Toward a typed foundation for method specialization and inheritance

John C. Mitchell *
Department of Computer Science
Stanford University
jcm@cs.stanford.edu

Abstract

This paper discusses the phenomenon of *method specialization* in object-oriented programming languages. A typed function calculus of objects and classes is presented, f eaturing method specialization when methods are added or redefined. The soundness of the typing rules (without subtyping) is suggested by a translation into a more traditional calculus with recursively-defined record types. However, semantic questions regarding the subtype relation on classes remain open.

1 Introduction

In spite of the increasing popularity of object-oriented programming, several issues do not seem to be well understood. In particular, although preliminary formal semantics have been proposed [Kam88, Red88, Yel89], there is neither an accepted basis for reasoning about basic issues such as program transformation or optimization, nor a sound basis for flexible typing disciplines. This paper presents a typed function calculus with simple forms of "objects" and "classes" which illustrate an essential feature of inheritance we call method specialization. To give some insight into the connection between this calculus and previous formal analysis, we also give a translation of objects and classes into records and recursively-defined record types. This clarifies some of the challenges involved in giving compositional, typed semantics to realistic object-oriented languages.

Apart from typing and mathematical semantics, the basic calculus used in this paper is relatively straightforward. The main idea is to provide a functional (*i.e.*, side-effect free) form of "prototyping," or "delegation" [Bor86, Lie86, LTP86, US87] so that one object may be created by inheriting methods from another. For simplicity, we treat methods and instance variables uniformly; methods may be replaced, and therefore instance variables may be regarded as methods that return a constant value. The set of messages an object will answer, and the types of their results, are specified by a form of type we call a class. One class will be considered a subclass (or subtype) of another if every object of the first is guaranteed to behave properly when considered as an element of the second class. Thus, in our view, classes are types whose elements are objects, and inheritance is a mechanism for constructing one object from another. The subclass relation is determined by behavioral characteristics of objects rather than program declarations. While this may not be the predominant view of object-oriented programming, it is consistent with at least one important practical view and it is a convenient model for our purposes.

Method specialization, which is described in some detail in Section 2, is achieved by treating objects as collections of functions, each representing a method of the object. When a method is invoked, the appropriate function is applied to the object itself. In other words, instead of using a special symbol *self* to allow a method to refer to the object to which it belongs, we use the first argument of the method. This approach is also used directly in T [RA82, AR88], which we were not aware of when we first began experimenting with this idea, and in the implementation of Modula 3 [CDG+88, CDJ+89]. The main point of the paper is not to promote this view of objects, but to develop

*Supported in part by an NSF PYI Award, matching funds from Digital Equipment Corporation, the Powell Foundation, and Xerox Corporation, and NSF grant CCR-8814921.

typing rules for methods which usefully reflect the way they are inherited.

One long-term goal is to develop a flexible, polymorphic typing discipline which could prevent such common run-time errors as *message not understood*. This is not an easy task, as illustrated by the vagaries of the early proposals for typing in Smalltalk [BI82, Suz81] and the subtle bugs surrounding *like self* [Coo89b] in the more recent language Eiffel [Mey88]. Another reason to develop typing rules is that in giving types, we are forced to specify exactly what kind of value is defined by each kind of expression in the language. This seems quite valuable when we consider substitution equivalence, which is critical to understanding or reasoning about transformation and optimization.

The calculus presented in this paper owes much to the recent line of work on record calculi with subtyping, beginning with Cardelli's 1984 paper [Car88]. A number of influential typing ideas, including bounded quantification, were sketched out in [CW85], and summarized in [DT88]. More recently, type inference techniques been presented in [Wan87], followed up by [Sta88, JM88, Rém89, Wan89]. From an untyped, denotational point of view, the primary studies seem to be Cook's thesis [Coo89a], which highlights method specialization, and the denotational semantics presented in [Kam88, Red88, Yel89]. The general perspective of this paper has developed from a tutorial presentation at the 1988 OOPSLA conference with Luca Cardelli [CM88], a subsequent joint paper [CM89], and numerous conversations with members of the ABEL group at HP Labs (Peter Canning, William Cook, Walt Hill and Walter Olthoff).

2 Method specialization

An important phenomena that seems essential to object-oriented programming will be referred to as *method specialization*. Although there is really only one basic idea, it will be helpful to separate method specialization into two forms, one involving the addition of methods, and the other involving method replacement, or "overriding." Both forms may be illustrated using example classes of points (*c.f.* [CW85, JM88]). The class *point* contains objects that have x, y and *move* methods. If points have integer coordinates, then the functionality of point objects may be summarized by the signature,

$$\begin{aligned}&\textbf{class } \textit{point} \textbf{ methods}\\ &\qquad x : \textit{int},\ y : \textit{int},\\ &\qquad \textit{move} : \textit{int} \times \textit{int} \rightarrow \textit{point}\end{aligned}$$

which we will regard as the type of all objects having x, y, and *move* methods of the indicated functionality. Note that we allow methods to return functions, which reduces method parameterization to ordinary function application. A more specialized class of points are the colored points, which have an additional method returning their color

$$\begin{aligned}&\textbf{class } \textit{colored\_point} \textbf{ methods}\\ &\qquad x : \textit{int},\ y : \textit{int},\ c : \textit{color},\\ &\qquad \textit{move} : \textit{int} \times \textit{int} \rightarrow \textit{colored\_point}\end{aligned}$$

and an appropriately revised type of *move* method. In a language such as Smalltalk [GR83], we might first define a *point* class and use *point* objects in writing a graphics package. Later, after upgrading to a color display, we might define the subclass of *colored_point*'s and use these instead. In a delegation– or prototype–based language such as Self, we might use a similar programming technique, although we would define the basic point methods in a prototype point, instead of a class declaration. An important aspect of object-oriented languages in general is that much of the code we write for *point*'s may be used directly on *colored_point*'s, eliminating what could otherwise be a significant amount of reprogramming.

When we define *colored_point* 's, either as a subclass (as in Smalltalk) or by prototyping (as in Self), the *move* method is *specialized* as it is inherited. In particular, the type of *move* changes when it is inherited. If we send the *move* message to a point, along with integer "displacements" δ_x and δ_y, we obtain a point with modified x and y coordinates. However, when we send the *move* message to a colored point, we obtain a colored point instead of an uncolored point. While this will be completely familiar to anyone who has written a program in an object-oriented language, it is worth noting that it is difficult to simulate this behavior within traditional typed languages such as Pascal and Ada; the typing constraints interfere (see [DCBA89], for example). In particular, the correct behavior of *move* on colored points cannot be simulated using only a "conversion" function mapping *colored_point* to *point* (*c.f.* [BL88, BTCGS89]). If we convert a *colored_point* p to a *point* p' and then send the *move* message to p', we obtain a *point* instead of a *colored_point*.

A more complex form of method specialization occurs when a method is overridden. To give an example, we need one method which depends on another. Let us assume we have another class of points, each having a method *slide* which moves the point one unit

up and to the right.

class *sl_point* **methods**
$x : int$, $y : int$,
$move : int \times int \rightarrow sl\_point$,
slide : *sl_point*

Since we have a *move* method, the natural implementation of *slide* is to send *move* with argument $\langle 1, 1 \rangle$. We now get an interesting form of method specialization if we replace *move* in some object (or subclass) which inherits *slide*. One subclass of *sl_point* might be the class *dir_point* of directed points which have x, y coordinates and a direction, say an angle *theta*.

class *dir_point* **methods**
$x : int$, $y : int$, *theta* : *real*,
$move : int \times int \rightarrow dir\_point$,
slide : *dir_point*

Let us assume that when we move a directed point, we wish to maintain its orientation toward some position on the perimeter of some bounding box, such as the boundary of the window or screen on which it is displayed. To achieve this behavior, we would redefine the *move* method to calculate a new direction whenever the x and y coordinates are altered. However, *slide* may be inherited directly from sliding points, because of the following phenomenon. With *slide* implemented by invoking *move*, the inherited *slide* method will invoke the more specialized *move* method associated with directed points. In other words, when *slide* is inherited by *dir_point*, this method is specialized in accordance with the *move* method on directed points, even though *slide* was declared as part of *sl_point*. This kind of behavior is relatively easy to implement. However, from a mathematical point of view, this form of specialization seems to be a fairly complex operation on functions.

