Simple, Efficient Object Encoding using Intersection Types

Karl Crary
Cornell University

April 23, 1998

Abstract

I present a type-theoretic encoding of objects that interprets method dispatch by self-application (*i.e.,* method functions are applied to the objects containing them) but still validates the expected subtyping relationships. The naive typing of self-application fails to validate the expected subtyping relationships because it is too permissive and allows application to similarly typed objects that are not self. This new encoding solves this problem by constraining methods to be applied only to self using existential and intersection types. Using this typing, I give a full account of objects including self types and method update. I also present another application of this object encoding to fully abstract, closure-passing closure conversion. The typing constructs used in this encoding appear to be quite rich, but they may be axiomatized in a novel, restricted fashion that is metatheoretically simple.

1 Introduction

Object-oriented programming languages usually provide built-in primitives for object-related computing. However, there is also considerable interest in explaining such object primitives in terms of type-theoretic constructs. Type-theoretic accounts of object systems are interesting for two main reasons: First, they provide a flexible framework in which to analyze object-oriented features and to explore combining them with other powerful programming features. Second, a type-preserving compiler must implement object features in terms of more basic, typed primitives. To satisfy both these needs, an type-theoretic object encoding must be faithful to the intended semantics (static and dynamic) of the object system, and must also be computationally efficient.

The *self-application* semantics [16] provides a reasonable explanation of the operational behavior of objects whose methods have access to self. In the self-application semantics, method invocation is performed by extracting the desired method from an object and then applying that method to the entire object as well as the method's arguments. Unfortunately, the naive typing of the self-application semantics does not justify the desired typing rules for objects; in particular, it blocks the expected subtyping relationship that objects with more methods may be used in place of objects with less.

This difficulty has led to several different proposals for type-theoretic encodings of objects. Recursive record interpretations [8, 9, 7] perform applications to self at the time objects are constructed, instead of at method invocation, resulting in records of methods where self is hardcoded. In existential interpretations [5, 24, 15], the self argument provides some hidden state of an object, but no access to methods; access to self methods is again settled before before object construction. Although each of these proposals supports basic functionality for object-oriented programming, none provide the full flexibility of the self-application semantics. For example, none allow methods to be updated once objects have been constructed.

To solve this problem, Abadi, Cardelli and Viswanathan devised an alternative interpretation [3], which retains the expressiveness of the self-application semantics. Their interpretation views objects as a record containing methods and a self field. The type of the self field is hidden, as in the existential interpretations, but is constrained to be a subtype of the full object's type. When objects are constructed, the self field is filled with a pointer to the object, and the pointer in that field is supplied to methods at method invocation, providing the essence of self-application.

Abadi *et al.'s* device provides a satisfactory model of objects in type theory. In particular, it justifies all the desired typing rules for objects and still allows the flexibility of the self-application semantics (such as method update). However, as an object encoding for use in a practical compiler, it results in some undesirable inefficiencies. As noted above, when invoking a method, the self argument is satisfied not by the object itself, but by the contents of a self field of the object. This means that a pointer to self must be stored in every object, which costs space, and that pointer must be extracted whenever methods are invoked, which costs time.

In an implementation where types may have different representations than their supertypes and therefore subsumption must involve the application of a coercion [4, 10], these costs are unavoidable. Methods expect their self parameter to supply the original object, not the result after some number of coercions, and consequently that original object must be retained. In such an implementation, the costs of storing and extracting self pointers are likely dwarfed by the necessary overhead of passing and applying coercions. However, in an implementation where types are included in their supertypes and subsumption is free, the self pointer just points back to the same data, making it operationally redundant, useful only for making the typing work out.

In this paper I present a type-theoretic encoding of objects that *directly* satisfies the self-application semantics, and therefore avoids the costs of the Abadi *et al.* encoding. I show that the naive typing of the self-application semantics is too permissive; it allows methods to be applied, not only to self, but to any object of the same type. However, that type may be a supertype of the original type, and therefore may be missing methods present in the original type. This means that methods cannot count on being supplied with all the methods they expect, even though those methods are present in the object itself. The solution arises by restricting the type so that methods are applied only to self. This is done using an existential type to abstract the type of self, and an intersection type to show that the object is both self and a collection of methods operating on self.

The ambient type theory required appears at first to be very rich, but I show that little of that expressiveness is required, and that the encoding may be performed in a simple and quite tractable type theory. At its core, neither bounded quantification nor higher-order type constructors are necessary (although there are good reasons to add both). The intersection type used is also well-behaved.

The paper is organized as follows: Section 2 informally develops the ideas of the encoding. Section 3 formalizes the encoding by presenting an object calculus and a translation that implements those objects with underlying type-theoretic constructs. The object calculus I present has considerable expressive power; for example, it supports a very natural encoding of the object calculus of Abadi and Cardelli [1, 2]. The underlying type theory makes explicit that certain operations (such as folding or unfolding recursive types) have no run-time effect, making it more straightforward to evaluate efficiency claims.

In Section 4, I show an application of this work to closure conversion: The mechanisms of this object encoding may be used to perform a closure-passing style of closure conversion, which improves efficiency over the environment-passing style, but retain full abstraction, which is not enjoyed by other closure-passing mechanisms. This technique follows naturally from the close connection between closures and objects. Some comparisons with other object encodings and concluding remarks appear in Section 5.

In the interest of brevity, this paper assumes basic familiarity with the Girard-Reynolds polymorphic lambda calculus [13, 25] and subtyping, with recursive types, and with existential types for data abstraction [18]. Some familiarity with the other object encodings discussed above will also be helpful, but is not required. Section 4 contains a brief review of the idea of closure conversion and typing mechanisms for it [17, 21]; some prior familiarity may be helpful there as well.

2 Informal Development

We begin by examining what makes the naive typing for self-application fail. By way of example, consider the object types `Point` and `ColorPoint` shown below.

$$\begin{array}{lcl} \texttt{Point} & \stackrel{\text{def}}{=} & \{\!|\, \texttt{getx} : \texttt{int} \,|\!\} \\ \texttt{ColorPoint} & \stackrel{\text{def}}{=} & \{\!|\, \texttt{getx} : \texttt{int}, \texttt{getc} : \texttt{color} \,|\!\} \end{array}$$

Since `ColorPoint` has all the methods of `Point`, we desire that `ColorPoint` be a subtype of `Point`. Unfortunately, this will not prove to be the case with the naive typing for self-application. In the naive typing, each object is encoded as a recursive record in which each method takes an entire object as an argument:

$$\begin{array}{lcl} \texttt{Point} & = & \mu\alpha.\{\texttt{getx} : \alpha \to \texttt{int}\} \\ & = & \{\texttt{getx} : \texttt{Point} \to \texttt{int}\} \\ \texttt{ColorPoint} & = & \mu\alpha.\{\texttt{getx} : \alpha \to \texttt{int}, \texttt{getc} : \alpha \to \texttt{color}\} \\ & = & \{\texttt{getx} : \texttt{ColorPoint} \to \texttt{int}, \texttt{getc} : \texttt{ColorPoint} \to \texttt{color}\} \end{array} \qquad \text{(naive)}$$

Suppose `cpt` is a `ColorPoint`. In order for `cpt` to be a member of `Point`, the `getx` field of `cpt` must be typeable as $\texttt{Point} \to \texttt{int}$. This is not the case; the `getx` field of `cpt` requires its argument be a `ColorPoint`, not just a `Point`. Consequently, `ColorPoint` is not a subtype of `Point` using the naive typing.

