Simple, Efficient Object Encoding using Intersection Types

Karl Crary

January 1999
CMU-CS-99-100

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also published as Fox Memorandum CMU-CS-FOX-99-01

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. 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.

This research was sponsored by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Keywords: Object encodings, intersection types, type theory.

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, a 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 [17] provides the most natural 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 with self application has led to several different proposals for type-theoretic encodings of objects. Recursive record interpretations [9, 11, 8] 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 [6, 22, 16], 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 records 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 the pointer must be extracted whenever methods are invoked, which costs time.

Recovering Self-Application Operationally speaking, these overheads are easily avoided by adopting the self-application semantics, and have been in untyped object systems. In a typed setting, the difficulty has been in assigning types to objects in a manner that makes possible the desired operations of an object calculus, particularly subtyping. In this paper I show that objects interpreted by self-application can be expressed in type theory without any additional overhead. The paramount concern is that the operational behavior of objects be *undisturbed* in any way by the typing machinery that is wrapped around it.

A secondary contribution of the paper is a careful analysis of the operational issues that make the various typing mechanisms necessary. For example, I show that the naive typing of the self-application semantics does not work because it is too permissive, and fails to adequately state the operational behavior of objects. In particular, it allows methods to be applied, not only to self (and self-application dictates), but to any object of the same type. However, that type may be a supertype of the object's 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 devising a type that does express the operational behavior of objects, 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. More generally, the methodology is to use types to *precisely* specify the allowable interfaces to objects; in several circumstances problems will be seen to arise if types are assigned too loosely.

The ambient type theory required for this encoding 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 restricted enough to admit a simple metatheory.

Overview The paper is organized as follows: Section 2 develops the basic ideas of the object encoding and presents the type theory that serves as the target of the encoding. Section 3 extends the encoding to support self types and method update. Section 4 compares the encoding in detail with other type-theoretic object encodings. Concluding remarks appear in Section 5.

In the interest of brevity, this paper assumes basic familiarity with the Girard-Reynolds polymorphic lambda calculus [14, 23], with 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.

2 Basic Developments

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:

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

Suppose `cpt` is a `ColorPoint`. In order for `cpt` to be a member of `Point`, the `getx` field of `cpt` must be typeable as `Point` → `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 need, 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 $o \triangleleft \ell$ and suppose o is a `Point`; then

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

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

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

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

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

I will refer to this encoding as the OEI encoding, for "objects using existential and intersection types," following the terminology of Bruce *et al.* [7]. In the remainder of this paper, 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 to support self types or method update, but in Section 3 I will introduce a similar encoding (called OREI) that is.

2.1 Object Construction

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

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

I will refer to members of O^* as *pre-objects.* 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

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

and so an object of type $\overline{O}$ is 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.

2.2 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 bounded quantification leads to serious difficulties for type checking and semantics [19, 20]. I do not use bounded quantification in this paper, but there are many good reasons 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 [4] and it restricts the possible models of the type theory [21]. 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: If e has type $\mu\alpha.\tau$ then *unfold* e has type $\tau[\mu\alpha.\tau/\alpha]$, but 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, subtyping is useful primarily as a convenience for the programmer, and is not vital in the intermediate languages of a compiler. Therefore, the target language of my object encoding dispenses with subtyping and instead uses explicit retyping coercions.

The target language, called F_C, is formalized in Appendix A. The novelty of F_C lies in its syntactic class of coercions. Coercions are separated out from ordinary terms because coercions are guaranteed to have no run-time effect; they serve only to change the type of a term from one type to another. When the compiler ultimately generates machine code, it may simply drop the coercions. All the typing mechanisms of this paper are performed using coercions, and therefore it is clear that the operational behavior of objects is identical to what it would be in an untyped setting. In particular, no efficiency is sacrificed to achieve strong typing.

In F_C, members of intersection types are introduced by a pair of coercions. If the coercions c_1 and c_2 coerce τ to τ_1 and τ_2, respectively, then the compound coercion $\langle c_1, c_2 \rangle$ coerces τ to $\tau_1 \wedge \tau_2$. Thus, a member of an intersection type is a single term with two different views.[2] The problem with recursive types discussed above is then handled by coercing a recursively typed term with both an identity coercion and an unfold coercion.

3 Further Developments

3.1 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:

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

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:

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

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

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

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

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

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] Dimock *et al.* [13] make use of a similar idea by defining intersections to contain pairs, the components of which are required to be identical when types are erased.

