in τnf. Hence by Lemma 4.10, we get for each τ_i in $\vec{\tau}$

$$\Gamma, t_1 : T, \ldots, t_n : T \underset{N}{\vdash} \tau_i \, : \, T$$

and

$$\Gamma, t_1 : T, \ldots, t_n : T \underset{N}{\vdash} R \, : \, [\vec{m}, \vec{l}]$$

are derivable. $\square$

Lemma 4.13. *If* $\Gamma \vdash_N \mathbf{class}\, t.\langle\!\langle R \,|\, \vec{m} : \vec{\tau} \rangle\!\rangle : T$ *is derivable, then so are* $\Gamma, t : T \vdash_N \tau_i \, : \, T$ *for each* τ_i *in* $\vec{\tau}$ *and* $\Gamma, t : T \vdash_N R \, : \, [\vec{m}]$.

The proof of Lemma 4.13 is by induction on the derivation of $\Gamma \vdash_N \mathbf{class}\, t.\langle\!\langle R \,|\, \vec{m} : \vec{\tau} \rangle\!\rangle : T$ to handle the (*weakening*) case. Most of the cases are vacuous, since they could not have been the last step in the derivation of the form $\Gamma \vdash_N \mathbf{class}\, t.\langle\!\langle R \,|\, \vec{m} : \vec{\tau} \rangle\!\rangle : T$. The (*type* β) case follows vacuously because it cannot appear in a $\vdash_N$-derivation. The (*class*) case follows immediately from Lemma 4.12.

Lemma 4.14. *If* $\Gamma \vdash_N e \, : \, \tau$ *is derivable, then so is* $\Gamma \vdash_N \tau \, : \, T$.

Proof. The proof of Lemma 4.14 is by induction on the derivation of $\Gamma \vdash_N e \, : \, \tau$. Below is the case for (*obj ext*), which is the most difficult. The other cases follow for similar reasons or directly from the inductive hypothesis.

If $\Gamma \vdash_N e \, : \, \tau$ is derived via (*obj ext*), then the judgement must have the form:

$$\Gamma \underset{N}{\vdash} \langle e_1 \longleftarrow\!\!+\, n{=}e_2 \rangle \, : \, \mathbf{class}\, t.\langle\!\langle R \,|\, \vec{m} : \vec{\tau},\, n : \tau \rangle\!\rangle.$$

By the hypotheses for (*obj ext*), we must have previously derived

$$\Gamma \underset{N}{\vdash} e_1 \, : \, \mathbf{class}\, t.\langle\!\langle R \,|\, \vec{m} : \vec{\tau} \rangle\!\rangle.$$

Applying the inductive hypothesis to this produces the judgement:

$$\Gamma \underset{N}{\vdash} \mathbf{class}\, t \centerdot \langle\!\langle R \,|\, \vec{m} : \vec{\tau} \rangle\!\rangle \, : \, T.$$

By Lemma 4.13 then, we have that for all τ_i in $\vec{\tau}$

$$\Gamma,\, t : T \underset{N}{\vdash} \tau_i \, : \, T$$

is derivable. The hypotheses for $(obj\ ext)$ also give us that

$$\Gamma,\, t : T \underset{N}{\vdash} R \, : \, [\vec{m}, n]$$

is derivable. By $|\vec{m}|$ applications of $(row\ ext)$, we may derive

$$\Gamma,\, t : T \underset{N}{\vdash} \langle\!\langle R \,|\, \vec{m} : \vec{\tau} \rangle\!\rangle \, : \, [n].$$

The final hypothesis for $(obj\ ext)$ tells us that the judgement:

$$\Gamma,\, r \, : \, T \rightarrow [\vec{m}, n] \underset{N}{\vdash} e_2 \, : \, [\mathbf{class}\, t \centerdot \langle\!\langle rt \,|\, \vec{m} : \vec{\tau},\, n : \tau \rangle\!\rangle / t](t \rightarrow \tau)$$

is derivable. Applying the inductive hypothesis produces the judgement:

$$\Gamma,\, r \, : \, T \rightarrow [\vec{m}, n] \underset{N}{\vdash} [\mathbf{class}\, t \centerdot \langle\!\langle rt \,|\, \vec{m} : \vec{\tau},\, n : \tau \rangle\!\rangle / t](t \rightarrow \tau) \, : \, T.$$

It then follows from an easily proved property of derivations that

$$\Gamma,\, r \, : \, T \rightarrow [\vec{m}, n] \underset{N}{\vdash} \mathbf{class}\, t \centerdot \langle\!\langle rt \,|\, \vec{m} : \vec{\tau},\, n : \tau \rangle\!\rangle \, : \, T$$

is also derivable. Lemma 4.13 gives us that

$$\Gamma,\, r \, : \, T \rightarrow [\vec{m}, n],\, t \, : \, T \underset{N}{\vdash} \tau \, : \, T$$

is also derivable. By the side condition for $(row\ ext)$, $r \notin FV(\tau)$, so as a consequence of Lemma 4.3 we may derive

$$\Gamma,\, t \, : \, T \underset{N}{\vdash} \tau \, : \, T.$$

Applying row extension, we may derive the judgement

$$\Gamma,\, t \, : \, T \underset{N}{\vdash} \langle\!\langle R \,|\, \vec{m} : \vec{\tau}, n \, : \, \tau \rangle\!\rangle \, : \, \emptyset.$$

Applying $(class)$ to this judgement produces

$$\Gamma \underset{N}{\vdash} \mathbf{class}\, t \centerdot \langle\!\langle R \,|\, \vec{m} : \vec{\tau}, n \, : \, \tau \rangle\!\rangle \, : \, T,$$

which is the judgement we wished to derive. $\square$

4.4 Type Preservation Lemmas

The next three lemmas show that the various components of the $(\xrightarrow{eval})$ relation preserve expression types.