However, in the self-application semantics, the argument to the `getx` field is not just any object of type `Point`. The argument will always be the object `cpt` itself, which is not just a `Point` but is also a `ColorPoint`, as desired! Therefore, the intended subtyping should work out so long as an object's methods are always applied to the object itself, as promised by self-application. The problem with the naive typing is that it is too permissive; it allows applying methods to objects that are not self. In other words, the promise of self-application is broken by the naive typing.

What we require, then, is a typing mechanism that can require methods to be applied to a particular object. This is achievable using abstraction. Consider the existential type $\exists\alpha.\,\alpha\times(\alpha\to\tau)$. This type arises in typed closure conversion, where α is the (unknown) type of the environment, and $\alpha\to\tau$ is the type of code. Since the type α is unknown, nothing can be done with the environment except pass it to the code, and likewise the code cannot be called without presenting the environment as an argument.[1] This is a general mechanism, we may require that a function be called only with a specific argument simply by abstracting the type of the argument and packaging it with the function.

In order to ensure methods are called with the appropriate argument, we abstract the type of the argument and package it with the record of methods. But for self-application, the argument and the collection of methods are one and the same. Thus we package them using an intersection type to indicate that the same object is both the argument and the record of methods:

$$\begin{array}{lcl}\texttt{Point} & = & \exists\alpha.\,\alpha\wedge\{\texttt{getx}:\alpha\to\texttt{int}\}\\ \texttt{ColorPoint} & = & \exists\alpha.\,\alpha\wedge\{\texttt{getx}:\alpha\to\texttt{int},\texttt{getc}:\alpha\to\texttt{color}\}\end{array}$$

Informally, a term is a member of the intersection type $\tau_1\wedge\tau_2$ if it a member of both τ_1 and τ_2. For this encoding it is easily shown that `ColorPoint` is a subtype of `Point`, as desired. To invoke a method, we just unpack the existential, extract the desired method and apply it to the object. For example, let the invocation of method ℓ from object o be denoted by $o\leftarrow\ell$ and suppose o is a `Point`; then

$$o\leftarrow\texttt{getx}\quad\stackrel{\text{def}}{=}\quad\text{unpack}\,[\alpha,x]=o\,\text{in}\,(x.\texttt{getx})\,x$$

where $r.\ell$ denotes the extraction of field ℓ from record r. Note that $o\leftarrow\texttt{getx}$ has type `int`, as desired.

More generally, suppose O is $\{\!|\,\ell_1:\tau_1,\ldots,\ell_n:\tau_n\,|\!\}$. Then O is interpreted as:

$$\overline{O}\quad\stackrel{\text{def}}{=}\quad\exists\alpha.\,\alpha\wedge\{\ell_1:\alpha\to\tau_1,\ldots,\ell_n:\alpha\to\tau_n\}$$

I will refer to this encoding as the OEI encoding, for "objects using existential and intersection types," following the terminology of Bruce *et al.* [6]. In the remainder of this section, I will explore the expressiveness of this encoding by showing how it deals with various issues in object-oriented programming. The OEI encoding will not prove to be sufficient for all the mechanisms I wish to support, but I will introduce a variant encoding (called OREI) that is.

2.1 Object Construction

Let O^* be the naive encoding for the object type O:

$$O^*\quad\stackrel{\text{def}}{=}\quad\mu\alpha.\{\ell_1:\alpha\to\tau_1,\ldots,\ell_n:\alpha\to\tau_n\}$$

Suppose *po* is a "pre-object" of type O^*. By unwinding the recursive type in O^* once, *po* can also be given the type $\{\ell_1:O^*\to\tau_1,\ldots,\ell_n:O^*\to\tau_n\}$. Therefore, *po* can also be given the intersection type

$$O^*\wedge\{\ell_1:O^*\to\tau_1,\ldots,\ell_n:O^*\to\tau_n\}$$

[1]Throughout this paper, I assume call-by-value evaluation; therefore the argument cannot be spoofed with a nonterminating expression of type α.

and so an object of type $\overline{O}$ may be constructed simply by hiding O^*:

$$\text{pack } po \text{ as } \exists\alpha.\,\alpha \wedge \{\ell_1 : \alpha \to \tau_1, \ldots, \ell_n : \alpha \to \tau_n\} \text{ hiding } O^*$$

Moreover, this packing operation has no run-time effect, provided we assume the implementation erases types at run time. (I discuss this assumption further in Section 3.)

This technique for object construction is satisfactory so long as we are willing for the type of methods' self arguments to be different than the actual object's type (O^* versus $\overline{O}$), but when I introduce an object calculus it will prove more convenient to have only one interpretation of object types that is used both internally and externally. Fortunately, this is easily achieved. Suppose $m_1, \ldots, m_n$ is a set of methods such that m_i has type $\overline{O} \to \tau_i$. We may create methods m'_i that instead take O^* as their argument by eta-expanding the method (to gain access to the argument) and packing the argument just as we packed the pre-object before:

$$m'_i \stackrel{\text{def}}{=} \lambda x{:}O^*.\, m_i\,(\text{pack } x \text{ as } \overline{O} \text{ hiding } O^*)$$

The full object may then be constructed as before:

$$\text{pack } \{\ell_1 = m'_1, \ldots, \ell_n = m'_n\} \text{ as } \overline{O} \text{ hiding } O^*$$

Unfortunately, the construction of m'_i from m_i has an obvious run-time effect, so it is no longer the case that object construction is free. Furthermore, the eta-expansion results in the cost of an additional function call for every method invocation, which likely to be quite expensive. These problems are resolved in Section 3 by a new construct added to the type theory that coerces function arguments without eta-expanding the functions.

2.2 Self Types

An important feature for an object encoding is to support methods whose type involves the "type of self." For example, the `Point` object type from before may be augmented with methods that functionally set or increment the point's position, returning a new point:

$$\texttt{Point} \stackrel{\text{def}}{=} \{\!|\, \texttt{getx} : \texttt{int}, \texttt{setx} : \texttt{int} \to \alpha, \texttt{incx} : \alpha \,|\!\} \text{ as } \alpha$$

In the above type, the type variable α stands for the type of self ("as α" serves as the binding occurrence for the self type variable). Thus the `setx` method takes an integer and returns a new object of type `Point`.

When interpreted using the OEI encoding, what we desire is a solution to the equation:

$$\texttt{Point} = \exists\beta.\,\beta \wedge \{\texttt{getx} : \beta \to \texttt{int}, \texttt{setx} : \beta \to \texttt{int} \to \texttt{Point}, \texttt{incx} : \beta \to \texttt{Point}\}$$

The solution is obtained in the natural manner, by wrapping an additional recursive type around the encoding:

$$\texttt{Point} = \mu\alpha.\,\exists\beta.\,\beta \wedge \{\texttt{getx} : \beta \to \texttt{int}, \texttt{setx} : \beta \to \texttt{int} \to \alpha, \texttt{incx} : \beta \to \alpha\}$$

More generally, suppose O is $\{\!|\, \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \,|\!\}$ as α. Then O is interpreted as:

$$\overline{O} \stackrel{\text{def}}{=} \mu\alpha.\,\exists\beta.\,\beta \wedge \{\ell_1 : \beta \to \tau_1, \ldots, \ell_n : \beta \to \tau_n\}$$

Note that the recursive variable α may appear free in τ_i. I will refer to this encoding as the OREI encoding, for "objects using recursive, existential and intersection types."