3.2 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 by

$$\begin{array}{llll} m_{\texttt{getx}} & : & \texttt{Point} \to \texttt{int} & = & \lambda o{:}\texttt{Point}^*.\, 12 \\ m_{\texttt{setx}} & : & \texttt{Point} \to \texttt{int} \to \texttt{Point} & = & \lambda o{:}\texttt{Point}^*.\, \lambda x{:}\texttt{int}.\, o \cdot \texttt{getx} \Leftarrow (\lambda o'{:}\texttt{Point}^*.\, x) \\ m_{\texttt{incx}} & : & \texttt{Point} \to \texttt{Point} & = & \lambda o{:}\texttt{Point}^*.\, (o \triangleleft \texttt{setx})(o \triangleleft \texttt{getx} + 1) \\ pt & : & \texttt{Point} & = & \{\!|\, \texttt{getx} = m_{\texttt{getx}}, \texttt{setx} = m_{\texttt{setx}}, \texttt{incx} = m_{\texttt{incx}} \,|\!\} \end{array}$$

where $\{\!|\, \ell_1 = e_1, \ldots, \ell_n = e_n \,|\!\}$ denotes object construction and $o \cdot \ell \Leftarrow e$ denotes the updating of object o by replacing its ℓ method with e.

Method update is implemented simply by installing the new method in the appropriate field:

$$\begin{array}{llll} m'_{\texttt{setx}} & : & \texttt{Point}^* \to \texttt{int} \to \texttt{Point}^* & = & \lambda o{:}\texttt{Point}^*.\, \lambda x{:}\texttt{int}.\, \{\texttt{getx} = \lambda o'{:}\texttt{Point}^*.\, x, \\ & & & & \qquad \texttt{setx} = o.\texttt{setx} \\ & & & & \qquad \texttt{incx} = o.\texttt{incx}\} \end{array}$$

Note that the function $m_{\texttt{setx}}$ takes and returns pre-objects of type $\texttt{Point}^*$, rather than objects. The pre-object result can (and generally would) be coerced to an object of type $\overline{\texttt{Point}}$ as discussed in Section 2.1. However, the argument necessarily must be a pre-object. Terms of object type (that is, $\overline{O}$ as opposed to O^*) may not have methods updated. This is clear from an inspection of the object type $\overline{O}$, but there is also a simple operational reason: Once in object type, it is impossible to determine the object's original width; any update could drop methods on which other methods depend.

Consequently, the object calculus being compiled must distinguish between two sorts of object type: "actual" object types and pre-object types. Pre-objects may have methods updated and objects may not. However, pre-objects must be promoted to become objects before any subtyping may be used.

3.3 Dedicated Update Methods

Unlike the encoding presented here, the object encoding of Abadi *et al.* supports updating of methods, even after subtyping is used. It is able to do so by adding dedicated update methods to objects. For each ordinary method in an object, their encoding adds a second "update method" that serves only to update its corresponding method. In essence, update methods make method update possible after subtyping because they can remember the record's original width.

Update methods may be used in the OREI encoding as well. I show here how they can be written in the object calculus by hand, but the encoding could add them automatically (as in Abadi *et al.*) as well. Let the type of pre-objects be written $!\{\!|\, \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \,|\!\}$ as α. Suppose a method `foo` has type τ, where τ does not make use of the self type variable. We may make `foo` updatable after subtyping by adding an extra method `update_foo` resulting in an object of type $\{\!|\, \texttt{foo} : \tau, \texttt{update\_foo} : (\alpha \to \tau) \to \alpha, \ldots \,|\!\}$ as α. The method `update_foo` is implemented by:

$$\begin{array}{lll} \texttt{update\_foo} & = & \lambda o{:}\texttt{PreFooType}.\, \lambda f{:}(\texttt{FooType} \to \tau).\, o \cdot \texttt{foo} \Leftarrow f \\ \text{where } \texttt{FooType} & = & \{\!|\, \texttt{foo} : \tau, \texttt{update\_foo} : (\alpha \to \tau) \to \alpha, \ldots \,|\!\} \text{ as } \alpha \\ \text{and } \texttt{PreFooType} & = & !\{\!|\, \texttt{foo} : \tau, \texttt{update\_foo} : (\alpha \to \tau) \to \alpha, \ldots \,|\!\} \text{ as } \alpha \end{array}$$

This simple encoding will serve to update many methods, but it depends essentially on the fact that τ does not use the self type variable α. If α appeared free in τ, then it would appear negatively in the type of the `update_foo` method, and that negative appearance would prevent subtyping from working properly.