LEMMA 4.15. *If* $\Gamma \vdash_N e : \tau$ *is derivable, and* $e \xrightarrow{book} e'$, *then* $\Gamma \vdash_N e' : \tau$ *is also derivable.*

PROOF. The proof of Lemma 4.15 consists of two parts: the first shows that a derivation from $\Gamma \vdash e : \tau$ can only depend on the form of τ, not on the form of e. More formally, if $\Gamma \vdash C[e] : \tau$ is derived from $\Gamma' \vdash e : \sigma$ and $\Gamma' \vdash e' : \sigma$ is also derivable, then so is $\Gamma \vdash C[e'] : \tau$. This fact may be seen by an inspection of the typing rules. The second part shows that if $\Gamma \vdash e : \tau$ is derivable, and $e \xrightarrow{book} e'$ by e matching the left-hand side of one of the $(\xrightarrow{book})$ axioms, then $\Gamma \vdash e' : \tau$ is also derivable. This fact follows from a case analysis of the four $(\xrightarrow{book})$ axioms. Row variables and Lemmas 4.4, 4.13, and 4.14 are essential for the (*switch ext ov*) case. $\square$

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

LEMMA 4.16. *If* $\Gamma, x : \tau_1, \Gamma' \vdash e_2 : \tau_2$ *and* $\Gamma \vdash e_1 : \tau_1$ *are both derivable, then so is* $\Gamma, \Gamma' \vdash [e_1/x]e_2 : \tau_2$.

PROOF. The proof of Lemma 4.16 is by induction on the derivation of $\Gamma, x : \tau_1, \Gamma' \vdash e_2 : \tau_2$. Several of the cases require a lemma of the form that if $\Gamma, x : \tau, \Gamma' \vdash A$ is derivable with x not in A, then the judgement $\Gamma, \Gamma' \vdash A$ is also derivable. The proof of this sub-lemma is similar in structure to the proof of Lemma 4.3. $\square$

An immediate consequence of the next lemma is that $(\Leftarrow)$-reduction preserves expression types.

LEMMA 4.17. *If* $\Gamma \vdash_N \langle\langle e \leftarrow\!\!\ominus_1 m_1{=}e_1\rangle \ldots \leftarrow\!\!\ominus_k m_k{=}e_k\rangle : \mathbf{class}\, t\,\mathbf{.}\langle\!\langle R \,|\, m_1 : \tau_1, \ldots, m_k : \tau_k\rangle\!\rangle$ *is derivable, where* $m_1, \ldots, m_k$ *are distinct and are precisely the method names that occur consecutively to the right of* e, *then* $\Gamma \vdash e_i : [\mathbf{class}\, t\,\mathbf{.}\langle\!\langle R \,|\, m_1 : \tau_1, \ldots, m_k : \tau_k\rangle\!\rangle/t](t \rightarrow \tau_i)$ *is derivable.*

PROOF. Suppose $\Gamma \vdash_N \langle\langle e \leftarrow\!\!\ominus_1 m_1{=}e_1\rangle \ldots \leftarrow\!\!\ominus_k m_k{=}e_k\rangle$ $: \mathbf{class}\, t\,\mathbf{.}\langle\!\langle R \,|\, m_1{:}\tau_1, \ldots, m_k{:}\tau_k\rangle\!\rangle$ is derivable. The proof then has two parts. The first part, which establishes Claim 4.1 below, gives a derivable type for each object obtained from $\langle\langle e \leftarrow\!\!\ominus_1 m_1{=}e_1\rangle \ldots \leftarrow\!\!\ominus_k m_k{=}e_k\rangle$ by stripping off some number of the $\leftarrow\!\!\ominus$ operations. Names for these "subobjects" are introduced below. The second part, establishing Claim 4.2, shows that we may derive the required type for each of the e_i's. Claim 4.2 follows from an analysis of the typing judgements given by Claim 4.1.

To represent the subobjects of $\langle\langle e \leftarrow\!\!\!\ominus_1 m_1{=}e_1\rangle \ldots \leftarrow\!\!\!\ominus_k m_k{=}e_k\rangle$, we introduce the following notation:

$$\begin{array}{lcl} O_0 & \overset{def}{=} & e \\ O_j & \overset{def}{=} & \langle O_{j-1} \leftarrow\!\!\!\ominus_j m_j{=}e_j\rangle. \end{array}$$

Note that the object in the assumption of the lemma is just O_k. We also need to be able to name particular collections of the methods defined in O_k and their associated types. To this end, let $M_j = [m_{j+1}, \ldots, m_k]$ and M_j^- be the subsequence of M_j such that $m_i \in M_j^-$ if and only if $\leftarrow\!\!\!\ominus_i = \longleftarrow\!\!\!\!+$. Similarly, let $\Sigma_j = [\tau_{j+1}, \ldots, \tau_k]$ and Σ_j^- be the subsequence of Σ_j such that $\tau_i \in \Sigma_j^-$ if and only if $m_i \in M_j^-$. Note that $M_k = M_k^- = \epsilon$, $M_0 = [m_1, \ldots, m_k]$, and $M_0^- = [m_i | \leftarrow\!\!\!\ominus_i = \longleftarrow\!\!\!\!+]$. Thus M_0^- is the sequence of all methods in O_k that are not defined in O_0. It follows that the methods in $(M_0 - M_0^-)$ are those methods that are overridden in O_k. Similarly, M_j^- is the sequence of all methods in O_k that are not defined in O_j, and the methods in $(M_j - M_j^-)$ are those methods defined in O_j that are later overridden in O_k.