2.3 Method Update

If a method is to return a new object of self type, it must do so by updating some of the methods (or by returning the original object unchanged). Some methods will do this simply by calling other methods; for example, the `incx` method may create a new point by calling the `setx` method. Other methods will do so directly; for example, the `setx` method is intended to return a new object with an updated `getx` method.

We may implement such a point as follows:

$$
\begin{array}{llll}
m_{\texttt{getx}} & : & \overline{\texttt{Point}} \to \texttt{int} & = \lambda o{:}\overline{\texttt{Point}}.\, 12 \\
m_{\texttt{setx}} & : & \texttt{Point}^* \to \texttt{int} \to \overline{\texttt{Point}} & = \lambda o{:}\texttt{Point}^*.\, \lambda x{:}\texttt{int}. \\
 & & & \qquad \text{pack } \{\texttt{getx} = \lambda o'{:}\texttt{Point}^*.\, x, \\
 & & & \qquad\qquad \texttt{setx} = o.\texttt{setx} \\
 & & & \qquad\qquad \texttt{incx} = o.\texttt{incx}\} \text{ as } \overline{\texttt{Point}} \text{ hiding } \texttt{Point}^* \\
m_{\texttt{incx}} & : & \overline{\texttt{Point}} \to \overline{\texttt{Point}} & = \lambda o{:}\overline{\texttt{Point}}.\, (o \leftarrow \texttt{setx})(o \leftarrow \texttt{getx} + 1) \\
m'_{\texttt{getx}} & : & \texttt{Point}^* \to \texttt{int} & = \lambda x{:}\texttt{Point}^*.\, m_{\texttt{getx}}(\text{pack } x \text{ as } \overline{O} \text{ hiding } O^*) \\
m'_{\texttt{incx}} & : & \texttt{Point}^* \to \overline{\texttt{Point}} & = \lambda x{:}\texttt{Point}^*.\, m_{\texttt{incx}}(\text{pack } x \text{ as } \overline{O} \text{ hiding } O^*) \\
pt & : & \overline{\texttt{Point}} & = \text{pack } \{\texttt{getx} = m'_{\texttt{getx}}, \texttt{setx} = m_{\texttt{setx}}, \texttt{incx} = m'_{\texttt{incx}}\} \\
 & & & \quad \text{as } \overline{\texttt{Point}} \text{ hiding } \texttt{Point}^*
\end{array}
$$

In this example, we are able to implement the `getx` and `incx` methods by functions taking the standard representation $\overline{\texttt{Point}}$, as before. Unfortunately, the `setx` method requires the pre-object representation $\texttt{Point}^*$. Also, while the `getx` and `incx` methods are simple, the `setx` method is relatively complicated and requires access to the internals of the encoding. We would like a mechanism to hide the details of method update from the programmer.

An Object Calculus As has been argued by Abadi and Cardelli [1, 2], object calculi provide a useful level of abstraction for object-oriented programming. In particular, the object calculus I now introduce provides such a mechanism for hiding method update from the programmer. Suppose O is $\{\!|\, \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \,|\!\}$ as α. An object of type $\overline{O}$ is constructed by the expression $\{\!|\, \ell_1 = M_1, \ldots, \ell_n = M_n \,|\!\}$, where each method implementation M_i is either

- some term e_i having type $\overline{O} \to \tau_i[\overline{O}/\alpha]$, or
- $\Upsilon(\ell_j)$, where $\tau_i = (\alpha \to \tau_j) \to \alpha$

Methods implemented by $\Upsilon(j)$ are update methods, taking a method implementation for method ℓ_j and returning a new object using that implementation instead. Similar facilities are provided by Abadi and Cardelli's object calculus, except that their update methods are implicit, while those of my calculus are explicit and invoked in the ordinary manner (and may even themselves be overridden by updates).

In this object calculus, points may be implemented as follows:

$$
\begin{array}{lcl}
\texttt{Point2} & = & \{\!|\, \texttt{getx} : \texttt{int}, \texttt{setx} : \texttt{int} \to \alpha, \texttt{incx} : \alpha, \texttt{update\_getx} : (\alpha \to \texttt{int}) \to \alpha \,|\!\} \text{ as } \alpha \\
pt2 : \overline{\texttt{Point2}} & = & \{\!|\, \texttt{getx} = \lambda o{:}\overline{\texttt{Point2}}.\, 12 \\
 & & \quad \texttt{setx} = \lambda o{:}\overline{\texttt{Point2}}.\, \lambda x{:}\texttt{int}.\, (o \leftarrow \texttt{update\_getx})(\lambda o'{:}\overline{\texttt{Point2}}.\, x) \\
 & & \quad \texttt{incx} = \lambda o{:}\overline{\texttt{Point2}}.\, (o \leftarrow \texttt{setx})(o \leftarrow \texttt{getx} + 1) \\
 & & \quad \texttt{update\_getx} = \Upsilon(\texttt{getx}) \,|\!\} \\
pt : \overline{\texttt{Point}} & = & pt2 \qquad \text{(using subtyping to forget the extra method } \texttt{update\_getx}\text{)}
\end{array}
$$

Note that all the visible methods take $\overline{\texttt{Point}}$ as their argument and that the `setx` method has a simple implementation. The mechanism previously used for the `setx` method remains in the `update_getx` method, but the object calculus has hidden that mechanism from the programmer.

2.4 A Simplified Type Theory

In the preceding development I have been using quite a rich type system. For example, intersection types are a critical part of my object encoding. On their own, intersection types are fairly innocuous, but combining them with them with bounded quantification leads to serious difficulties for type checking and semantics [23]. I do not use bounded quantification in this paper, but there are many good reasons to want to include it in a practical object system.

For another example, when packaging pre-objects in Section 2.1, I implicitly made use of a rule stating that terms belonging to the recursive type $\mu\alpha.\tau$ also belong to the once-unrolled version of that type $\tau[\mu\alpha.\tau/\alpha]$. This rule is natural according to the intuitive semantics of the recursive type, but it makes type checking considerably more difficult and it restricts the possible models of the type theory [22]. The usual solution to this difficulty is to use explicit fold and unfold operations between $\mu\alpha.\tau$ and $\tau[\mu\alpha.\tau/\alpha]$, but this solution cannot be applied directly in my setting: Suppose e has type $\mu\alpha.\tau$; although *unfold* e has type $\tau[\mu\alpha.\tau/\alpha]$, neither e nor *unfold* e has the required type $\mu\alpha.\tau \wedge \tau[\mu\alpha.\tau/\alpha]$.

The difficulties resulting from the richness of the type system may lead the reader to wonder about the practical applicability of the OREI encoding. Fortunately, both the intersection type and the recursive type are used only in a restricted, stylized manner. In ordinary intersection type calculi, an important feature of the intersection type $\tau_1 \wedge \tau_2$ is that it is a subtype of both τ_1 and τ_2. However, I do not depend on this subtyping relationship in order to get the desired subtyping relationship on objects.