While method specialization is very useful in a variety of programming situations, method specialization seems to complicate static analysis. In particular, let us say two expressions are *substitution equivalent* (or observationally congruent) if we may substitute one for the other any place inside any program, without changing the overall program behavior. This is an important relation in any language, since it characterizes the local program transformations or optimizations that may be applied safely in any context. In a Pascal-like language, it is relatively easy to state a simple condition guaranteeing substitution equivalence of two procedures: if both return the same results and have the same side-effects, for all possible values of the input parameters, then either may be substituted for the other in any program. However, looking only at a single method body in an object-oriented language, it is difficult to see whether a simple local transformation could change the behavior of the entire program. The novice might suspect, for example, that in-line substitution of a method body might preserve program meaning. However, if we replace the reference to *move* in the method body of *slide* by in-line code, this would change the way that *slide* works when inherited by directed points. Specifically, if *slide* does not refer to *move*, then overriding *move* has no effect on the behavior of *slide*.

3 Method specialization and natural transformation

There is a simple and intuitive connection between method specialization and natural transformation. We will illustrate the main idea using an elementary view of objects resembling [Car88]. Although the formal development of the paper does not depend on this section, the correspondence will hopefully gives some insight into the typing rules of the calculus presented in Section 4. For the remainder of this section, we will use *simple object* to mean a record of some type, and *simple method* to mean a certain kind of function on simple objects. A *simple class* is therefore a record type.

At the risk of overdoing a single example, let us consider simple classes of cartesian points. The most basic is the class *simple_point* whose elements are simple objects (records) with integer x and y components. Using the notation of [CM89], we may write this record type as follows[1].

$$simple\_point \ ::= \ \langle\!\langle x : int, y : int \rangle\!\rangle$$

It is also useful to consider a simple class of colored points

$$simple\_col\_point \ ::= \ \langle\!\langle x : int, y : int, c : color \rangle\!\rangle$$

Writing $<:$ for the subtype relation, we have $simple\_col\_point <: simple\_point$ by accepted subtyping rules explained in [Car88, CM89] and summarized in Appendices A and B.

A function which moves points as in Section 2 should change the x and y coordinates of any record with integer x and y fields, and preserve any additional components, such as $c : color$. If we define *simple_move* as a function from *simple_point* to *simple_point*, then we may apply *simple_move* to any

[1] In [CM89], a record type may be written by enclosing a list of labels (or field names) and types within double angle brackets $\langle\!\langle , \rangle\!\rangle$, as described in Appendix A.

simple object of any subtype of *simple_point* (see rule (*subsum*) in Appendix B). However, this always gives us a *simple_point*, rather than an element of a subtype such as *simple_col_point*. In order to get a map from *simple_col_point* to *simple_col_point*, and similarly for any other type of records with $x : int$, $y : int$ and additional components, *simple_move* must be a function of a more complicated type. The correct functionality in this case corresponds to a natural transformation.

Since natural transformations are maps between functors over categories, it might seem that we should now introduce a lot of categorical machinery. However, by working within a calculus that has the appropriate form of polymorphic functions, we may define functors and natural transformations using syntactic expressions of the calculus. In doing so, we consider the types of the calculus as objects (in the categorical sense of the word) of a category. There are two choices of morphisms. One might be the class of morphisms given by closed function expressions (or, equivalently, open expressions with exactly one free variable; see [MS89]). However, the more appropriate category seems to be the preorder given by the provable subtyping assertions of our calculus. We will be primarily concerned with subcategories of this category which consist of all subtypes of a given type. Since these categories are preorders, a functor is determined by a function F from subtypes of some A to types such that whenever $s <: t <: A$, we have $Fs <: Ft$. In type systems based on [CW85], the "kind" $\forall s <: A.T$ is the collection of all functions which map every subtype s of A to a type (element of the kind T of all types). It is a helpful notational convention to use a double colon "::" for kind membership and reserve the single colon for types. Using this notation, the functors we consider are given by type functions $F :: \forall s <: A.T$. Rather than explain the general idea in any more detail (*c.f.* [Rey84, RP89]), we will illustrate the approach by the example at hand. It is hoped that the main ideas will be immediately clear to those familiar with category theory, and still reasonably accessible to those without.

To consider *simple_move* as a natural transformation, we must generalize *simple_point* from a type to a functor. The codomain of the functor we want should be the collection of subtypes of *simple_point*, since *simple_move* acts as a function on each subtype of *simple_point*. Using record type expressions of [CM89], we may define the map

$$F \;::=\; \lambda R <: \langle\!\rangle\backslash x\, y.\langle\!| R \,|\, x : int, y : int \,|\!\rangle$$

from subtypes of $\langle\!\rangle\backslash x\, y$ to record types, which is explained below. It is easy to verify that F is a functor. In words, this type function maps any type R of records without x and y fields to the type $\langle\!| R \,|\, x : int, y : int \,|\!\rangle$ of all records obtained by adding integer x and y fields to some record from R. In more detail, $\langle\!\rangle$ is the type of all records, and (consequently) $\langle\!\rangle\backslash x\, y$ is the type of all records without x or y fields. The constraint that formal parameter R of F must be a subtype of $\langle\!\rangle\backslash x\, y$ first implies that the domain of F is the collection of all subtypes of $\langle\!\rangle\backslash x\, y$, and second guarantees that the type expression $\langle\!| R \,|\, x : int, y : int \,|\!\rangle$ is well-formed, since in the [CM89] calculus we may only add fields to records which are known not to already have these fields. The range of the functor F is the collection of all subtypes of *simple_point* which are obtained by adding new fields[2]. Thus F is a functor on the subcategory of our language whose objects are record types without x and y fields and whose morphisms are given by the subtyping preorder on these types.

The natural extension of *simple_move* to a map on arbitrary types of the form $\langle\!| R \,|\, x : int, y : int \,|\!\rangle$ is the polymorphic function

$$\begin{array}{l} simple\_move \;::= \\ \quad \lambda R <: \langle\!\rangle\backslash x\, y. \\ \quad\quad \lambda a : \langle\!| R \,|\, x : int, y : int \,|\!\rangle. \\ \quad\quad\quad \langle a\backslash x\, y \,|\, x{=}a.x + 1,\; y{=}a.y + 1\rangle. \end{array}$$

In words, the first parameter of this function may be any type R which is a subtype of $\langle\!\rangle\backslash x\, y$, which means that R may be any type of records without x and y fields. The second parameter is a record a of type $\langle\!| R \,|\, x : int, y : int \,|\!\rangle$. Since R may be arbitrary, we know that a has integer x and y fields, but do not know what other fields this record might have. The record expression $a\backslash x\, y$ denotes the result of removing x and y fields from a, and the form $\langle r \,|\, x = M, y = N\rangle$ is used to extend a record r by adding x and y fields with values M and N, respectively. Thus the function body $\langle a\backslash x\, y \,|\, x{=}a.x + 1,\; y{=}a.y + 1\rangle$ defines a record a' which is identical to a, but with x and y fields each incremented by 1. An elementary calculation within the record calculus shows that *simple_move* defines a natural transformation. This means that if $R <: S$,

[2] In general, a subtype of *simple_point* may have the form $\langle\!| R \,|\, x : \sigma, y : \tau \,|\!\rangle$ where $R <: \langle\!\rangle\backslash x\, y$ and both σ and τ are subtypes of int. However, since arbitrary subtypes of int may not be closed under addition, the appropriate types to use in discussing the functionality of *simple_move* have the form $\langle\!| R \,|\, x : int, y : int \,|\!\rangle$. Put another way, we wish to describe the functionality of *simple_move* using a functor whose range is the collection of all types which are closed under *simple_move*. When we apply *simple_move* to a simple object of some arbitrary subtype $\langle\!| R \,|\, x : \sigma, y : \tau \,|\!\rangle$, the best our type system can do is guarantee that the result has "more generous" type $\langle\!| R \,|\, x : int, y : int \,|\!\rangle$.

and consequently

$$\langle R \mid x : int, y : int \rangle <: \langle S \mid x : int, y : int \rangle$$

are two subtypes of *simple_point*, and if we begin with any element r of the smaller subclass $\langle R \mid x : int, y : int \rangle$, the following two computations yield the same result. The first applies *move* to r, and then "converts" r to the larger subclass $\langle S \mid x : int, y : int \rangle$ according to the subtyping assertion $R <: S$. The second computation converts r to the larger subclass before applying *move*. The fact that these two give the same result seems to capture the intuitive property that however we specialize *simple_move* on subclasses, it should respect the behavior of *simple_move* on *simple_point*'s.