Lemma 4.17 follows immediately from the following two claims, written using the notation just defined:

Claim 4.1. *If* $\Gamma \vdash_N O_k : \mathbf{class}\, t.\langle\!\langle R \,|\, m_1 : \tau_1, \ldots, m_k : \tau_k \rangle\!\rangle$ *is derivable, then*

$$\Gamma \underset{N}{\vdash} O_{k-n} : \mathbf{class}\, t.\langle\!\langle R \,|\, m_1 : \tau_1, \ldots, m_{k-n} : \tau_{k-n},$$

$$(M_{k-n} - M_{k-n}^-) : (\Sigma_{k-n} - \Sigma_{k-n}^-)\rangle\!\rangle$$

is also derivable for all $0 \le n \le k$.

Claim 4.2. *If* $\Gamma \vdash_N O_k : \mathbf{class}\, t.\langle\!\langle R \,|\, m_1 : \tau_1, \ldots, m_k : \tau_k \rangle\!\rangle$ *and*

$$\Gamma \underset{N}{\vdash} O_i : \mathbf{class}\, t.\langle\!\langle R \,|\, m_1 : \tau_1, \ldots, m_i : \tau_i, (M_i - M_i^-) : (\Sigma_i - \Sigma_i^-)\rangle\!\rangle$$

are derivable, where $1 \le i \le k$, *then*

$$\Gamma \underset{N}{\vdash} e_i : [\mathbf{class}\, t.\langle\!\langle R \,|\, m_1 : \tau_1, \ldots, m_k : \tau_k \rangle\!\rangle / t](t \to \tau_i)$$

is also derivable.

The proof of Claim 4.1 is by induction on n. Claim 4.2 follows by an induction on the derivation of the typing for O_i. (An induction is necessary to handle the (*weakening*) case. Otherwise a simple case analysis would suffice.) The only cases besides (*weakening*) that need to be considered are (*obj ext*) and (*obj over*). Both of these rules have a hypothesis of the form:

$$\begin{array}{l} \Gamma',\, r : T \to [m_1, \ldots, m_i, (M_i - M_i^-)] \vdash_N \\ \quad e_i : [\mathbf{class}\, t.\langle\!\langle rt \,|\, m_1 : \tau_1, \ldots, m_i : \tau_i, (M_i - M_i^-) : (\Sigma_i - \Sigma_i^-)\rangle\!\rangle / t](t \to \tau_i). \end{array}$$

Lemmas 4.13 and 4.14 are used to build a row expression that is substituted via Lemma 4.4 for the row variable r in the above judgement to give the desired type to e_i. □

THEOREM 4.1. (SUBJECT REDUCTION) *If* $\Gamma \vdash e : \tau$ *is derivable, and* $e \xrightarrow{eval} e'$*, then* $\Gamma \vdash e' : \tau$ *is also derivable.*

PROOF. The proof is similar in outline to that of Lemma 4.15; 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 4.15, the (β) case from Lemma 4.16 and the ($\Leftarrow$) case from Lemma 4.17. □

4.5 Type Soundness

Now that we have subject reduction, we need to formalize the notion of message-not-understood errors to show that our type system prevents them. Intuitively, a message-not-understood error occurs when a message m is sent to an expression that does not define an object with an m-method. To formalize this notion, we define mutually recursive functions *eval* and get_m via proof rules in the style of structured operational semantics. The ideas behind this proof system are discussed below. The full system is given in Appendix B

The *eval* function is the standard lazy evaluator from lambda calculus, extended to our object calculus in the following way. Object expressions other than message sends evaluate to themselves. On expressions of the form $e \Leftarrow m$, *eval* uses the function get_m to extract the m method from e. This behavior is specified in the ($eval \ \Leftarrow$) proof rule:

$$(eval \ \Leftarrow) \qquad \frac{\begin{array}{c} get_m(e) = \langle e_1 \leftarrow m = e_2 \rangle \\ eval(e_2 \langle e_1 \leftarrow n = e_2 \rangle) = z \end{array}}{eval(e \Leftarrow m) = z}$$

which we may read as follows. Once get_m has extracted the m method from e by returning an expression of the form $\langle e_1 \leftarrow m = e_2 \rangle$, we "send" the message m to this expression by applying e_2 to the object. This resulting expression is then recursively evaluated to z, which is then returned as the value of the original message send. Meta-variable z is either an expression or the special "value" *error*.

How does get_m extract an m-method from its expression? There are two different forms of expressions from which get_m may immediately extract an m method: $\langle e_1 \leftarrow m = e_2 \rangle$ and $\langle e_1 \leftarrow\!\!+ \ m = e_2 \rangle$. The corresponding axioms are:

$$(getm \ \leftarrow) \qquad get_m(\langle e_1 \leftarrow m = e_2 \rangle) = \langle e_1 \leftarrow m = e_2 \rangle$$

$$(getm \ \leftarrow\!\!+) \qquad get_m(\langle e_1 \leftarrow\!\!+ \ m = e_2 \rangle) = \langle \langle e_1 \leftarrow\!\!+ \ m = e_2 \rangle \leftarrow m = e_2 \rangle$$

The second of these rules converts its object $\langle e_1 \leftarrow\!\!+\, m = e_2 \rangle$ to the equivalent object $\langle\langle e_1 \leftarrow\!\!+\, m = e_2\rangle \leftarrow m = e_2\rangle$ so that get_m returns objects in a standard form.

To extract an m method from more complicated expressions, we recursively use the *eval* and get_m functions. The $(getm \Leftarrow)$ rule is representative of these cases:

$$(getm\ \Leftarrow) \qquad \frac{\begin{array}{c} get_n(e) = \langle e_1 \leftarrow n = e_2 \rangle \\ get_m(e_2\langle e_1 \leftarrow n = e_2\rangle) = z \end{array}}{get_m(e \Leftarrow n) = z}$$

To find an m-method in an expression of the form $e \Leftarrow n$, we first find an n-method in the expression e by calling $get_n(e)$, which returns an expression of the form $\langle e_1 \leftarrow n = e_2\rangle$. We then "send" the object $\langle e_1 \leftarrow n = e_2\rangle$ the message n by extracting the n method and applying it to $\langle e_1 \leftarrow n = e_2\rangle$. We return the result of recursively looking for an m-method in the resulting expression.