In Section 3, I formalize a simplified type theory that instead uses the considerably weaker subtyping rule

$$\frac{\Gamma \vdash \tau_1 \leq \tau_1' \quad \Gamma \vdash \tau_2 \leq \tau_2'}{\Gamma \vdash \tau_1 \wedge \tau_2 \leq \tau_1' \wedge \tau_2'}$$

and requires an explicit coercion to convert $\tau_1 \wedge \tau_2$ to either τ_1 or τ_2. This makes the intersection type closely resemble the ordinary product type, and gives it similar metatheoretic properties.[2]

Members of intersection types are introduced by taking a single term and applying two coercions, each of which has no run-time effect. The problem with recursive types discussed above is then

[2]Dimock *et al.* [12] make use of a similar idea by defining intersections to be products where the components are required to be identical when types are erased.

$$
\begin{array}{llll}
\text{types} & \tau & ::= & \alpha \mid \texttt{int} \mid \tau_1 \to \tau_2 \mid \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \mid \tau_1 \wedge \tau_2 \mid \forall\alpha.\tau \mid \exists\alpha.\tau \mid \mu\alpha.\tau \mid \texttt{top} \\
\text{terms} & e & ::= & x \mid i \mid \lambda x{:}\tau.e \mid e_1 e_2 \mid \{\ell_1 = e_1, \ldots, \ell_n = e_n\} \mid e.\ell \mid \Lambda\alpha.e \mid e[\tau] \mid \\
 & & & \quad \text{unpack}\,[\alpha, x] = e_1 \text{ in } e_2 \mid c\,e \\
\text{coercions} & c & ::= & \text{id}[\tau] \mid c_1 \to c_2 \mid \{\ell_1 : c_1, \ldots, \ell_n : c_n\} \mid c_1 \wedge c_2 \mid \langle c_1, c_2\rangle \mid \pi_i[\tau_1 \wedge \tau_2] \mid \\
 & & & \quad \forall\alpha.c \mid \exists\alpha.c \mid \text{hide}\,\tau_1 \text{ in } \exists\alpha.\tau_2 \mid \text{fold}[\mu\alpha.\tau] \mid \text{unfold}[\mu\alpha.\tau] \mid c_1 \circ c_2 \\
\text{contexts} & \Gamma & ::= & \bullet \mid \Gamma[\alpha] \mid \Gamma[x : \tau]
\end{array}
$$

Figure 1: Syntax of F_C

handled by coercing a recursively type term with both an identity coercion and an unfold coercion. Each of these coercions belong to a coercion calculus of operators that change types without any run-time effect. This calculus also includes higher-order coercion constructors that make it possible, for example, to coerce the argument to a function without changing the function itself. This resolves the eta-expansion problem observed in Section 2.1.

3 Formal Presentation

I begin my formal account of the OREI encoding by presenting the underlying type theory, called F_C, that I discussed in the previous section. Then I will formalize the object calculus built over top of it and show how to encode it in F_C.

The syntax of F_C is given in Figure 1. Most of its constructs are standard. The type $\tau_1 \wedge \tau_2$ denoted the restricted intersection type discussed in the previous section. The type `top` is a supertype of all well-formed types. As usual, expressions that differ only by alpha-conversion are considered identical.

The novelty of F_C is in the coercion constructors. The constructor $\mathit{hide}\,\tau_1\ \mathit{in}\ \exists\alpha.\tau_2$ constructs existential packages where the type τ_1 is hidden. The constructors $\mathit{fold}[\mu\alpha.\tau]$ and $\mathit{unfold}[\mu\alpha.\tau]$ roll and unroll recursive types. Values of intersection types are constructed with the $\langle c_1, c_2\rangle$ coercion: if c_1 takes τ to τ_1 and c_2 takes τ to τ_2 then $\langle c_1, c_2\rangle$ takes τ to $\tau_1 \wedge \tau_2$. Intersection types are destructed with the $\pi_i[\tau_1 \wedge \tau_2]$ coercion, which takes $\tau_1 \wedge \tau_2$ to τ_i.

The identity coercion is denoted by $\mathit{id}[\tau]$ and coercions are composed by $c_1 \circ c_2$. The remaining constructors extend these basic constructors to higher types. The constructor $c_1 \to c_2$ coerces functions by applying c_1 to the argument and c_2 to the result. The constructor $\{\ell_1 : c_1, \ldots, \ell_n : c_n\}$ coerces records by applying each coercion to the appropriate field. The other higher-order coercion constructors work in the obvious manner. (Not all of the higher-order coercion constructors are actually used in the object encoding. I include them all here in the interest of completeness, since they add little extra complexity.)

In order to interpret coercions as having no run-time effect, it is necessary to assume an interpretation where types are erased at run time. This is desirable not only because it makes more efficient the object encoding of this paper, but also because it avoids hidden costs of passing run-time type information in general and it considerably simplifies polymorphic closure conversion [21]. However, it complicates support for run-time type analysis [11], which is required for some advanced

$$\frac{\Gamma \vdash \tau_i \leq \tau_i' \quad (\text{for } 1 \leq i \leq m) \quad \Gamma \vdash \tau_i \text{ type} \quad (\text{for } m < i \leq n)}{\Gamma \vdash \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \leq \{\ell_1 : \tau_1', \ldots, \ell_m : \tau_m'\}} \; (m \leq n)$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \qquad \frac{\Gamma \vdash c : \tau_1 \Rightarrow \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash c\, e : \tau_2}$$

$$\frac{\Gamma \vdash c_1 : \tau \Rightarrow \tau_1 \quad \Gamma \vdash c_2 : \tau \Rightarrow \tau_2}{\Gamma \vdash \langle c_1, c_2 \rangle : \tau \Rightarrow \tau_1 \wedge \tau_2} \qquad \frac{\Gamma \vdash \tau_1 \wedge \tau_2 \text{ type}}{\Gamma \vdash \pi_i[\tau_1 \wedge \tau_2] : \tau_1 \wedge \tau_2 \Rightarrow \tau_i} \; (i = 1, 2)$$

$$\frac{\Gamma \vdash c_1 : \tau_1' \Rightarrow \tau_1 \quad \Gamma \vdash c_2 : \tau_2 \Rightarrow \tau_2'}{\Gamma \vdash c_1 \to c_2 : (\tau_1 \to \tau_2) \Rightarrow (\tau_1' \to \tau_2')} \qquad \frac{\Gamma \vdash c_1 : \tau_2 \Rightarrow \tau_3 \quad \Gamma \vdash c_2 : \tau_1 \Rightarrow \tau_2}{\Gamma \vdash c_1 \circ c_2 : \tau_1 \Rightarrow \tau_3}$$

$$\frac{\Gamma \vdash \mu\alpha.\tau \text{ type}}{\Gamma \vdash \text{fold}[\mu\alpha.\tau] : \tau[\mu\alpha.\tau/\alpha] \Rightarrow \mu\alpha.\tau} \qquad \frac{\Gamma \vdash \mu\alpha.\tau \text{ type}}{\Gamma \vdash \text{unfold}[\mu\alpha.\tau] : \mu\alpha.\tau \Rightarrow \tau[\mu\alpha.\tau/\alpha]}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \leq \tau'}{\Gamma \vdash e : \tau'} \qquad \frac{\Gamma \vdash c : \tau_1 \Rightarrow \tau_2 \quad \Gamma \vdash \tau_1' \leq \tau_1 \quad \Gamma \vdash \tau_2 \leq \tau_2'}{\Gamma \vdash c : \tau_1' \Rightarrow \tau_2'}$$

Figure 2: Selected rules of F_C

implementation techniques [14, 19, 27, 20, 26].