It is worth mentioning that the view of methods as natural transformations not only gives us a reasonable "default" for specializing a method to subclasses with additional properties, but allows for the possibility of "redefining" methods in any way that is consistent with our interpretation of subtyping. However, within the framework of "simple objects," we have no linguistic or semantic mechanism for redefining methods. This leads us to include methods as components of objects, as we do in the next section.

4 Classes and objects

4.1 Class types

In the record calculus of [CM89], a record type determines a finite map from field names to types. Since the type of an object would naturally be a list of method names and their types, it is expedient to use record field names as method names, and define class types using record types. As a consequence of using the type expressions of [CM89], we get variables ranging over finite maps with certain method names guaranteed *not* to be in their domain. This is exactly what we need in order to type methods on objects (*c.f.* [JM88]). While it is certainly possible to define class types without mentioning records, it is convenient to use the following formation rule.

$$\frac{\Gamma, t :: T \triangleright R <: \langle\rangle}{\Gamma \triangleright \mathbf{class}\, t.\{R\} :: T}$$

In words, if R is any record type expression, possible containing a free type variable t, then $\mathbf{class}\, t.\{R\}$ is a type. The type variable t is bound in $\mathbf{class}\, t.\{R\}$ If R is an explicit record type of the form $\langle \ldots \rangle$, then it is convenient to omit the angle brackets from the corresponding **class** type expression. For example, the class of points may be written

$$point ::= \mathbf{class}\, t.\{x : int,\ y : int,\ move : int \times int \to t\}$$

In words, the type expression for *point* defines the class t with methods $x : int, y : int$ and $move : int \times int \to t$.

4.2 Operations on objects

An object is a value which accepts messages. The simplest object is the "empty" object, which accepts no messages at all. We will write $\{\}$ for the empty object, and $o \Leftarrow m$ for the result of sending message m to object o. Since the result of sending a message may be a function (from objects to objects, for example), there is no special syntax for message parameters.

In addition to sending a message, there are two basic operations on objects, adding a method and replacing a method. Suppose o is an object accepting messages $m_1, \ldots, m_k$ and that we want to extend o to an object o' accepting an additional message n. We begin by choosing a "method body" e, which must be a function; the result of sending message n to the new object o' will be the result of the application eo' of method body e to object o'. A reasonable syntax for the object obtained by extending o with method body e for n might be

$$\mathbf{extend}\ o\ \mathbf{with}\ n = e$$

Using the syntax $o \Leftarrow n$ for sending message n to o, we could then evaluate message send by a rule such as

$$\begin{aligned}(\mathbf{extend}\ o\ \mathbf{with}\ n = e) \Leftarrow n \\ = e(\mathbf{extend}\ o\ \mathbf{with}\ n = e)\end{aligned}$$

Since e is passed the entire object as a parameter, the method body may send any other message $m_1, \ldots, m_k$ to the object, or send the message n if desired. In this way, recursion and "self-reference" become inherent parts of our object model.

There is a minor technical difficulty with the simple syntax presented above. Suppose we send a message m_i to the object o' above. This is not the "most recently added" method, but a method implemented in the "old" object o. The method body for m_i is therefore designed to be applied to some object that does not have an n method, but only methods among $m_1, \ldots, m_k$. Therefore, before we apply the appropriate method body to o', we must somehow alter the method body to accept an argument with additional methods. As a "bookkeeping" mechanism for keeping the types of methods straight, we will use a syntax for objects that indicates, for each method, the list of methods that were known at the

time this method was added. Specifically, if o is an object accepting messages $m_1, \ldots, m_k$, we will write $\{o \mid n(m_1, \ldots, m_k, n)=e\}$ for the result of extending o with method body e for n. The equational rules for manipulating object expressions will allow us to "update" the types of method bodies to account for methods added later (see Table 2 and related discussion).

If we replace a method, then we write $\{o \leftarrow m_i(m_1, \ldots, m_k)=e\}$ for the object obtained from o by redefining m_i to be e. Since an object containing a method e may later be altered by adding or replacing methods, our typing rules must guarantee that e makes sense for any object obtained in this way.

4.3 Typing rules for objects

The first typing rule specifies that the empty object belongs to the class which does not promise any methods.

$$\frac{\Gamma \text{ context}}{\Gamma \rhd \{\} : \mathbf{class}\, t.\{\}}$$

In words, if Γ is a well-formed context (designating types for variables; see Appendix B), then we have the *judgement* $\Gamma \rhd \{\} : \mathbf{class}\, t.\{\}$ asserting that in context Γ, the expression $\{\}$ has type $\mathbf{class}\, t.\{\}$.

The next rule describes method addition, which is relatively complicated. There are two main features of this rule. The first is that a method must be a function applicable to the object obtained by adding this method, and that the type of the result of sending a message is the type of this function application. The second overall objective is to guarantee that the method will make sense for all "future" objects constructed from this one. In intuitive terms, we require that a method have the polymorphic type of a "natural transformation" on the "functor" which produces extensions of the present class. (Technically, it is worth noting that that the "functor" is actually a map from types to types which does not always seem to respect the subtyping preorder.) If we begin with an object o of type $\mathbf{class}\, t.\{m_1 : \sigma_1, \ldots, m_k : \sigma_k\}$, then every object obtained by adding or redefining methods will have a type of the form $\mathbf{class}\, t.\{Ft \mid m_1 : \sigma_1, \ldots, m_k : \sigma_k\}$, where F is a function from types to record types such that Ft never involves the names $m_1, \ldots, m_k$ of methods of o. Since the types of "future" objects are characterized by type functions, we want any method we add to o to define a natural transformation on a functor whose domain is a category of maps from types to types. (The morphisms of this category correspond to the point-wise subtyping preorder on $T \Rightarrow T$ described in Appendix B). In a notation following [CM89], we require that the new method body have a type of the form

$$\forall F <: (\lambda t :: T.\langle\!\rangle \backslash m_1 \ldots m_k n).\ [\mathbf{class}\, t.\{Ft \mid m_1 : \sigma_1, \ldots, m_k : \sigma_k, n : \tau\}/t](t \rightarrow \tau)$$

where the constraint $F <: (\lambda t :: T.\langle\!\rangle \backslash m_1 \ldots m_k n)$ guarantees that the function F from types to types always produces a record type without $m_1 \ldots m_k n$, and the square brackets in the subexpression $[\ldots/t](t \rightarrow \tau)$ indicate substitution. In words, the new method body must be a polymorphic function which, for any "possible future" type $\mathbf{class}\, t.\{Ft \mid m_1 : \sigma_1, \ldots, m_k : \sigma_k, n : \tau\}$, maps objects of this type to some result type possibly depending on the type of objects involved. The formal rule, which is illustrated by example in Section 4.4, appears at the top of Table 1. Reading the rule in words, we begin with an object e_1 which has methods $m_1, \ldots, m_k$ and wish to add another method n implemented using method body e_2. For this to make good sense, e_2 must be a function which, for any addition methods Ft, makes sense on an object of type $\mathbf{class}\, t.\{Ft \mid m_1 : \sigma_1, \ldots, m_k : \sigma_k, n : \tau\}$, where τ is the type of result of sending message n to the new object. The constraint at the binding occurrence of F is that for any argument t, the type Ft must be a subtype of $\langle\!\rangle \backslash m_1 \ldots m_k$. This is a formal way of saying that Ft may be any finite function from field names to types "containing" t which does not associate a type to any $m_1, \ldots, m_k, n$.