How could get_m fail to find an m method? There are three different ways in which get_m may immediately "realize" that its object does not have the required method m. Its object could be a variable, a lambda abstraction, or the empty object $\langle\rangle$. These possibilities are described by the following three axioms:

$$(getm\ var) \qquad get_m(x) = error$$

$$(getm\ \langle\rangle) \qquad get_m(\langle\rangle) = error$$

$$(getm\ \lambda) \qquad get_m(\lambda x.\, e) = error$$

When called on more complicated expressions, get_m fails to find its desired method if one of its recursive calls fails. The $(getm \Leftarrow err)$ and $(getm \Leftarrow)$ rules are representative of these cases:

$$(getm\ \Leftarrow\ err) \qquad \frac{get_n(e) = error}{get_m(e \Leftarrow n) = error}$$

$$(getm\ \Leftarrow) \qquad \frac{\begin{array}{c} get_n(e) = \langle e_1 \leftarrow n = e_2 \rangle \\ get_m(e_2\langle e_1 \leftarrow n = e_2\rangle) = z \end{array}}{get_m(e \Leftarrow n) = z}$$

These rules reflect the fact that there are two ways we could fail to find an m-method in an expression of the form $e \Leftarrow n$. The first, described by $(getm \Leftarrow err)$, occurs when we cannot find an n-method in e. The second, described by the same $(getm \Leftarrow)$ rule that we saw above, occurs when we cannot find an m method in the expression obtained by invoking e's n-method.

We need to define these mutually recursive functions instead of using a single *eval* function because the notion of a value changes within the context of a message send. In particular, when we are not looking for a method, any object expression of the form $\langle e \leftarrow\!\!\circ\ m = e' \rangle$ is a value. If we are looking for an m method, then expressions of the form $\langle e \leftarrow m = e' \rangle$ are still values. However, if we are looking for an n method, $\langle e \leftarrow\!\!\circ\ m = e' \rangle$ is not a value and must be evaluated further.

Our specific *eval* and get_m functions are designed so that we may demonstrate that our type system prevents message-not-understood errors in programs. The same technique would allow us to show that terms typed as function expressions in programs either diverge or reduce to lambda abstractions. To do this, we would need to add a third evaluation function, get_λ, that "looks" for lambda abstractions and returns a tagged error value $function - error$ when called on an expression that cannot reduce to a lambda abstraction. To simplify the presentation, we only consider message-not-understood errors here.

Using the proof rules, we may show formally that typable programs of our object calculus do not produce message-not-understood errors.

Lemma 4.18. *If* $ev(e_1) = e_2$*, then* $e_1 \overset{eval}{\twoheadrightarrow} e_2$*, where* ev *is either* get_m *or* $eval$*.*

Proof. The proof is by induction on the derivation of $ev(e_1) = e_2$. The base case for the ($getm\ \leftarrow\!\!+$) axiom follows from the (*add ov*) bookkeeping rule. The other base cases are either vacuous, since *error* is not an expression, or immediate.

The inductive case for the ($getm\ \Leftarrow$) proof rule is given below since it is representative of the non-vacuous inductive cases. If we have derived $get_m(e \Leftarrow n) = e'$ via ($getm\ \Leftarrow$), then we must previously have derived that

$$get_n(e) = \langle e_1 \leftarrow n = e_2 \rangle$$

and

$$get_m(e_2 \langle e_1 \leftarrow n = e_2 \rangle) = e'.$$

Note that $get_m(e_2\langle e_1 \leftarrow n = e_2\rangle)$ cannot equal *error*, since we know that $get_m(e \Leftarrow n) = e'$ and e' is an expression, not *error*. Thus

$$\begin{aligned} e \Leftarrow n &\overset{eval}{\twoheadrightarrow} \langle e_1 \leftarrow n = e_2 \rangle \Leftarrow n \\ &\overset{eval}{\rightarrow} e_2 \langle e_1 \leftarrow n = e_2 \rangle \\ &\overset{eval}{\rightarrow} e' \end{aligned}$$

The first step above follows from the inductive hypothesis and the fact that the reduction rules are a congruence relation. The second step follows from the ($\Leftarrow$)-reduction rule, and the final one via the inductive hypothesis. $\square$

It remains to show that if we may derive $\emptyset \vdash e : \tau$, then $eval(e) \neq error$. Because there are two ways in which $eval(e)$ could not equal $error$, either by returning an expression or being undefined (which happens when e diverges under lazy evaluation), it is simpler to prove the contrapositive:

LEMMA 4.19. *If $ev(e) = error$, then*

- *if $ev = get_m$, then $\emptyset \not\vdash e : \mathbf{class}\, t\boldsymbol{.}\langle\!\langle R \,|\, m : \tau \rangle\!\rangle$ for any row R and type τ, and*
- *if $ev = eval$, then $\emptyset \not\vdash e : \tau$ for any type τ,*

where $\Gamma \not\vdash A$ indicates that the judgment $\Gamma \vdash A$ is not derivable.

PROOF. The proof is by induction on the derivation of $ev(e) = error$. The base cases are either vacuous or follow by inspection of the typing rules. We give the inductive cases for the $(getm \Leftarrow err)$ and $(getm \Leftarrow)$ rules, since they are representative of the non-vacuous inductive cases.