The typing rules of F_C are given in Appendix A and consist of four judgements. The usual judgements for type well-formedness, subtyping and typing of terms are written $\Gamma \vdash \tau$ type; $\Gamma \vdash \tau_1 \leq \tau_2$; and $\Gamma \vdash e : \tau$. The typing judgement for coercions is written $\Gamma \vdash c : \tau_1 \Rightarrow \tau_2$ and indicates that c takes members of type τ_1 to type τ_2. Some representative rules appear in Figure 2. The typing rules for coercions are well-behaved and the remaining rules are standard, so it is easy to show the decidability of type checking in F_C.

A somewhat unusual design feature of F_C is that subtypes always have the same data representations as their supertypes. This is important because it ensures there is no run-time effect of subsumption. In contrast, a calculus where supertypes may have different representations requires the expense of passing and applying coercion functions at run-time [4, 10] (or the even greater expense of run-time type checking and coercion). The main impact of this design decision is a restricted rule for record subtyping: permuted records are not considered identical, and record subtyping respects extension only on the right. The main practical impact of restricting record subtyping in this way is that it makes multiple inheritance impossible.

It is worthwhile to note that this design decision is independent of the other contributions of the paper. The results I present are all valid with the more permissive record subtyping (in fact a few results are easier), but they then correspond to a less efficient implementation. In the sequel, I point out the particular places in which this decision has an impact.

3.1 An Object Calculus

I extend the calculus F_C to an object calculus (F_C+*obj*) by adding the following object constructs:

$$\begin{array}{lrcl} \text{types} & \tau & ::= & \cdots \mid \{\!|\, \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \,|\!\} \text{ as } \alpha \\ \text{terms} & e & ::= & \cdots \mid \{\!|\, \ell_1 : \tau_1 = M_1, \ldots, \ell_n : \tau_n = M_n \,|\!\} \text{ as } \alpha \mid e \overset{\tau}{\leftarrow} \ell \\ \text{methods} & M & ::= & e \mid \Upsilon(\ell) \end{array}$$

$$\frac{\Gamma[\alpha] \vdash_{obj} \tau_i \ \text{type} \quad (\text{for } 1 \leq i \leq n)}{\Gamma \vdash_{obj} \{\!| \ell_1 : \tau_1, \ldots, \ell_n : \tau_n |\!\} \text{ as } \alpha \ \text{type}}$$

$$\frac{\Gamma[\alpha] \vdash_{obj} \tau_i \leq \tau_i' \quad (\text{for } 1 \leq i \leq m) \quad \Gamma[\alpha] \vdash_{obj} \tau_i \ \text{type} \quad (\text{for } m < i \leq n)}{\Gamma \{\!| \ell_1 : \tau_1, \ldots, \ell_n : \tau_n |\!\} \text{ as } \alpha \leq \{\!| \ell_1 : \tau_1', \ldots, \ell_m : \tau_m' |\!\} \text{ as } \alpha} \ (m \leq n)$$

$$\frac{\Gamma \vdash_{obj} e : \tau}{\Gamma \vdash_{obj} e \stackrel{\tau}{\leftarrow} \ell_i : \tau_i[\tau/\alpha]} \ (\tau = \{\!| \ell_1 : \tau_1, \ldots, \ell_n : \tau_n |\!\} \text{ as } \alpha)$$

$$\frac{\Gamma \vdash_{obj} \tau \ \text{type} \quad \Gamma, \tau \vdash_{obj} M_i : \ell_i \quad (\text{for } 1 \leq i \leq n)}{\Gamma \vdash_{obj} \{\!| \ell_1 : \tau_1 = M_1, \ldots, \ell_n : \tau_n = M_n |\!\} \text{ as } \alpha : \tau} \ (\tau = \{\!| \ell_1 : \tau_1, \ldots, \ell_n : \tau_n |\!\} \text{ as } \alpha)$$

$$\frac{\Gamma \vdash_{obj} e : \tau \to \tau_i[\tau/\alpha]}{\Gamma, \tau \vdash_{obj} e : \ell_i} \ (\tau = \{\!| \ell_1 : \tau_1, \ldots, \ell_n : \tau_n |\!\} \text{ as } \alpha)$$

$$\frac{}{\Gamma, \{\!| \ell_1 : \tau_1, \ldots, \ell_n : \tau_n |\!\} \text{ as } \alpha \vdash_{obj} \Upsilon(\ell_j) : \ell_i} \ (\tau_i = (\alpha \to \tau_j) \to \alpha)$$

Figure 3: Additional typing rules for $F_C{+}obj$

The type $\{\!| \ell_1 : \tau_1, \ldots, \ell_n : \tau_n |\!\}$ as α is the type of objects where method ℓ_i has type τ_i, in which α stands for the type of self. Method implementations M are either simple terms or update methods $\Upsilon(\ell)$ for method ℓ. In method terms, the self variable α is bound in the type annotations τ_i, but *not* in the method implementations M_i. In both object types and terms, if the self variable alpha does not appear free in any of the types τ_i, I will sometimes omit the "as α" suffix. Method invocations are also annotated with their object's type. When the appropriate types are clear, I will often omit all the type annotations from object construction and method invocation terms. In a practical system, this information would be supplied by type inference.

The new typing rules for objects appear in Figure 3. I indicate judgements in the extended $F_C{+}obj$ calculus by $\vdash_{obj}$. There is one new judgement form for the object calculus: the method typing judgement $\Gamma, \tau \vdash_{obj} M : \ell$ (where τ is an object type) indicates that the method implementation M may serve as method ℓ in an object of type τ.

This object calculus is very similar in expressive power to Abadi and Cardelli's first-order calculus of primitive objects $Ob_{1<:}$ [1, 2]. The difference is that in $Ob_{1<:}$ all methods are updatable; explicit update methods are not required as in $F_C{+}obj$. Thus, $Ob_{1<:}$ may be encoded in $F_C{+}obj$ by a translation that adds explicit update methods (as in the object interpretation of Abadi, *et al.* [3]).[3]

3.2 Translation

I am now ready to describe the encoding of objects in F_C. The encoding is specified by two mappings, a mapping $\overline{\tau}$ of types τ from $F_C{+}obj$ into F_C, and a mapping $|e|$ of terms e from $F_C{+}obj$ into F_C. (I do not define a mapping for coercions, since coercions are intended for the implementation of objects, but it would be straightforward to add a higher-order coercion for objects, and then to define a mapping of that coercion on to underlying coercions.) Each mapping is specified by induction on the syntactic structure. I give the mapping only for the object constructs;

[3]This translation applies to $Ob_{1<:}$ restricted so that subtyping respects extension only on the right, as in $F_C{+}obj$. To encode full $Ob_{1<:}$ requires the variant of $F_C{+}obj$ that permits extension in the middle.

the remaining cases are handled in the obvious manner. Object types are translated using OREI exactly as in Section 2.2:

$$\overline{\{\!|\, \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \,|\!\}\text{ as }\alpha} \stackrel{\text{def}}{=} \mu\alpha.\, \exists\beta.\, \beta \wedge \{\ell_1 : \beta \to \overline{\tau_1}, \ldots, \ell_n : \beta \to \overline{\tau_n}\}$$
$$\text{(where } \beta \text{ is not free in } \tau_1, \ldots, \tau_n\text{)}$$