Operationally, the problem is that arguments to the `update_foo` method (themselves functions) that return the self type might create entirely new objects, instead of producing their results using their own self arguments. Those new objects could then be missing fields expected by other methods of the present object.

We can resolve this problem by adding bounded quantification to the system, and using it to require that the arguments to an update method produce their results *uniformly,* that is, only by using their self arguments. Consider the type of `update_foo` in:

$$\{\!|\, \texttt{foo} : \tau, \texttt{update\_foo} : (\forall \beta {\leq} \alpha.\, \beta \to \tau[\beta/\alpha]) \to \alpha, \ldots \,|\!\} \text{ as } \alpha$$

Since β is abstract, a prospective new method (*i.e.* an argument to `update_foo`) cannot create an object of type β from scratch, it must construct it using its self argument (of type β) and the operations on that self argument available by virtue of being a subtype of α. In other words, the new method must produce self-typed results using only its self argument.

With this type, subtyping is permitted because α appears only positively. (A similar device is used by Abadi and Cardelli [2].) Note that in order to write any interesting uniform functions, the object calculus must support structural rules for method invocation [3], where, for example, if e has type $\beta \leq \{\!|\, \texttt{foo} : \alpha, \ldots \,|\!\}$ as α, then $e \lhd \texttt{foo}$ has type β, rather than the looser type $\{\!|\, \texttt{foo} : \alpha, \ldots \,|\!\}$ as α.[3]

3.4 Formalization

The discussion so far gives an informal account of the OREI object encoding. We make all this precise by defining a object calculus and giving a type directed translation from that object calculus into the target language F_C.

The syntax for the object calculus appears in Figure 1. To review the notation, $\{\!|\, \ell_i : \tau_i{}^{[i=1 \ldots n]} \,|\!\}$ as α and $!\{\!|\, \ell_i : \tau_i{}^{[i=1 \ldots n]} \,|\!\}$ as α represents the types of object and pre-objects, respectively. Pre-object types are subtypes of object types. The term $\{\!|\, \ell_i = e_i{}^{[i=1 \ldots n]} \,|\!\}$ creates a pre-object (and therefore an object, by subtyping), the term $e \lhd \ell$ invokes method ℓ of object e, and $e_1 \cdot \ell \Leftarrow e_2$ updates method ℓ of pre-object e_1 by e_2. The remaining notation is standard.

[3] Alternatively, we can obviate the need for structural rules and still write uniform update methods by adding higher-order bounded quantification and quantifying over type *operators* that are pointwise subtypes of the operator corresponding to α [22].

$$
\begin{array}{llll}
\text{types} & \tau & ::= & \alpha \mid \texttt{int} \mid \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau \mid \\
 & & & \{\!|\, \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \,|\!\}\ \mathrm{as}\ \alpha \mid\ !\{\!|\, \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \,|\!\}\ \mathrm{as}\ \alpha \\
\text{terms} & e & ::= & x \mid i \mid \{\ell_1 = e_1, \ldots, \ell_n = e_n\} \mid e.\ell \mid \lambda x{:}\tau.e \mid e_1 e_2 \mid \Lambda\alpha.e \mid \\
 & & & e[\tau] \mid \{\!|\, \ell_1 = e_1, \ldots, \ell_n = e_n \,|\!\} \mid e \triangleleft \ell \mid e_1 \cdot \ell \leftleftarrows e_2 \\
\text{contexts} & \Gamma & ::= & \epsilon \mid \Gamma[\alpha] \mid \Gamma[x : \tau]
\end{array}
$$

Figure 1: Source Syntax

Types are translated from the source to target language as discussed above:

$$
\begin{array}{rcl}
|\alpha| & = & \alpha \\
|\texttt{int}| & = & \texttt{int} \\
|\{\ell_i : \tau_i{}^{[i=1\ldots n]}\}| & = & \{\ell_i : |\tau_i|^{[i=1\ldots n]}\} \\
|\tau_1 \to \tau_2| & = & |\tau_1| \to |\tau_2| \\
|\forall\alpha.\tau| & = & \forall\alpha.|\tau| \\
|\{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\}\ \mathrm{as}\ \alpha| & = & \mu\alpha.\, \exists\beta.\, \beta \wedge \{\ell_i : \beta \to |\tau_i|^{[i=1\ldots n]}\} \qquad (\text{where } \beta \text{ is not free in } \tau_i) \\
|!\{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\}\ \mathrm{as}\ \alpha| & = & \mu\alpha.\, \{\ell_i : (\alpha \to |\tau_i|[\,|\{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\}\ \mathrm{as}\ \alpha|/\alpha])^{[i=1\ldots n]}\}
\end{array}
$$