$(getm \Leftarrow err)$ If we derive $ev(e) = error$ via $(getm \Leftarrow err)$, then ev is get_m, e must be of the form $e' \Leftarrow n$, and we must have previously derived that $get_n(e') = error$. Applying the inductive hypothesis, we get that

$$\emptyset \not\vdash e' : \mathbf{class}\, t\boldsymbol{.}\langle\!\langle R' \,|\, n : \tau' \rangle\!\rangle$$

for any row R' and any type τ'. Then an inspection of the typing rule for message send $(meth\ app)$ reveals that

$$\emptyset \not\vdash e' \Leftarrow n : \tau''$$

for any type τ''. *A fortiori*,

$$\emptyset \not\vdash e' \Leftarrow n : \mathbf{class}\, t\boldsymbol{.}\langle\!\langle R \mid m : \tau \rangle\!\rangle$$

for any row R and type τ, which is what we needed to show.

$(getm \Leftarrow)$ If we derive $ev(e) = error$ via $(getm \Leftarrow)$, then ev is get_m and e must be of the form $e' \Leftarrow n$. Also, we must have previously derived that $get_n(e') = \langle e_1 \leftarrow n = e_2 \rangle$ and $get_m(e_2 \langle e_1 \leftarrow n = e_2 \rangle) = error$. Applying the inductive hypothesis to the second of these equations, we get

$$\emptyset \not\vdash e_2 \langle e_1 \leftarrow n = e_2 \rangle : \mathbf{class}\, t\boldsymbol{.}\langle\!\langle R \,|\, m : \tau \rangle\!\rangle.$$

By Lemma 4.18, $e' \overset{eval}{\twoheadrightarrow} \langle e_1 \leftarrow n = e_2 \rangle$, so

$$\begin{array}{rcl} e' \Leftarrow n & \overset{eval}{\twoheadrightarrow} & \langle e_1 \leftarrow n = e_2 \rangle \Leftarrow n \\ & \overset{eval}{\rightarrow} & e_2 \langle e_1 \leftarrow n = e_2 \rangle. \end{array}$$

Since $\emptyset \not\vdash e_2 \langle e_1 \leftarrow n = e_2 \rangle : \mathbf{class}\, t\boldsymbol{.}\langle\!\langle R \,|\, m : \tau \rangle\!\rangle$, we get by Subject Reduction (Theorem 4.1) that

$$\emptyset \not\vdash e' \Leftarrow n : \mathbf{class}\, t\boldsymbol{.}\langle\!\langle R \,|\, m : \tau \rangle\!\rangle,$$

which is what we needed to show. □

THEOREM 4.2. (TYPE SOUNDNESS) *If the judgement $\emptyset \vdash e : \tau$ is derivable, then $eval(e) \neq error$.*

5. Expressive power

Since the function part of our calculus is simply typed, every typable expression that does not contain object operations is strongly normalizing. It follows that using only function expressions, only a subset of the total recursive functions are representable. In contrast, we show that all partial recursive functions are representable in the calculus of functions and objects. Intuitively, this is because class types are a form of recursive type that allows self application. We begin with a simple example of a nonterminating expression, illustrating the failure of strong normalization for typable object expressions.

Divergent Computation

The object Ω has one method $\mathtt{m}$, which sends the message $\mathtt{m}$ to the object itself. Consequently, the evaluation of $\Omega \Leftarrow \mathtt{m}$ produces an infinite sequence of message sends.

$$\begin{array}{lcl} \Omega & \overset{def}{=} & \langle \mathtt{m} = \lambda\mathtt{self}.(\mathtt{self} \Leftarrow \mathtt{m})\rangle \\ & : & \mathbf{class}\, t.\langle\!\langle \mathtt{m} : t \rangle\!\rangle \\[1em] \Omega \Leftarrow \mathtt{m} & \xrightarrow{eval} & (\lambda\mathtt{self}.(\mathtt{self} \Leftarrow \mathtt{m}))\,\Omega \\ & \xrightarrow{eval} & \Omega \Leftarrow \mathtt{m} \end{array}$$

Fixed-point Operator

Before considering natural numbers and their basic operations, we show how to define a fixed-point operator on any type. The main idea is illustrated by the following object $\mathtt{X}$, whose $\mathtt{rec}$ (for "recurse") method causes the function $\mathtt{f}$ to be applied to $\mathtt{X} \Leftarrow \mathtt{rec}$.

$$\begin{array}{lcl} \mathtt{X} & \overset{def}{=} & \langle \mathtt{rec} = \lambda\mathtt{self}.\,\mathtt{f}(\mathtt{self} \Leftarrow \mathtt{rec})\rangle \\[1em] \mathtt{X} \Leftarrow \mathtt{rec} & \xrightarrow{eval} & \mathtt{f}(\mathtt{X} \Leftarrow \mathtt{rec}) \end{array}$$

We define a general fixed point operator by treating the object $\mathtt{X}$ above as a function of $\mathtt{f}$. Using the context

$$\begin{array}{lcl} \Gamma & = & \mathtt{f} : \sigma \to \sigma, \\ & & r : T \to [\mathtt{rec}], \\ & & \mathtt{x} : \mathbf{class}\, t.\langle\!\langle rt \mid \mathtt{rec} : \sigma \rangle\!\rangle \end{array}$$

the following typing derivation shows that the resulting function has type $(\sigma \to \sigma) \to \sigma$, for any σ that does not contain t.

$$\begin{array}{rcl}
\Gamma & \vdash & \mathtt{f(x \Leftarrow rec)} : \sigma \\
\Gamma - \mathtt{x} & \vdash & \lambda\mathtt{x}.\,\mathtt{f(x \Leftarrow rec)} \\
 & & : \mathbf{class}\, t.\langle\!\langle rt \mid \mathtt{rec} : \sigma \rangle\!\rangle \rightarrow \sigma \\
\Gamma - \mathtt{x} - r & \vdash & \langle \mathtt{rec}{=}\, \lambda\mathtt{x}.\,\mathtt{f(x \Leftarrow rec)} \rangle \\
 & & : \mathbf{class}\, t.\langle\!\langle \mathtt{rec} : \sigma \rangle\!\rangle \\
\Gamma - \mathtt{x} - r & \vdash & \langle \mathtt{rec}{=}\, \lambda\mathtt{x}.\,\mathtt{f(x \Leftarrow rec)} \rangle \Leftarrow \mathtt{rec} : \sigma \\
\epsilon & \vdash & \lambda\mathtt{f}.\langle \mathtt{rec}{=}\, \lambda\mathtt{x}.\,\mathtt{f(x \Leftarrow rec)} \rangle \Leftarrow \mathtt{rec} \\
 & & : (\sigma \rightarrow \sigma) \rightarrow \sigma
\end{array}$$