Before I specify the term translation, I first define some convenient notation. Suppose O is the object type $\{\!|\, \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \,|\!\}$ as α. Then O^* is the type of O "pre-objects" and O^{**} is that same type unrolled once:

$$\begin{aligned} O^* &\stackrel{\text{def}}{=} \mu\beta.\, \{\ell_1 : \beta \to \overline{\tau_1}[\overline{O}/\alpha], \ldots, \ell_n : \beta \to \overline{\tau_n}[\overline{O}/\alpha]\} \\ &\phantom{\stackrel{\text{def}}{=}} \text{(where } \beta \text{ is not free in } \tau_1, \ldots, \tau_n\text{)} \\ O^{**} &\stackrel{\text{def}}{=} \{\ell_1 : O^* \to \overline{\tau_1}[\overline{O}/\alpha], \ldots, \ell_n : O^* \to \overline{\tau_n}[\overline{O}/\alpha]\} \end{aligned}$$

The coercion $\mathcal{C}(O) : O^{**} \Rightarrow \overline{O}$ coerces pre-objects to objects (without any run-time action):

$$\begin{aligned} \mathcal{C}(O) \stackrel{\text{def}}{=} \;& \text{fold}[\overline{O}] \circ \\ & \text{hide}\, O^* \text{ in } \exists\beta.\, \beta \wedge \{\ell_1 : \beta \to \overline{\tau_1}[\overline{O}/\alpha], \ldots, \ell_n : \beta \to \overline{\tau_n}[\overline{O}/\alpha]\} \circ \\ & \langle \text{fold}[O^*], \text{id}[O^{**}] \rangle \\ & \text{(where } \beta \text{ is not free in } \tau_1, \ldots, \tau_n\text{)} \end{aligned}$$

The method coercion $\mathcal{M}(O, \sigma) : (\overline{O} \to \sigma) \Rightarrow (O^* \to \sigma)$ takes method implementations for O, which accept the external object type $\overline{O}$, and transforms them to functions that accept pre-methods of type O^*:

$$\mathcal{M}(O, \sigma) \stackrel{\text{def}}{=} (\mathcal{C}(O) \circ \text{unfold}[O^*]) \to \text{id}[\sigma]$$

An auxiliary translation $|M|_{O,\ell_i}$ translates the method implementation M to an appropriate term for method ℓ_i of a pre-object of type O; that is, it returns a function with type $O^* \to \tau_i[\overline{O}/\alpha]$ (still assuming that O is $\{\!|\, \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \,|\!\}$ as α). For ordinary methods it does this just by applying $\mathcal{M}$; for update methods it returns an update function:

$$\begin{aligned} |e|_{O,\ell_i} &\stackrel{\text{def}}{=} \mathcal{M}(O, \tau_i[\overline{O}/\alpha])\, e \\ |\Upsilon(\ell_j)|_{O,\ell_i} &\stackrel{\text{def}}{=} \lambda o{:}O^*.\, \lambda m{:}(\overline{O} \to \tau_i[\overline{O}/\alpha]). \\ &\qquad \mathcal{C}(O)\, \{\ell_1 = (\text{unfold}[O^*]\, o).\ell_1, \ldots, \ell_{j-1} = (\text{unfold}[O^*]\, o).\ell_{j-1}, \\ &\qquad\qquad \ell_j = \mathcal{M}(O, \tau_j[\overline{O}/\alpha])\, m, \\ &\qquad\qquad \ell_{j+1} = (\text{unfold}[O^*]\, o).\ell_{j+1}, \ldots, \ell_n = (\text{unfold}[O^*]\, o).\ell_n\} \end{aligned}$$

Then the term encoding is defined as follows:

$$\begin{aligned} |\{\!|\, \ell_1 : \tau_1 = M_1, \ldots, \ell_n : \tau_n = M_n \,|\!\}\text{ as }\alpha| &\stackrel{\text{def}}{=} \mathcal{C}(\tau)\, \{\ell_1 = |M_1|_{\tau,\ell_1}, \ldots, \ell_n = |M_n|_{\tau,\ell_n}\} \\ |e \stackrel{\tau}{\leftarrow} \ell| &\stackrel{\text{def}}{=} \text{unpack}\,[\beta, x] = \text{unfold}[\overline{\tau}]\, e \text{ in} \\ &\qquad (\pi_2[\tau']\, x).\ell\, (\pi_1[\tau']\, x) \end{aligned}$$

$$\begin{aligned} &\text{(where } \tau \text{ is } \{\!|\, \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \,|\!\}\text{ as }\alpha\text{, and} \\ &\beta \text{ is not free in } \tau_1, \ldots, \tau_n\text{, and} \\ &\tau' \text{ is } \beta \wedge \{\ell_1 : \beta \to \overline{\tau_1}[\overline{\tau}/\alpha], \ldots, \ell_n : \beta \to \overline{\tau_n}[\overline{\tau}/\alpha]\}\text{)} \end{aligned}$$

These encodings introduce no (run-time) overhead: Method invocation only extracts the desired method and applies it to the object. Object construction assembles the methods into an object, adding only the update methods and those are closed and can be pre-allocated. In each case, most of the term is just notation to coax the types through, and in practice that notation will be visible only to the compiler.

The natural type correctness result is easy to show:

Proposition 3.1 *Let the mapping $\overline{\Gamma}$ be the obvious extension of the type mapping to contexts; then:*

- *If* $\Gamma \vdash_{obj} \tau$ type *then* $\overline{\Gamma} \vdash \overline{\tau}$ type.
- *If* $\Gamma \vdash_{obj} \tau_1 \leq \tau_2$ *then* $\overline{\Gamma} \vdash \overline{\tau_1} \leq \overline{\tau_2}$.
- *If* $\Gamma \vdash_{obj} e : \tau$ *then* $\overline{\Gamma} \vdash |e| : \overline{\tau}$.
- *If* $\Gamma, O \vdash_{obj} M : \ell_i$ *(where* O *is* $\{\!| \ell_1 : \tau_1, \ldots, \ell_n : \tau_n |\!\}$ as α*) then* $\overline{\Gamma} \vdash |M|_{O,\ell_i} : O^* \rightarrow \tau_i[\overline{O}/\alpha]$.

In a formalized operational semantics, dynamic correctness may also be shown, using a straightforward (and uninteresting) simulation argument.

4 Fully Abstract Closure Passing

Compilers for languages that support first-class functions typically perform the *closure conversion* program transformation. Closure conversion makes all functions closed by rewriting them to take an extra environment argument, and rewriting their bodies so that free variables are replaced by references to the new environment argument. Functions, which are then closed, may be preallocated at compile time. The free variables are placed in an environment that is supplied to the function when it is called. First-class functions may then be supported by allocating *closures*, which contain the pre-allocated, closed code and the relevant environment.

Suppose f has type $\tau_1 \rightarrow \tau_2$, then the type of f's closure would be $\exists\alpha. \{\text{code} : (\tau_1 \times \alpha) \rightarrow \tau_2, \text{env} : \alpha\}$ [17], where α is the type of the environment. The abstraction of the environment's type is made abstract serves two important purposes. The first is to ensure that that all closures for functions $\tau_1 \rightarrow \tau_2$ have the same type, regardless of the number and types of their free variables. The second is to ensure that closures are *fully abstract.*

Full abstraction for closures states that no operations may be performed on closures that may not be performed on the original functions. The only operation that may be performed on a function is to call it. That is also the only operation that may be performed on a closure; since the type of the environment is abstract, nothing may be done with it but pass it to the code, and similarly, the code cannot be called without the appropriate environment. In contrast, if the environment's type were made transparent, the environment could be analyzed or a new environment could be synthesized for the code. Full abstraction in compilation is not only of theoretical interest; in systems where programmers may write code in lower-level intermediate languages [21], it is desirable that abstraction properties in the source language be protected in the lower-level intermediate languages as well.