The rule for replacing one method by another is similar, but somewhat less complicated. If we begin with an object $e_1 : \mathbf{class}\, t.\{m_1 : \sigma_1, \ldots, m_k : \sigma_k\}$ and wish to replace m_i, then we need an alternate method body with the type required to produce a result of type σ_i. While it seems reasonable to allow the new method body to have a type corresponding to some subtype of σ_i, we will make the simplifying assumption that the new method returns the same type of result as the old. The formal rule which accomplishes appears in Table 1.

Since we may only add methods one-at-a-time, the reader may wonder whether it is possible to define an object with two mutually recursive methods, m and n, for example. It is generally possible to do this, but in a fuller development of the calculus it would probably be worthwhile to use more general rules allowing simultaneous addition of several methods. There is no technical problem in doing this, but the typing rules become more difficult to read.

The typing rule for message send specifies that

$$(add\ \sout{meth}) \quad \frac{\Gamma \triangleright e_1 : \mathbf{class}\, t.\{m_1 : \sigma_1, \ldots, m_k : \sigma_k\} \quad \Gamma \triangleright e_2 : \forall F <: (\lambda t :: T.\langle\!|\rangle \backslash m_1 \ldots m_k n).[\mathbf{class}\, t.\{Ft \mid m_1 : \sigma_1, \ldots, m_k : \sigma_k, n : \tau\}/t](t \to \tau)}{\Gamma \triangleright \{e_1 \mid n(m_1, \ldots m_k, n) = e_2\} : \mathbf{class}\, t.\{m_1 : \sigma_1, \ldots, m_k : \sigma_k, n : \tau\}}$$

$$(ovw\ \sout{meth}) \quad \frac{\Gamma \triangleright e_1 : \mathbf{class}\, t.\{m_1 : \sigma_1, \ldots, m_k : \sigma_k\} \quad \Gamma \triangleright e_2 : \forall F <: (\lambda t :: T.\langle\!|\rangle \backslash m_1 \ldots m_k).[\mathbf{class}\, t.\{Ft \mid m_1 : \sigma_1, \ldots, m_k : \sigma_k\}/t](t \to \sigma_i)}{\Gamma \triangleright \{e_1 \leftarrow m_i(m_1, \ldots, m_k) = e_2\} : \mathbf{class}\, t.\{m_1 : \sigma_1, \ldots, m_k : \sigma_k\}}$$

$$(\mathbf{class}\ E) \quad \frac{\Gamma \triangleright e : \mathbf{class}\, t.\{m_1 : \sigma_1, \ldots, m_k : \sigma_k\}}{\Gamma \triangleright e \Leftarrow m_i : [\mathbf{class}\, t.\{m_1 : \sigma_1, \ldots, m_k : \sigma_k\}/t]\sigma_i}$$

$$\{o \mid m(m_1, \ldots, m_k) = e\} \Leftarrow m \quad = \quad e\ (\lambda t :: T.\langle\!|\rangle)\ \{o \mid m(m_1, \ldots, m_k) = e\} XS$$

Table 1: Typing and evaluation rules for objects.

the result of sending the message has whatever type is specified. The formal rule (**class** E) appears in Table 1. To illustrate (**class** E) by example, a object representing a number, with its own addition method (as described in [GR83], for example) might be defined by an expression with type $e : \mathbf{class}\, t.\{val : num, plus : t \to t\}$. Sending the addition message to this object produces a function from this class to itself

$$e \Leftarrow plus : \mathbf{class}\, t.\{val : num, plus : t \to t\} \to \mathbf{class}\, t.\{val : num, plus : t \to t\}$$

Therefore, $e \Leftarrow plus\ e$ produces another object of the same class.

The evaluation rules for objects compute the result of message send by applying the appropriate method to the object itself. The equational axiom at the bottom of Table 1 is based on this idea, with type application to the constant "empty record type" function used to make the application type correct. The corresponding axiom for a redefined method is similar.

There are several equational axioms for manipulating object expressions, most of them following the pattern of record axioms explained in [CM89]. One nontrivial axiom allows us to permute the order of methods. In general, an object which has been extended twice will have the form

$$\{\{o \mid n(m_1, \ldots, m_k, n) = e\} \mid n'(m_1, \ldots, m_k, n, n') = e'\}$$

Note that the method body e does not assume the object has a method named n', since the second method was added later. However, e' assumes a method named n. The equational axiom for exchanging the order of methods is given in Table 2, where the change in method body accounts for the presence of n'. Intuitively, this type manipulation is related to the fact that a natural transformation must be applied to the type of an argument before it is applied to the argument itself. However, in our calculus, we do not have a basic operation that returns the type of an expression. Therefore, we must "precompute" the type of a method argument incrementally as we build the object itself.

4.4 An example

As an example, we will show how to define an object of class *point*. Recall that a point has x, y and *move* methods. We will define a point whose x and y methods are constant functions, returning integer coordinates x_0 and y_0 The *move* method will return a function which, given a pair of integers, returns a point with x and y coordinates altered accordingly. We begin by adding polymorphic constant functions (as methods) to the empty object $\{\} : \mathbf{class}\, t.\{\}$. Since we will also use constant functions in the definition of *move*, it is helpful to introduce the following general form for methods returning constant methods,

$$c\_meth_{\bar{m},\bar{\sigma}}[e] \quad ::= \quad \lambda F <: (\lambda t :: T.\langle\!|\rangle \backslash \bar{m}).\ \lambda x : (\mathbf{class}\, t.\{Ft \mid \bar{m} : \bar{\sigma}\}).\ e$$

for any sequence $\bar{m}$ of method names, sequence $\bar{\sigma}$ of corresponding types, and expression e not containing F or x free. (If $\bar{m} = m_1 \ldots m_k$ and $\bar{\sigma} = \sigma_1 \ldots \sigma_k$, we write $\bar{m} : \bar{\sigma}$ for $m_1 : \sigma_1 \ldots, m_k : \sigma_k$.) For any integer

$$\frac{\Gamma \triangleright \{\{o \mid n(\bar{m},n)=e\} \mid n'(\bar{m},n,n')=e'\} : \mathbf{class}\, t.\{Ft \mid n:\tau, n':\tau'\}}{\Gamma \triangleright \{\{o \mid n(\bar{m},n)=e\} \mid n'(\bar{m},n,n')=e'\} \;=\; \begin{array}{l}\{\{o \mid n'(\bar{m},n,n')=e'\} \mid n(\bar{m},n,n')= \\ \quad \lambda G <: (\lambda t :: T.\langle\!\langle\rangle\!\rangle \backslash \bar{m} n n').e\,(\lambda t :: T.\langle\!\langle Gt \mid n':\tau'\rangle\!\rangle)\}\end{array}} \; n' \text{ not among } \bar{m}, n$$

Table 2: Equational rule for permuting methods.

expression e without F or x free, the constant function $c\_meth_{\bar{m},\bar{\sigma}}[e]$ has the following "bounded polymorphic type."

$$\forall F <: (\lambda t :: T.\langle\!\langle\rangle\!\rangle \backslash \bar{m}).\; (\mathbf{class}\, t.\{Ft \mid \bar{m}:\bar{\sigma}\}) \to int$$

Therefore, by the object extension rule, the object

$$\{\{\} \mid x(x) = c\_meth_{x,int}[x_0]\}$$

with method x returning the integer coordinate x_0 has type $\mathbf{class}\, t.\{x : int\}$. This object may be extended with a constant y method returning integer coordinate y_0.

$$\begin{array}{lllll} p_{xy} ::= & \{\{\{\}| & x(x) & = & c\_meth_{x,int}[x_0] \\ & \} \mid & y(x,y) & = & c\_meth_{(xy),(int\ int)}[y_0] \\ & \} & & & \end{array}$$

This gives us an object with two integer methods.

It it useful to make several observations about the object $p_{x,y}$. First, note that the first method added, x, "expects" to be passed an object with only one method, while the second method expects both x and y. This is indicated by the lists of method names in the object expression. If we send the message y to $p_{x,y}$, the the result may be computed directly using the equational rule at the bottom of Table 1. However, if we send the x message, then we must first permute the order of methods using the equational rule on Table 2. This rule changes the type of the method body for x so that the function may be applied to any object with at least the two methods x and y. We now continue the example by adding a *move* method to $p_{x,y}$.