Object Numerals

The representation of natural numbers by object expressions is more complicated than the preceding examples. The main idea is to represent the natural number n by an object $\underline{n}$ with a method that sends its formal parameter the message `m` a total of n times. This is similar in spirit to the Church numeral $\lambda f.\,\lambda x.\,f^n x$, where the number n is represented using a function that does n function applications. The differences are that the numeral for n involves n message sends instead of n function applications, and that since message/method names cannot be parameters, the message that is sent n times must be fixed, rather than being a formal parameter. While it is possible to type the Church numerals in our calculus, we have not been able to type a predecessor or equality function on Church numerals.

The "representation" of a number is "stored" as the `rep` method of the numeral. In addition to the `rep` method, it is convenient to include successor and predecessor as methods. This gives us numerals with three methods: `S` for successor, `P` for predecessor, and `rep` for "representation." Numerals have type

$$\begin{array}{rcl}
nat & \overset{def}{=} & \mathbf{class}\, t.\langle\!\langle \mathtt{S} : t,\ \mathtt{P} : t, \\
 & & \qquad \mathtt{rep} : test(t) \rightarrow test(t) \rangle\!\rangle \\
test(t) & \overset{def}{=} & \mathbf{class}\, s.\langle\!\langle \mathtt{m} : s,\ \mathtt{b} : t \rightarrow t \rightarrow t \rangle\!\rangle
\end{array}$$

where the type of `rep` is chosen to make it easy to test whether a number is zero and evaluate one of two argument expressions accordingly.

The numeral for n is defined as follows, using the combinator `K` and postponing the full definition of predecessor until after zero test is defined.

$$\begin{array}{rcl}
\underline{n} & \overset{def}{=} & \langle \quad \mathtt{rep}{=}\mathtt{K}(\lambda\mathtt{x}.\,\mathtt{x} \Leftarrow \overbrace{\mathtt{m} \Leftarrow \mathtt{m} \Leftarrow \ldots \Leftarrow \mathtt{m}}^{n}), \\
 & & \quad \mathtt{S}{=}\, \lambda\mathtt{self}.\langle \mathtt{self} \leftarrow \\
 & & \qquad \mathtt{rep}{=}\mathtt{K}(\lambda\mathtt{x}.(\mathtt{self} \Leftarrow \mathtt{rep}\,\mathtt{x}) \Leftarrow \mathtt{m})\rangle, \\
 & & \quad \mathtt{P}{=}\, \lambda\mathtt{self}.\langle \mathtt{self} \leftarrow \mathtt{rep}{=}\mathtt{K}(\lambda\mathtt{x}.\ \ldots\ )\rangle \\
 & & \rangle
\end{array}$$

It is easy to see that successor increments the number of times `m` is sent by `rep` without relying on the predecessor method. For each n, we may show $\underline{n} : nat$.

Zero test

The function `IFZ`, for zero test, has the property that $\texttt{IFZ}\,\underline{n}$ yields $\texttt{true} \overset{def}{=} \lambda\texttt{x}.\,\lambda\texttt{y}.\,\texttt{x} : nat \rightarrow nat \rightarrow nat$ if $n = 0$ and $\texttt{false} \overset{def}{=} \lambda\texttt{x}.\,\lambda\texttt{y}.\,\texttt{y} : nat \rightarrow nat \rightarrow nat$ otherwise. As a result, $\texttt{IFZ}\,\underline{n}\; e_1\; e_2$ is e_1 if $n = 0$ and e_2 otherwise. The functions `true` and `false` are used in the objects

$$\begin{array}{lcl} \texttt{F} & \overset{def}{=} & \langle \texttt{m}{=}\,\lambda\texttt{x}.\,\texttt{x},\ \texttt{b}{=}\texttt{K}\,\texttt{false}\rangle \\ \texttt{T} & \overset{def}{=} & \langle \texttt{m}{=}\texttt{K}\,\texttt{F},\ \texttt{b}{=}\texttt{K}\,\texttt{true}\rangle \end{array}$$

with the properties that $\texttt{T} \Leftarrow \texttt{m} \xrightarrow{eval} \texttt{F}$, $\texttt{F} \Leftarrow \texttt{m} \xrightarrow{eval} \texttt{F}$, $\texttt{T} \Leftarrow \texttt{b} \xrightarrow{eval} \texttt{true}$ and $\texttt{F} \Leftarrow \texttt{b} \xrightarrow{eval} \texttt{false}$. Using these objects, we define zero test as follows.

$$\begin{array}{lcl} \texttt{IFZ} & \overset{def}{=} & \lambda\texttt{x}.((\texttt{x} \Leftarrow \texttt{rep})\ \texttt{T}) \Leftarrow \texttt{b} \\ \texttt{IFZ}\,\underline{n} & \xrightarrow{eval} & \texttt{T}\overbrace{\Leftarrow \texttt{m} \ldots \Leftarrow \texttt{m}}^{n} \Leftarrow \texttt{b} \\ & \xrightarrow{eval} & \begin{cases} \texttt{true} & \text{if } n = 0 \\ \texttt{false} & \text{if } n > 0 \end{cases} \end{array}$$