Although the above type for closures is fully abstract, it is not efficient for recursive functions. The function f is passed only the environment; so a new closure must be constructed if f is used within the body. This costs an allocation and initialization of a closure for every recursive call. Fortunately, such costs may be avoided by passing the entire closure, instead of just the environment. The naive typing for such a closure-passing style of closure conversion would be $\exists\alpha.\, \mu\beta.\, \{\text{code} : (\tau_1 \times \beta) \to \tau_2,\ \text{env} : \alpha\}$.

Unfortunately, this typing sacrifices full abstraction. The abstraction of α prevents meddling with the environment (the *env* field), but it does not prevent altering the *code* field. That is, it is possible to construct a new closure with different code (but an identical type), and pass that new closure to the original code. Such a modified function call executes the original code but turns over control to the new code on every recursive call; this facility might not be desirable (for reasons of abstraction) but certainly it is not possible at the source level.

The problem here is the same one that faced the self-application semantics for objects: that the naive typing allows the code to be applied to any closure of the appropriate type, not just self. This problem can be solved in a now familiar manner, using abstraction and an intersection type:

$$f_{\text{closure}} : \exists\alpha.\, \alpha \wedge \{\text{code} : (\tau_1 \times \alpha) \to \tau_2\}$$

This type amounts to the object type $\{\!|\, \text{code} : \tau_1 \to \tau_2 \,|\!\}$, where the contents of the environment are instance variables in the object that have been hidden by subtyping. The code method can only be applied with its own closure (object), regaining full abstraction.

This is but one example of the close relationship that exists between closures and objects. The simpler, environment-passing style of closures discussed at the start of this section (with type $\exists\alpha.\, \{\text{code} : (\tau_1 \times \alpha) \to \tau_2,\ \text{env} : \alpha\}$) also corresponds to an object with a single *code* method, but using the existential object encoding of Pierce and Turner [24] instead.

5 Conclusions and Comparisons

The OREI encoding is the first type-theoretic object encoding to use the efficient self-application semantics to explain objects' operational behavior and also to give objects types that justify the intended subtyping relationships. The enabling observations are that the typing of objects must enforce that objects are used only in a self-applicative manner, and that such enforcement may be done simply, using abstraction and restricted intersection types.

The OREI encoding may be understood as occupying a far end of the spectrum of abstraction-oriented object encodings examined by Bruce *et al.* [6]. On the end opposite OREI is the purely existential (OE) encoding of Pierce, Turner, and Hoffman [24, 15]. Their encoding views objects as pairs of state and methods, where the structure of the state is completely unspecified. Functional update methods return only the state, which must be paired again with the methods by the caller to produce a new object. Next is the (ORE) encoding of Bruce [5]. The ORE encoding is like the OE encoding except that the repacking of new state with methods to form objects is performed by the method, rather than the caller. However, like OE, the ORE encoding leaves the structure of the state unspecified. Next is the (ORBE) encoding of Abadi, Cardelli and Viswanathan [3], which additional commits that an object's state will be another object of the same type. (In practice the state is actually the same object, but the type leaves open the possibility it might be different.) Finally, OREI fits in as the most specific of the group; the type specifies that an object's state is the object itself, not just any object.

OREI bears the greatest resemblance to the ORE encoding of Bruce. In ORE an object of type $\{\!| \ell_1 : \tau_1, \ldots, \ell_n : \tau_n |\!\}$ as α is encoded by an expression with type

$$\mu\alpha.\, \exists\beta.\, \beta \times \{\ell_1 : \beta \to \tau_1, \ldots, \ell_n : \beta \to \tau_n\}$$

which differs from the type given by OREI only in the replacement of OREI's intersection type with an ordinary product. In other words, OREI is just ORE where the state and methods are constrained to be the same. This similarity allows OREI to enjoy many of the same properties as ORE, such as the ability to support binary methods (at the expense of some subtyping).

The novel axiomatization of the intersection type in F_C may lead the reader to wonder about its essential nature (and perhaps whether it should properly be called an intersection type at all). Intersection types are typically associated with their import in subtyping systems, but that import is removed by the F_C calculus. What remains in F_C (and what is used by the OREI encoding) is the import of intersection types for controlled information hiding. Existential types are used [18] to hide type information by replacing the information to be hidden with an existentially quantified type variable, but this sort of hiding is all-or-nothing. Using existential types alone, data can be given an abstract view, but cannot be given *multiple* abstract views without making copies. Intersection types allow greater control over information hiding by making it possible for data to be given multiple different views simultaneously. In other words, intersection types allow data to be placed in the intersection of two views. This application need not have anything to do with subtyping, and alone it allows the intersection type to enjoy a considerably simpler metatheory.

Acknowledgements

This material is based on work supported in part by AFOSR grant F49620-97-1-0013 and by ARPA/RADC grant F30602-96-1-0317. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

A Typing rules of F_C

$\boxed{\Gamma \vdash \tau \text{ type}}$

$$\frac{FV(\tau) \subseteq \Gamma}{\Gamma \vdash \tau \text{ type}}$$

$\boxed{\Gamma \vdash \tau_1 \leq \tau_2}$

$$\frac{\Gamma \vdash \tau \text{ type}}{\Gamma \vdash \tau \leq \tau} \qquad \frac{\Gamma \vdash \tau_1 \leq \tau_2 \quad \Gamma \vdash \tau_2 \leq \tau_3}{\Gamma \vdash \tau_1 \leq \tau_3} \qquad \frac{\Gamma \vdash \tau \text{ type}}{\Gamma \vdash \tau \leq \texttt{top}} \qquad \frac{\Gamma \vdash \tau_1' \leq \tau_1 \quad \Gamma \vdash \tau_2 \leq \tau_2'}{\Gamma \vdash \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau_i \leq \tau_i' \quad (\text{for } 1 \leq i \leq n) \\ \Gamma \vdash \tau_i \text{ type} \quad (\text{for } n < i \leq n+k)\end{array}}{\Gamma \vdash \{\ell_1 : \tau_1, \ldots, \ell_{n+k} : \tau_{n+k}\} \leq \{\ell_1 : \tau_1', \ldots, \ell_n : \tau_n'\}} \qquad \frac{\Gamma \vdash \tau_1 \leq \tau_1' \quad \Gamma \vdash \tau_2 \leq \tau_2'}{\Gamma \vdash \tau_1 \wedge \tau_2 \leq \tau_1' \wedge \tau_2'}$$

$$\frac{\Gamma[\alpha] \vdash \tau \leq \tau'}{\Gamma \vdash \forall\alpha.\tau \leq \forall\alpha.\tau'} \qquad \frac{\Gamma[\alpha] \vdash \tau \leq \tau'}{\Gamma \vdash \exists\alpha.\tau \leq \exists\alpha.\tau'} \qquad \frac{\Gamma[\alpha] \vdash \tau \leq \tau'}{\Gamma \vdash \mu\alpha.\tau \leq \mu\alpha.\tau'} \left(\begin{array}{l}\alpha \text{ appears only posi-} \\ \text{tively in } \tau \text{ and } \tau'\end{array}\right)$$