The move method for $p_{x,y}$ will be a polymorphic function of type

$$\begin{array}{l}\forall F <: (\lambda t :: T.\langle\!\langle\rangle\!\rangle \backslash x\, y\, move). \\ \mathbf{class}\, t.\{Ft \mid x : int,\ y : int,\ move : int \times int \to t\} \\ \quad \to int \times int \to \\ \quad \mathbf{class}\, t.\{Ft \mid x : int,\ y : int,\ move : int \times int \to t\}\end{array}$$

since *move* must map any object with x, y, *move* and additional methods to another object of the same type. An appropriate method body for *move* is given at the top of Table 3. This function takes any object with x, y and *move* methods and replaces the x and y methods by constant functions returning new coordinates. Note that the new coordinates are calculated by sending x and y messages to the object and incrementing the results. We obtain a point object by adding this method to the object p_{xy}

$$pt ::= \{p_{xy} \mid move(x, y, move) = move\_meth\}$$

To complete the example, we will compute the value of $pt \Leftarrow move$ in Table 3. Thus $pt \Leftarrow move$ is a function which, given a displacement $d : int \times int$, returns an object which is identical to pt, but with x and y coordinates incremented by the first and second components of d.

One very important fact about this calculation is that if we had a more "specialized" kind of point pt', with any number of additional methods, the same calculation would give us a new point identical to pt', but with x and y methods replaced by constant functions returning new coordinates. It is exactly this uniform behavior of methods, guaranteed by the typing rules, that allows us to inherit *move* from pt and use it on extensions of pt.

5 A translation of objects into records

The object and class expressions introduced in the previous section may be interpreted in the calculus with records and recursive type definitions summarized in Appendix B, in the sense that under an appropriate syntactic translation, all of the typing and equational rules for objects are derived rules of the record calculus. Although we have not studied the semantics of the target record calculus when recursive type declarations are allowed [CM89], the calculus is close enough to other systems so that semantic soundness seems very likely (see [BTCGS89]). However, as outlined in Section 6, this translation does not respect the natural subtyping relation on classes. Thus a semantic account of class subtyping remains an open problem.

The translation into records is syntax-directed, proceeding by induction on the formation of an ex-

$$
\begin{aligned}
move\_meth \;::=\; & \lambda F <: (\lambda t :: T.\langle\!\langle\rangle\!\rangle \backslash x\, y\, move).\\
& \quad \lambda o : \mathbf{class}\, t.\{Ft \mid x : int,\ y : int,\ move : int \times int \to t\}.\\
& \quad\quad \lambda d : int \times int.\\
& \quad\quad\quad \{\{o \leftarrow x(x, y, move) = c\_meth[(o \Leftarrow x) + fst\ d]\}\\
& \quad\quad\quad\quad \leftarrow y(x, y, move) = c\_meth[(o \Leftarrow y) + snd\ d]\}
\end{aligned}
$$

$$
\begin{aligned}
pt \Leftarrow move \;&=\; \{p_{xy} \mid move(x, y, move) = move\_meth\} \Leftarrow move\\
&=\; (move\_meth)\ (\lambda t :: T.\langle\!\langle\rangle\!\rangle)\ pt\\
&=\; \lambda d : int \times int.\\
&\quad \{\{pt \leftarrow x(x, y, move) = c\_meth[(pt \Leftarrow x) + fst\ d]\}\\
&\quad\quad \leftarrow y(x, y, move) = c\_meth[(pt \Leftarrow y) + snd\ d]\}
\end{aligned}
$$

Table 3: *Move* method body and example calculation.

pression. The translation of an object is essentially a record containing the methods of the object. More precisely, we translate the empty object to the empty record, and $\{e_1 \mid m(m_1, \ldots, m_k, m) = e_2 : F\}$ to the record expression $\langle Trans_{m:F}(e_1) \mid m = e_2\rangle$, where $Trans_{m:F}$ is a translation which makes sure that the type of each method in e is "adjusted" to take the presence of the additional method into account. A class type expression $\mathbf{class}\, t.\{m_1 : \sigma_1, \ldots, m_k : \sigma_k\}$ may be interpreted as the recursive type expression

$$\mu t.\langle\!\langle m_1 : t \to \sigma_1, \ldots, m_k : t \to \sigma_k\rangle\!\rangle$$

Note however that we may select components directly from a record of this type, whereas message send only gives us "indirect" access to the methods of an object. This is the main reason why the translation of objects into records does not respect the natural subtype ordering on class types. The translation becomes slightly more complicated in the presence of record type variables (which must also be translated), but is essentially routine given the development of [CM89]. Details are omitted from this conference paper.

6 Subtyping

The subtype relation on classes is relatively subtle. However, since ordinary bounded quantification is not the only way to define polymorphic functions over all classes of a certain form [CCH+89], our ability to write useful programs is not as dependent on the subtyping relation as might at first appear. By analogy with record types, one might think that if one class type is obtained from another by adding methods, this should be a subclass. However, consider the classes

$$
\begin{aligned}
A \;::=\; & \mathbf{class}\, t.\{x : int,\ y : int,\ plus : t \to t\}\\
B \;::=\; & \mathbf{class}\, t.\{x : int,\ plus : t \to t\}
\end{aligned}
$$

Should we consider $A <: B$? The bottom line is that we may only adopt $A <: B$ if, in any context, any expression of type B could safely be replaced by an expression of type A. But consider the expression $o \Leftarrow plus\ o$, where $o : B$. If the *plus* method for some $o' : A$ is implemented by using both x and y methods, then it certainly does not make sense to replace the first occurrence of o by $o' : A$. Thus $A <: B$ is unsound. On the other hand, it certainly seems reasonable that any class of the form

$$\mathbf{class}\, t.\{\ldots, print : string, \ldots\}$$

should be a subtype of $printable ::= \mathbf{class}\, t.\{print : string\}$. This would be useful, for example, in writing a print queue which collects *printable* objects and prints each one in turn (each using its own *print* method).

This example raises an important issue regarding the difference between the natural subtyping relation on classes and subtyping on recursive record types. For example, a class with print method might be interpreted, under the translation mentioned above, as the recursive record type

$$R ::= \mu t.\langle x : t \to \sigma,\ print : t \to string\rangle$$

while the class *printable* would be interpreted as

$$printable\_rcd ::= \mu t.\langle\!\langle print : t \to string\rangle\!\rangle$$

Under the accepted notion of subtyping for recursive record types (see Appendix B), we do *not* have

$R <: printable\_rcd$. (This is similar to the $A <: B$ example above.) This illustrates that in semantic models of the calculus of objects and classes, it is important to take seriously the fact that methods may not be selected from objects, only applied to the object itself. For without this consideration, the expected subtyping relation on classes cannot be semantically justified.

7 Conclusion

We have developed a preliminary function calculus with objects and classes, and justified the typing rules by translation into a more commonly studied calculus which is believed sound. Although we may explain method specialization using the more familiar framework of recursively-defined record types, semantic justification of reasonable subtyping rules seems a difficult open problem.

The calculus of objects and classes is presented using record type expressions from the record calculus of [CM89]. This is convenient when it comes to translating objects into records, but from a programming point of view it seems unnecessarily complex. In future work, it might be useful to simplify the language to those type expressions that are absolutely necessary to program realistic object-oriented examples, and eliminate records in favor of objects.

The main long-term objectives of this work are to provide a basis for reasoning about object-oriented programming languages, and to design flexible polymorphic type systems. In future work, it seems worthwhile to consider languages with different sets of basic operations, in hopes that we could more easily guarantee that the meanings of expressions have the types outline here, without requiring as much type information in the syntax itself. Although type inference algorithms might help, it seems more useful to take up the connection with natural transformations in earnest and define a language in which application of natural transformation is a basic operation. Or, since many of the type functions do not induce functors (*i.e.*, do not respect subtyping), a treatment based on presheaf categories seems more promising. This might alleviate much of the complication and could lead to a more elegant language. Another research direction is to try to adapt the typing concepts presented here to a prototyping-based language such as Self. It is hoped that some of the optimizations achieved through dynamic typing in [CU89], for example, could be guaranteed by a static typing discipline along the lines suggested here.