Predecessor

In rough terms, the main idea for predecessor is to pass an object of the form $\langle \texttt{m}{=}\texttt{x},\ \texttt{b}{=}\texttt{K}(\ldots)\rangle$ to the `rep` method of the object. (The `b` method is not used, so any function of the right type will do.) If `rep` sends this object the message `m` n times, the net effect will be to send `x` the message `m` $n-1$ times. This gives us a way of decrementing the number of times `m` is sent to the formal parameter of a function. Zero test, `IFZ`, is used to take care of the special case $n = 0$, where the result would otherwise be the object $\langle \texttt{m}{=}\texttt{x},\ \texttt{b}{=}\texttt{K}(\ldots)\rangle$, rather than the numeral for 0. The predecessor method of each numeral is defined as follows.

$$\begin{array}{l} \texttt{P}{=}\,\lambda\texttt{self}.\langle \texttt{self} \leftarrow \texttt{rep}{=}\texttt{K}(\lambda\texttt{x}.(\\ \qquad (\texttt{IFZ}\ \texttt{self}\ \underline{1}\ \texttt{self}) \Leftarrow \texttt{rep}\ \langle \texttt{m}{=}\texttt{x},\ \texttt{b}{=}\texttt{K}(\ldots)\rangle))\rangle \end{array}$$

Since the function $(\texttt{IFZ}\ \texttt{self}\ \underline{1}\ \texttt{self}) \Leftarrow \texttt{rep}$ sends `m` to $\langle \texttt{m}{=}\texttt{x},\ \texttt{b}{=}\texttt{K}(\ldots)\rangle$ a total of $min\{1, \texttt{self}\}$ times, the predecessor method works as follows.

$$
\begin{aligned}
\mathtt{P}\ \underline{n} \quad &\xrightarrow{eval} \quad \langle \underline{n} \leftarrow \mathtt{rep}{=}\mathtt{K}(\lambda\mathtt{x}. \\
&\qquad\qquad (\mathtt{IFZ}\ \underline{n}\ \underline{1}\ \underline{n}) \Leftarrow \mathtt{rep} \\
&\qquad\qquad\quad \langle \mathtt{m}{=}\mathtt{x},\ \mathtt{b}{=}\mathtt{K}(\ldots)\rangle)\rangle \\
&\xrightarrow{eval} \quad \langle \underline{n} \leftarrow \mathtt{rep}{=}\mathtt{K}(\lambda\mathtt{x}. \\
&\qquad\quad \langle \mathtt{m}{=}\mathtt{x},\ \mathtt{b}{=}\mathtt{K}(\ldots)\rangle \underbrace{\Leftarrow \mathtt{m} \Leftarrow \mathtt{m} \ldots \Leftarrow \mathtt{m}}_{min\{1,n\}})\rangle \\
&\xrightarrow{eval} \quad \langle \underline{n} \leftarrow \mathtt{rep}{=}\mathtt{K}(\lambda\mathtt{x}.\,\mathtt{x} \underbrace{\Leftarrow \mathtt{m} \ldots \Leftarrow \mathtt{m}}_{min\{1,n\}-1})\rangle
\end{aligned}
$$

This concludes the set of basic functions on natural numbers that are needed to represent all partial recursive functions.

6. Conclusion

We have given a computationally expressive typed calculus of functions and objects with a sound type system. In this “kernel language,” we define objects and their interfaces (called “classes” here) directly instead of through some subsidiary calculus of records. This gives an axiomatic presentation of a simple object-oriented language and its type system in a form that we hope is conducive to further work. A feature of the calculus is method specialization: using method redefinition (expressions of the form $\langle e \leftarrow n{=}e' \rangle$), we may define functions whose type and behavior change in a natural and useful way as a result of inheritance. This capability seems very difficult to achieve directly with any calculus of records. While it seems too early to claim that we have captured “the essence of inheritance in a simple form,” it seems that some progress has been made in this direction.

There are many technical open problems, including development of a denotational model, a proof system for equivalence that is sufficiently powerful to derive nontrivial equations between method bodies, and the investigation of subtyping and substitutivity in this language. Hopefully, the calculus presented here will provide a basis for studying these mathematical problems in a manner that is faithful to substantial uses of object-oriented programming in practice.

References

[1] M. Abadi. Baby Modula-3 and a theory of objects. Technical Report 95, DEC Systems Research Center, 1993. To appear in *J. Functional Prog.*

[2] N. Adams and J. Rees. Object-oriented programming in Scheme. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, pages 277–288, July 1988.

[3] L. Cardelli, J. Donahue, L. Galssman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical Report SRC-31, DEC Systems Research Center, 1988.

[4] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The Modula-3 type system. In *Sixteenth ACM Symp. Principles of Programming Languages*, pages 202–212, 1989.

[5] L. Cardelli and J.C. Mitchell. Operations on records. *Math. Structures in Computer Science*, 1(1):3–48, 1991. Summary in *Math. Foundations of Prog. Lang. Semantics*, Springer LNCS 442, 1990, pp 22–52.
[6] W.R. Cook. A proposal for making Eiffel type-safe. In *European Conf. on Object-Oriented Programming*, pages 57–72, 1989.
[7] 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.
[8] M. Ellis and B. Stroustrop. *The Annotated C^{++} Reference Manual.* Addison-Wesley, 1990.
[9] A. Goldberg and D. Robson. *Smalltalk–80: The language and its implementation.* Addison Wesley, 1983.
[10] B. Meyer. *Eiffel: The Language.* Prentice-Hall, 1992.
[11] J.C. Mitchell, F. Honsell, and K. Fisher. A lambda calculus of objects and method specialization. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 26–38, 1993.
[12] 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.
[13] J. Rees and N. Adams. T, a dialect of Lisp, or lambda: the ultimate software tool. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, pages 114–122, August 1982.
[14] D. Ungar and R.B. Smith. Self: The power of simplicity. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 227–241, 1987.