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash x : \tau} \; (\Gamma(x) = \tau) \qquad \frac{}{\Gamma \vdash i : \texttt{int}} \qquad \frac{\Gamma[x : \tau] \vdash e : \tau'}{\Gamma \vdash \lambda x{:}\tau.e : \tau \to \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \qquad \frac{\Gamma \vdash e_i : \tau_i \quad (\text{for } 1 \leq i \leq n)}{\Gamma \vdash \{\ell_1 = e_1, \ldots, \ell_n = e_n\} : \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}}$$

$$\frac{\Gamma \vdash e : \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}}{\Gamma \vdash e.\ell_i : \tau_i} \; (1 \leq i \leq n) \qquad \frac{\Gamma[\alpha] \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau} \qquad \frac{\Gamma \vdash e : \forall\alpha.\tau' \quad \Gamma \vdash \tau \text{ type}}{\Gamma \vdash e[\tau] : \tau'[\tau/\alpha]}$$

$$\frac{\Gamma \vdash e_1 : \exists\alpha.\tau \quad \Gamma[\alpha][x : \tau] \vdash e_2 : \tau'}{\Gamma \vdash \text{unpack}\,[\alpha, x] = e_1 \text{ in } e_2 : \tau'} \; (\alpha \notin \tau') \qquad \frac{\Gamma \vdash c : \tau_1 \Rightarrow \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash c\,e : \tau_2} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \leq \tau'}{\Gamma \vdash e : \tau'}$$

$\boxed{\Gamma \vdash c : \tau_1 \Rightarrow \tau_2}$

$$\frac{\Gamma \vdash \tau \text{ type}}{\Gamma \vdash \text{id}[\tau] : \tau \Rightarrow \tau} \qquad \frac{\Gamma \vdash c_1 : \tau_1' \Rightarrow \tau_1 \quad \Gamma \vdash c_2 : \tau_2 \Rightarrow \tau_2'}{\Gamma \vdash c_1 \to c_2 : (\tau_1 \to \tau_2) \Rightarrow (\tau_1' \to \tau_2')}$$

$$\frac{\Gamma \vdash c_i : \tau_i \Rightarrow \tau_i' \quad (\text{for } 1 \leq i \leq n)}{\begin{array}{r}\Gamma \vdash \{\ell_1 : c_1, \ldots, \ell_n : c_n\} : \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \Rightarrow \\ \{\ell_1 : \tau_1', \ldots, \ell_n : \tau_n'\}\end{array}} \qquad \frac{\Gamma \vdash c_1 : \tau_1 \Rightarrow \tau_1' \quad \Gamma \vdash c_2 : \tau_2 \Rightarrow \tau_2'}{\Gamma \vdash c_1 \wedge c_2 : \tau_1 \wedge \tau_2 \Rightarrow \tau_1' \wedge \tau_2'}$$

$$\frac{\Gamma \vdash c_1 : \tau \Rightarrow \tau_1 \quad \Gamma \vdash c_2 : \tau \Rightarrow \tau_2}{\Gamma \vdash \langle c_1, c_2 \rangle : \tau \Rightarrow \tau_1 \wedge \tau_2} \qquad \frac{\Gamma \vdash \tau_1 \wedge \tau_2 \text{ type}}{\Gamma \vdash \pi_i[\tau_1 \wedge \tau_2] : \tau_1 \wedge \tau_2 \Rightarrow \tau_i} \; (i = 1, 2)$$

$$\frac{\Gamma[\alpha] \vdash c : \tau \Rightarrow \tau'}{\Gamma \vdash \forall\alpha.c : \forall\alpha.\tau \Rightarrow \forall\alpha.\tau'} \qquad \frac{\Gamma[\alpha] \vdash c : \tau \Rightarrow \tau'}{\Gamma \vdash \exists\alpha.c : \exists\alpha.\tau \Rightarrow \exists\alpha.\tau'} \qquad \frac{\Gamma \vdash \tau_1 \ \text{type} \quad \Gamma \vdash \exists\alpha.\tau_2 \ \text{type}}{\Gamma \vdash \text{hide}\,\tau_1 \ \text{in} \ \exists\alpha.\tau_2 : \tau_2[\tau_1/\alpha] \Rightarrow \exists\alpha.\tau_2}$$

$$\frac{\Gamma \vdash \mu\alpha.\tau \ \text{type}}{\Gamma \vdash \text{fold}[\mu\alpha.\tau] : \tau[\mu\alpha.\tau/\alpha] \Rightarrow \mu\alpha.\tau} \qquad \frac{\Gamma \vdash \mu\alpha.\tau \ \text{type}}{\Gamma \vdash \text{unfold}[\mu\alpha.\tau] : \mu\alpha.\tau \Rightarrow \tau[\mu\alpha.\tau/\alpha]}$$

$$\frac{\Gamma \vdash c_1 : \tau_2 \Rightarrow \tau_3 \quad \Gamma \vdash c_2 : \tau_1 \Rightarrow \tau_2}{\Gamma \vdash c_1 \circ c_2 : \tau_1 \Rightarrow \tau_3} \qquad \frac{\Gamma \vdash c : \tau_1 \Rightarrow \tau_2 \quad \Gamma \vdash \tau_1' \leq \tau_1 \quad \Gamma \vdash \tau_2 \leq \tau_2'}{\Gamma \vdash c : \tau_1' \Rightarrow \tau_2'}$$

References

[1] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. In *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 296–320. Springer-Verlag, April 1994.

[2] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[3] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *Twenty-Third ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 296–409, St. Petersburg, Florida, January 1996.

[4] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

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

[6] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software*, Sendai, Japan, September 1997.

[7] Peter Canning, William Cook, Walter Hill, John Mitchell, and Walter Olthoff. F-bounded quantification for object-oriented programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[8] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[9] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Seventeenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, January 1990.

[10] Karl Crary. Foundations for the implementation of higher-order subtyping. In *1997 ACM SIGPLAN International Conference on Functional Programming*, pages 125–135, Amsterdam, June 1997.

[11] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. Technical report, Department of Computer Science, Cornell University, 1998.

[12] Allyn Dimock, Robert Muller, Franklyn Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In *1997 ACM SIGPLAN International Conference on Functional Programming*, pages 11–24, Amsterdam, June 1997.

[13] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Co., 1971.

[14] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, January 1995.

[15] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995.

[16] Samuel Kamin. Inheritance in Smalltalk-80: A denotational definition. In *Fifteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 80–87, San Diego, January 1988.

[17] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Twenty-Third ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, Florida, January 1996.

[18] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[19] Greg Morrisett. *Compiling with Types.* PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, December 1995.

[20] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1997.

[21] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Twenty-Fifth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 85–97, San Diego, January 1998. Extended version published as Cornell University technical report TR97-1651.

[22] Benjamin Pierce. Personal communication, 1998.

[23] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, December 1991.

[24] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[25] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, 1974.

[26] Zhong Shao. Flexible representation analysis. In *1997 ACM SIGPLAN International Conference on Functional Programming*, pages 85–98, Amsterdam, June 1997.

[27] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *ACM Conference on Lisp and Functional Programming*, pages 1–11, Orlando, June 1994.