Acknowledgements: I am grateful to Luca Cardelli of DEC Systems Research Center and the members of the ABEL group at HP Laboratories (Peter Canning, William Cook, Walt Hill and Walter Olthoff) for many discussions. Thanks also to Eugenio Moggi, Gordon Plotkin and Andre Scedrov for their comments and insight.

References

[AR88] N. Adams and J. Rees. Object-oriented programming in Scheme. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, pages 277–288, July 1988.

[BI82] A.H. Borning and D.H. Ingalls. A type declaration and inference system for Smalltalk. In *ACM Symp. Principles of Programming Languages*, pages 133–141, 1982.

[BL88] K. Bruce and G. Longo. A modest model of records, inheritance and bounded quantification. In *Third IEEE Symp. Logic in Computer Science*, pages 38–51, 1988.

[BMM89] K. B. Bruce, A. R. Meyer, and J. C. Mitchell. The semantics of second-order lambda calculus. *Information and Computation*, 1989. (to appear).

[Bor86] A.H. Borning. Classes versus prototypes in object-oriented languages. In *ACM/IEE Fall Joint Computer Conf.*, pages 36–40, 1986.

[BTCGS89] V. Breazu-Tannen, T. Coquand, C.A. Gunter, and A. Scedrov. Inheritance and explicit coercion. In *Fourth IEEE Symp. Logic in Computer Science*, page (to appear), 1989.

[Car86] L. Cardelli. Amber. In *Combinators and Func. Programming*, pages 21–47. Springer-Verlag LNCS 242, 1986.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Special issue devoted to *Symp. on Semantics of Data Types*, Sophia-Antipolis (France), 1984.

[CCH+89] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded

quantification for object-oriented programming. In *Functional Prog. and Computer Architecture*, 1989. To appear.

[CDG+88] 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.

[CDJ+89] 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.

[CM88] L. Cardelli and J.C. Mitchell. Semantic methods for object-oriented languages. Unpublished OOPSLA tutorial, 1988.

[CM89] L. Cardelli and J.C. Mitchell. Operations on records. In *Math. Foundations of Prog. Lang. Semantics*, 1989. To appear.

[Coo89a] W.R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[Coo89b] W.R. Cook. A proposal for making Eiffel type-safe. In *European Conf. on Object-Oriented Programming*, pages 57–72, 1989.

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

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[DCBA89] A. DiMaio, C. Cardingno, R. Bayan, and C. Atkinson. Dragoon: an Ada-based object-oriented language. In *Proc. Ada-Europe Conference*, 1989. To appear.

[DT88] S. Danforth and C. Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–72, 1988.

[Gir71] J.-Y. Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J.E. Fenstad, editor, *2nd Scandinavian Logic Symposium*, pages 63–92. North-Holland, 1971.

[Gir72] J.-Y. Girard. Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur. These D'Etat, Universite Paris VII, 1972.

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

[JM88] L. Jategaonkar and J.C. Mitchell. ML with extended pattern matching and subtypes. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, pages 198–212, July 1988.

[Kam88] S. Kamin. Inheritance in smalltalk-80: a denotational definition. In *ACM Symp. Principles of Programming Languages*, pages 80–87, 1988.

[Lie86] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 214–223, October 1986.

[LTP86] W.R. LaLonde, D.A. Thomas, and J.R. Pugh. An exemplar based Smalltalk. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 322–330, October 1986.

[Mey88] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[Mit84] J.C. Mitchell. Coercion and type inference (summary). In *Proc. 11-th ACM Symp. on Principles of Programming Languages*, pages 175–185, January 1984.

[MPS86] D. MacQueen, G Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, 1986.

[MS89] J.C. Mitchell and P.J. Scott. Typed lambda calculus and cartesian closed categories. In *Proc. Conf. Computer Science and Logic June 14-20, 1987, Univ. Colorado Boulder*, volume 92 of *Contemporary Mathematics*, pages 301–316. Amer. Math. Society, 1989.

[RA82] 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.

[Red88] U.S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, pages 289–297, July 1988.

[Rém89] D. Rémy. Typechecking records and variants in a natural extension of ML. In *16-th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[Rey84] J.C. Reynolds. Polymorphism is not set-theoretic. In *Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France), Springer LNCS 173*, pages 145–156. Springer-Verlag, 1984.

[RP89] J.C. Reynolds and G.D. Plotkin. On functors expressible in the polymorphic lambda calculus. *Information and Computation*, page to appear, 1989.

[Sta88] R. Stansifer. Type inference with subtypes. In *Proc. 15-th ACM Symp. on Principles of Programming Languages*, pages 88–97, January 1988.

[Suz81] N. Suzuki. Inferring types in Smalltalk. In *ACM Symp. Principles of Programming Languages*, pages 187–199, 1981.

[US87] 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.

[Wan87] M. Wand. Complete type inference for simple objects. In *Proc. 2-nd IEEE Symp. on Logic in Computer Science*, pages 37–44, 1987. Corrigendum in *Proc. 3-rd IEEE Symp. on Logic in Computer Science*, page 132, 1988.

[Wan89] M. Wand. Type inference for record concatenation and simple objects. In *Proc. 4-nd IEEE Symp. on Logic in Computer Science*, pages 92–97, 1989.

[Yel89] P.M. Yelland. First steps towards fully-abstract semantics for object-oriented languages. In *European Conf. on Object-Oriented Programming*, pages 347–367, 1989.

A Summary of Cardelli-Mitchell record operations

A.1 Introduction

This appendix contains an intuitive summary of the record operations presented in [CM89]. The general idea of [CM89] is to extend a polymorphic type system with a notion of subtyping at all types. Record types are then introduced as specialized type constructions with some specialized subtyping rules.

A.2 Record values

A record value is essentially a finite map from labels to values, where the values may belong to different types. Syntactically, a record value is a collection of fields, where each field is a labeled value. To capture the notion of a map, the labels in a given record must be distinct. Hence the labels can be used to identify the fields, and the fields should be regarded as unordered. This is the notation we use:

$\langle\rangle$	the empty record.
$\langle x = 3, y = true\rangle$	a record with two fields, labeled x and y, and equivalent to $\langle y = true, x = 3\rangle$.

There are three basic operations on record values, *extension, restriction,* and *extraction*. These have the following basic properties.

Extension $\langle r|x = a\rangle$ adds a field of label x and value a to a record r, provided a field of label x is not already present. This restriction will be enforced statically by the type system. The additional brackets placed around the operator help to make the examples more readable; we also write $\langle r|x = a|y = b\rangle$ for $\langle\langle r|x = a\rangle|y = b\rangle$.

Restriction $r\backslash x$ removes the field of label x, if any, from the record r. We write $r\backslash xy$ for $(r\backslash x)\backslash y$.

Extraction $r.x$ extracts the value corresponding to the label x from the record r, provided a field having

$\langle\langle x = 3\rangle \mid y = true\rangle$	=	$\langle x = 3, y = true\rangle$	extension
$\langle x = 3, y = true\rangle \backslash y$	=	$\langle x = 3\rangle$	restriction (canceling y)
$\langle x = 3, y = true\rangle \backslash z$	=	$\langle x = 3, y = true\rangle$	restriction (no effect)
$\langle x = 3, y = true\rangle.x$	=	3	extraction
$\langle\langle x = 3\rangle \mid x = 4\rangle$	invalid extension		
$\langle x = 3\rangle.y$	invalid extraction		

Table 4: Example record expressions

that label is present. This restriction will be enforced statically by the type system.

We have chosen these three operations because they seem to be fundamental constituents of more complex operations. Some examples are given in Table 4.

Some additional operators may be defined in terms of the ones above.

Renaming $r[x \leftarrow y] \stackrel{\text{def}}{=} \langle r \backslash x | y = r.x\rangle$ changes the name of a record field.