Appendix A. Typing rules

General Rules

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

$(projection)$ $$\frac{\Gamma \vdash * \qquad u : v \in \Gamma}{\Gamma \vdash u : v}$$

$(weakening)$ $$\frac{\Gamma \vdash A \qquad \Gamma, \Gamma' \vdash *}{\Gamma, \Gamma' \vdash A}$$

Rules for type expressions

$(type\ var)$ $$\frac{\Gamma \vdash * \qquad t \notin dom(\Gamma)}{\Gamma, t : T \vdash *}$$

$(type\ arrow)$ $$\frac{\Gamma \vdash \tau_1 : T \qquad \Gamma \vdash \tau_2 : T}{\Gamma \vdash \tau_1 \rightarrow \tau_2 : T}$$

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

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 or *class* and applications of the principle
$$\langle\!\langle \langle\!\langle R \,|\, n : \tau_1 \rangle\!\rangle \,|\, m : \tau_2 \rangle\!\rangle = \langle\!\langle \langle\!\langle R \,|\, m : \tau_2 \rangle\!\rangle \,|\, n : \tau_1 \rangle\!\rangle$$
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*) $$\frac{\Gamma, t : T \vdash R : T^n \rightarrow [m_1, \ldots, m_k]}{\Gamma \vdash \lambda t.\, R : T^{n+1} \rightarrow [m_1, \ldots, m_k]}$$

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

Rules for assigning types to terms

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

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

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

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

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

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

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

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

Appendix B. Definition of evaluation strategy

Axioms

In the following, meta-variable z represents either an expression or $error$.

$(getm\ var)$ $$get_m(x) = error$$

$(getm\ \langle\rangle)$ $$get_m(\langle\rangle) = error$$

$(getm\ \lambda)$ $$get_m(\lambda x.\, e) = error$$

$(getm\ \leftarrow)$ $$get_m(\langle e_1 \leftarrow m = e_2\rangle) = \langle e_1 \leftarrow m = e_2\rangle$$

$(getm\ \leftarrow\!+)$ $$get_m(\langle e_1 \leftarrow\!+\, m = e_2\rangle) = \langle\langle e_1 \leftarrow\!+\, m = e_2\rangle \leftarrow m = e_2\rangle$$

$(eval\ var)$ $$eval(x) = x$$

$(eval\ \langle\rangle)$ $$eval(\langle\rangle) = \langle\rangle$$

$(eval\ \lambda)$ $$eval(\lambda x.\, e) = \lambda x.\, e$$

$(eval\ \leftarrow)$ $$eval(\langle e_1 \leftarrow m = e_2\rangle) = \langle e_1 \leftarrow m = e_2\rangle$$

$(eval\ \leftarrow\!+)$ $$eval(\langle e_1 \leftarrow\!+\, m = e_2\rangle) = \langle e_1 \leftarrow\!+\, m = e_2\rangle$$

Inference Rules

$(getm\ app\ err)$ $$\frac{eval(e_1) = error}{get_m(e_1e_2) = error}$$

$(getm\ app)$ $$\frac{\begin{array}{c} eval(e_1) = \lambda x.\, e_1' \\ get_m([e_2/x]e_1') = z \end{array}}{get_m(e_1e_2) = z}$$

$(getm \Leftarrow err)$ $$\frac{get_n(e) = error}{get_m(e \Leftarrow n) = error}$$

$$(getm\ \Leftarrow) \qquad \frac{\begin{array}{c} get_n(e) = \langle e_1 \leftarrow n = e_2 \rangle \\ get_m(e_2 \langle e_1 \leftarrow n = e_2 \rangle) = z \end{array}}{get_m(e \Leftarrow n) = z}$$

$$(getm\ \leftarrow\ err) \qquad \frac{\begin{array}{c} n \neq m \\ get_m(e_1) = error \end{array}}{get_m(\langle e_1 \leftarrow n = e_2 \rangle) = error}$$

$$(getm\ \leftarrow) \qquad \frac{\begin{array}{c} n \neq m \\ get_m(e_1) = \langle e_3 \leftarrow m = e_4 \rangle \end{array}}{get_m(\langle e_1 \leftarrow n = e_2 \rangle) = \langle \langle e_3 \leftarrow n = e_2 \rangle \leftarrow m = e_4 \rangle}$$

$$(getm\ \leftarrow\!\!\!+\ err) \qquad \frac{\begin{array}{c} n \neq m \\ get_m(e_1) = error \end{array}}{get_m(\langle e_1 \leftarrow\!\!\!+\ n = e_2 \rangle) = error}$$

$$(getm\ \leftarrow\!\!\!+) \qquad \frac{\begin{array}{c} n \neq m \\ get_m(e_1) = \langle e_3 \leftarrow m = e_4 \rangle \end{array}}{get_m(\langle e_1 \leftarrow\!\!\!+\ n = e_2 \rangle) = \langle \langle e_3 \leftarrow\!\!\!+\ n = e_2 \rangle \leftarrow m = e_4 \rangle}$$

$$(eval\ app\ err) \qquad \frac{eval(e_1) = error}{eval(e_1 e_2) = error}$$

$$(eval\ app) \qquad \frac{\begin{array}{c} eval(e_1) = \lambda x.\, e_1' \\ eval([e_2/x]e_1') = z \end{array}}{eval(e_1 e_2) = z}$$

$$(eval\ \Leftarrow\ err) \qquad \frac{get_n(e) = error}{eval(e \Leftarrow n) = error}$$

$$(eval\ \Leftarrow) \qquad \frac{\begin{array}{c} get_n(e) = \langle e_1 \leftarrow n = e_2 \rangle \\ eval(e_2 \langle e_1 \leftarrow n = e_2 \rangle) = z \end{array}}{eval(e \Leftarrow n) = z}$$