Overriding $\langle r \leftarrow x = a\rangle \stackrel{\text{def}}{=} \langle r \backslash x | x = a\rangle$. If x is present in r, replace its value with one of a possibly unrelated type, otherwise extend r with $x = a$ (compare with [Wan89]). Given adequate type restrictions, this can be seen as an updating operator, or a method overriding operator. We write $\langle r \leftarrow x = a, y = b\rangle$ for $\langle\langle r \leftarrow x = a\rangle \leftarrow y = b\rangle$.

It is clear that any record may be constructed from the empty record using extension operations. In fact, it is convenient to regard the syntax for a record of many fields as an abbreviation for iterated extensions of the empty record, *e.g.*,

$$\langle x = 3, y = true\rangle \stackrel{\text{def}}{=} \langle\langle\langle\rangle | x = 3\rangle | y = true\rangle.$$

This approach to record values allows us to express the fundamental properties of records using combinations of simple operators of fixed arity, as opposed to n-ary operators. Hence we never have to use schemas with ellipses, such as $\langle x_1 = a_1, ..., x_n = a_n\rangle$, in our formal treatment.

Since $r \backslash x = r$ whenever r lacks a field of label x, we may write $\langle x = 3, y = true\rangle$ using any of the following expressions:

$$\begin{aligned} \langle\langle\rangle | x = 3 | y = true\rangle &= \langle\langle\langle\rangle \backslash x | x = 3\rangle \backslash y | y = true\rangle \\ &= \langle\langle\rangle, x = 3, y = true\rangle \end{aligned}$$

The latter forms match a similar definition for record types, given in the next section.

A.3 Record types

In describing operations on record values, we made positive assumptions of the form "a field of label x must occur in record r" and negative assumptions of the form "a field of label x must not occur in record r". These constraints will be verified statically by the type system. To accomplish this, record types must convey both positive and negative information. Positive information describes the fields that members of a record type must have, while negative information describes the fields the members of that type must not have. Within these constraints, the members of a record type may or may not have additional fields or lack additional fields. It is worth emphasizing that both positive and negative constraints restrict the elements of a type, hence increasing either kind of constraint will lead to smaller sets of values. The smallest amount of information is expressed by the "empty" record type $\langle\!\langle\rangle\!\rangle$. The "empty" record type is empty only in that it places no constraints on its members – every record has type $\langle\!\langle\rangle\!\rangle$, since all records have at least no fields and lack at least no fields. Some examples are given in Table 5.

As with record values, we have three basic operations on record types.

Extension $\langle\!\langle R | x : A\rangle\!\rangle$ This type denotes the collection obtained from R by adding x fields with values in A in all possible ways (provided that none of the elements of R have x fields). More precisely, this is the collection of those records $\langle r | x = a\rangle$ such that r is in R and a is in A, provided that a positive type field x is not already present in R (this will be enforced statically). We sometime write $\langle\!\langle R | x : A | y : B\rangle\!\rangle$ for $\langle\!\langle\langle\!\langle R | x : A\rangle\!\rangle | y : B\rangle\!\rangle$.

Restriction $R \backslash x$ This type denotes the collection obtained from R by removing the field x (if any) from all its elements. More precisely, this is the collection of those records $r \backslash x$ such that r is in R. We write $R \backslash xy$ for $(R \backslash x) \backslash y$.

Extraction $R.x$ This is the type associated to label

$\langle\!\rangle$	the type of all records. Contains, *e.g.*, $\langle\rangle$ and $\langle x = 3\rangle$.
$\langle\!\rangle\backslash x$	the type of all records which lack a field labeled$x$. *E.g.*, $\langle\rangle$, $\langle y = true\rangle$, ***but not*** $\langle x = 3\rangle$.
$\langle x : Int, y : Bool\rangle$	the type of all records which have at least fields labeled x and y, with values of types *Int* and *Bool*. $E.g.,\langle x = 3, y = true\rangle, \langle x = 3, y = true, z = str\rangle$ but not $\langle x = 3, y = 4\rangle, \langle x = 3\rangle$.
$\langle x : Int\rangle\backslash y$	the type of all records which have at least a field labeled x of type *Int*, and no field with label y. $E.g.,\langle x = 3, z = str\rangle$, but not $\langle x = 3, y = true\rangle$.

Table 5: Example record type expressions.

$\langle\langle x : Int\rangle\backslash y \vert y : Bool\rangle$	=	$\langle x : Int, y : Bool\rangle$	extension
$\langle x : Int, y : Bool\rangle\backslash y$	=	$\langle x : Int\rangle\backslash y$	restriction (canceling y)
$\langle x : Int, y : Bool\rangle\backslash z$	=	$\langle x : Int, y : Bool\rangle\backslash z$	restriction (no effect)
$\langle x : Int, y : Bool\rangle.x$	=	*Int*	extraction
$\langle\langle\rangle \vert x : Bool\rangle$	invalid extension		
$\langle x : Int\rangle.y$	invalid extraction		

Table 6: Record type extension examples

x in R, provided R has such a positive field. This provision will be enforced statically. Again, several derived operators can be defined from these.

Renaming $R[x, y] \stackrel{\text{def}}{=} \langle R\backslash x \vert y = R.x\rangle$ changes the name of a record type field.

Overriding $\langle R \leftarrow x : A\rangle \stackrel{\text{def}}{=} \langle R\backslash x \vert x : A\rangle$ if a type field x is present in R, replaces it with a field x of type A, otherwise extends R. Given adequate type restrictions, this can be used to override a method type in a class signature (i.e. record type) with a more specialized one, to produce a subclass signature.

One crucial formal difference between these operators on types and the similar ones on values is that $\langle\!\rangle\backslash y \neq \langle\!\rangle$, since records belonging to the "empty" type may have y fields, whereas $\langle\rangle\backslash y = \langle\rangle$. In forming record types, one must always make a field restriction before a type extension, as illustrated by example in Table 6.

It helps to read the examples in terms of the collections they represent. For example, the first example for restriction says that if we take the collection of records that have x and y (and possibly more) fields, and remove the y field from all the elements in the collection, then we obtain the collection of records that have x (and possibly more) but no y. In particular, we do not obtain the collection of records that have x and possibly more fields, because those would include y.

The way positive and negative information is formally manipulated is actually easier to understand if we regard record types as abbreviations, as we did for record values:

$$\langle x : Int, y : Bool\rangle \stackrel{\text{def}}{=} \langle\langle\langle\rangle\backslash x \vert x : Int\rangle\backslash y \vert y : Bool\rangle$$

Then, when considering $\langle y{:}Bool\rangle\backslash y$, we actually have $\langle\langle\rangle\backslash y \vert y{:}Bool\rangle\backslash y$. If we allow the outside positive and negative y labels to cancel, we are still left with $\langle\!\rangle\backslash y$. The inner y restriction reminds us that y fields have been eliminated from records of this type.

A.4 Subtyping

The subtyping rules for record types are essentially that every record type is a subtype of $\langle\!\rangle$, and the subtyping relation respects the type operations of extension, restriction and extraction. Writing $A <: B$ for A *is a subtype of* B, we have the following examples.

$\langle x : Int, y : Bool\rangle <: \langle\!\rangle$	
$\langle R \vert x : A\rangle <: \langle S \vert x : A\rangle$	if $R <: S <: \langle\!\rangle\backslash x$
$R\backslash x <: S\backslash x$	if $R <: S <: \langle\!\rangle$
$R.x <: S.x$	if $R <: S <: \langle x : A\rangle$

In general, a record type R will be a subtype of another record type S if every positive constraint (la-

beled field) associated with R is also a positive constraint imposed by S, and similarly for negative constraints (fields required to be absent). There are some subtleties. For example, $\langle R\backslash x | x : Int \rangle$ is not necessarily a subtype of R, and *never* a subtype of $R\backslash x$, even though this might seem consistent with the point of view expressed in [Car88], for example.

B Summary of full typed calculus

B.1 Overview

The basic calculus we use is a higher-order typed lambda calculus, in the general style of Girard's F_ω [Gir71, Gir72] and many subsequent systems. In addition to function types and polymorphism, we will use recursive type definitions, subtyping and a relatively elaborate form of record types and operations. All of this is quite "standard," in the sense of being syntactically familiar to type theorists, with the exception of the record calculus and subtyping. The main subtyping notions are primarily due to Cardelli [Car88, CW85], with some remnants of [Mit84]. The record calculus with subtyping is explored in some depth in [CM89], although certain semantic and pragmatic questions remain. While many of the syntactic aspects of this calculus will be familiar, the reader should not take this as an indication that the semantics are well-understood. In particular, the combination of record operations, subtyping and recursive types poses a number of mathematical challenges However, it seems that the bulk of the the problems here have been identified and discussed in the literature (see, for example, [BTCGS89, BL88, CM89]). Due to space limitations, we will only summarize the basic parts of the calculus.

B.2 Kinds

The types and type-producing functions of the calculus are characterized by *kinds*, following [Gir72, MPS86, BMM89]; kinds are called *orders* in [Gir72] and some subsequent work. The kinds consist of the base kind T, the kind of all types, and kinds of functions over T. Since we wish to consider type constructors defined only on subtypes of some type A, for example, we will allow "dependent" kinds of the form $\forall\, s <: A . \kappa_1$, where $A :: \kappa_2$. While we could also introduce product kinds $\kappa_1 \times \kappa_2$, we have no immediate need for them in this paper. We write "κ kind" if κ is a well-formed kind expression.

B.3 Contexts

Free variables are given types or kinds using *contexts*, which are ordered lists of assumptions about variables. We write "Γ context" if Γ is a well formed context. The basic rules for context are standard, with $\emptyset$ context. We write $v :: \kappa$ to indicate that v has kind κ, and $x : \tau$ to indicate that x has type τ. Kinded variables are added to contexts using the rule

$$\frac{\Gamma \text{ context}, \quad \kappa \text{ kind}}{\Gamma, v :: \kappa \text{ context}} \quad v \notin Dom(\Gamma).$$

We omit the analogous rule for typed variables.

Since subtyping is a basic notion of the calculus, we may assert in a context that a fresh type variable denotes a subtype of a given type. It is also useful to have "subtyping" at higher kinds, with the intuitive meaning that $<:$ on kind $\kappa_1 \Rightarrow \kappa_2$ is the pointwise ordering

$$F <:_{\kappa_1 \Rightarrow \kappa_2} G \quad \text{iff} \quad \forall v : \kappa_1 . f v <:_{\kappa_2} g v.$$

We generally omit subscripts from the subtype relation.

Subtyping assumptions are added to contexts according to the rule

$$\frac{\Gamma \triangleright A :: \kappa}{\Gamma, U <: A \text{ context}}.$$

B.4 Subtyping

The main typing rule associated with subtyping is called *subsumption.*

$$(subsum) \quad \frac{\Gamma \triangleright e : \sigma, \quad \Gamma \triangleright \sigma <: \tau}{\Gamma \triangleright e : \tau}$$

This rule lets us apply a function $f : \sigma \to \tau$ to an argument $x : \sigma'$, whenever $\sigma' <: \sigma$, for example.

B.5 Function Types

We have ordinary function types, formed according to the typing rule

$$(\to) \quad \frac{\Gamma \triangleright A :: T, \quad \Gamma \triangleright B :: T}{\Gamma \triangleright A \to B :: T}$$

and polymorphism over all kinds, as in F_ω.

$$(\forall) \quad \frac{\Gamma, v :: \kappa \triangleright A :: T}{\Gamma \triangleright \forall v :: \kappa . B :: T}$$

In addition, as in [CW85], we have bounded polymorphism.

$$(\forall\ bdd) \quad \frac{\Gamma, u <: A \triangleright B :: T}{\Gamma \triangleright \forall u <: A . B :: T}$$

Introduction and elimination rules defining terms of these types are standard and omitted.

We do not seem to need the so-called F-bounded polymorphism of [CCH+89], since most useful cases seem to be handled by ordinary bounded quantification over kind $T \Rightarrow T$. (We have not done a thorough study of this point, and it may eventually be necessary to introduce F-bounded polymorphism.)

B.6 Record Types

The "empty" or "universal" record type is written $\langle\!\rangle$, following [CM89].

$$\frac{\Gamma \text{ context}}{\Gamma \rhd \langle\!\rangle :: T}$$

This type contains all records.

Additional types of records are formed by adding constraints of the form $x : \tau$, which assert that all elements of this type must have an x component of type τ, and constraints of the form $\backslash x$, which assert that no elements of this type may have an x component.

$$\frac{\Gamma \rhd R <: \langle\!\rangle\backslash x, \quad \Gamma \rhd A :: T}{\Gamma \rhd \langle R|x : A\rangle :: T}$$

$$\frac{\Gamma \rhd R <: \langle\!\rangle}{\Gamma \rhd R\backslash x : T}$$

In [CM89], if R is a type of records, with every record guaranteed to have an x component, then $R.x$ is the type of all x components of records from R. However, we do not seem to need type expressions of the form $R.x$ in this paper.

Using equational rules, which we omit, every record type of the pure calculus may be simplified to the form

$$\langle\!\langle R\backslash x_1\backslash x_2 \ldots \mid y_1 : A_1 \mid \ldots \rangle\!\rangle$$

where R is either the empty record type $\langle\!\rangle$, a record type variable, or an application beginning with a variable of some functional kind. It is convenient to write $\langle x_1 : A_1, \ldots, x_k : A_k\rangle$ for $\langle\, \langle\!\rangle\backslash x_1\backslash x_2 \ldots \mid x_1 : A_1 \mid \ldots \mid x_k : A_k \rangle\!\rangle$.

B.7 Record Subtyping

Every record type is a subtype of the type $\langle\!\rangle$ containing all records. Moreover, subtyping respects record type extension and restriction .

$$\frac{\Gamma \rhd R <: S <: \langle\!\rangle\backslash x, \quad \Gamma \rhd A <: B}{\Gamma \rhd \langle R|x : A\rangle\!\rangle <: \langle S|x : B\rangle\!\rangle}$$

$$\frac{\Gamma \rhd R <: S <: \langle\!\rangle}{\Gamma \rhd R\backslash x <: S\backslash x}$$

A derived rule is is that from $\Gamma \rhd \sigma_1 <: \tau_1, \ldots, \Gamma \rhd \sigma_k <: \tau_k$ we may derive that $\langle\!\langle m_1 : \sigma_1, \ldots, m_k : \sigma_k, n_1 : \rho_1, \ldots, n_\ell : \rho_\ell\rangle\!\rangle$ is a subtype of $\langle\!\langle m_1 : \tau_1, \ldots, m_k : \tau_k\rangle\!\rangle$, which may be familiar from [Car88].

B.8 Recursive Types

Recursive types will be used to describe records which contain methods applicable to the records themselves. Rather than write recursive types in the common programming language form

$$tree = leaf + \langle\!\langle l : tree, r : tree\rangle\!\rangle$$

we will use the more concise syntax

$$\mu t.leaf + \langle\!\langle l : t, r : t\rangle\!\rangle.$$

The formation rule for record types is

$$\frac{\Gamma, v :: T \rhd \sigma :: T}{\Gamma \rhd \mu v :: T.\sigma :: T}$$

and the natural subtyping rule for record types [Car86, BTCGS89] is

$$\frac{\Gamma, (s <: t) \rhd \sigma <: \tau, \quad \Gamma \rhd t :: T}{\Gamma \rhd (\mu v :: T.\sigma) <: (\mu v :: T.\tau)}$$

The usual term formation rules associated with recursive types are

$$(\mu\ I) \qquad \frac{\Gamma \rhd e : [\mu v :: T.\sigma/t]\sigma}{\Gamma \rhd abs\ e : \mu v :: T.\sigma}$$

$$(\mu\ E) \qquad \frac{\Gamma \rhd e : \mu v :: T.\sigma}{\Gamma \rhd rep\ e : [\mu v :: T.\sigma/t]\sigma}$$

where *abs* and *rep* are names for the isomorphism between $\mu v :: T.\sigma$ and $[\mu v :: T.\sigma/t]\sigma$. For simplicity of notation, we will omit *abs* and *rep* from expressions.