In the interest of brevity, we present the static semantics of the source language and the translation into the target language simultaneously. Appendix B gives rules governing three judgements:

1. $\Gamma \vdash_S \tau$ type indicates that τ is a well-formed source type (in context Γ).

2. $\Gamma \vdash_S e : \tau \leadsto e'$ indicates that source term e has type τ, and that e' is its translation into the target language.

3. $\Gamma \vdash_S \tau_1 \leq \tau_2 \leadsto c$ indicates that the source type τ_1 is a subtype of the source type τ_2, and that c is a target language coercion witnessing that subtyping.

The source language judgements $\Gamma \vdash_S e : \tau$ and $\Gamma \vdash_S \tau_1 \leq \tau_2$ have the obvious meanings, and their rules are obtained by dropping the $\leadsto e$ and $\leadsto c$ suffixes from the translation rules.

With the translation formalized, the natural type correctness result is easy to show:

Proposition 3.1 *Let the context encoding* $|\Gamma|$ *be defined by:*

$$
\begin{array}{rcl}
|\epsilon| & = & \epsilon \\
|\Gamma[\alpha]| & = & |\Gamma|[\alpha] \\
|\Gamma[x : \tau]| & = & |\Gamma|[x : |\tau|]
\end{array}
$$

Then:

1. *If* $\Gamma \vdash_S \tau$ type *then* $|\Gamma| \vdash_T |\tau|$ type.

2. *If* $\Gamma \vdash_S e : \tau \leadsto e'$ *then* $|\Gamma| \vdash_T e' : |\tau|$.

Source	$\{\!\mid \ell_i : \tau_i(\alpha)^{[i=1\ldots n]} \mid\!\}$ as α
OE	$\exists\beta.\quad \beta \times \{\ell_i : \beta \to \tau_i(\beta)^{[i=1\ldots n]}\}$
ORE	$\mu\alpha.\ \exists\beta.\quad \beta \times \{\ell_i : \beta \to \tau_i(\alpha)^{[i=1\ldots n]}\}$
ORBE	$\mu\alpha.\ \exists\beta{\leq}\alpha.\ \beta \times \{\ell_i : \beta \to \tau_i(\beta)^{[i=1\ldots n]}\}$
OREI	$\mu\alpha.\ \exists\beta.\quad \beta \wedge \{\ell_i : \beta \to \tau_i(\alpha)^{[i=1\ldots n]}\}$

Figure 2: Encodings of Object Types

3. *If* $\Gamma \vdash_S \tau_1 \leq \tau_2 \leadsto c$ *then* $|\Gamma| \vdash_T c : |\tau_1| \Rightarrow |\tau_2|$.

With a formalized operational semantics, dynamic correctness of the translation can also be shown, using a straightforward (and uninteresting) simulation argument.

In the interest of simplicity, the source language does not support bounded quantification. As we have seen, bounded quantification is not necessary for many basic object-oriented features. However, there are many excellent reasons to support bounded quantification. For example, as discussed in the previous section, bounded quantification (in conjunction with structural rules or higher-order type constructors) makes it possible to write dedicated update methods for methods using self type. It is not difficult to extend the translation to support bounded quantification using a variant of the Penn interpretation [5, 12].

4 Comparisons

The OREI encoding discussed in this paper is closely related to three other abstraction-oriented object encodings: the OE encoding of Pierce, Turner and Hoffman [22, 16], the ORE encoding of Bruce [6], and the ORBE encoding of Abadi, *et al.* discussed previously. Bruce *et al.* [7] cast each of these encodings in a common framework, and explore the interrelations between them.

The OREI encoding and each of the three encodings from Bruce *et al.* encode object *types* in ways that appear fairly similar. Figure 2 summarizes the encodings of object types. However, these syntactic similarities mask significant differences in the operational behavior of the various encodings. In this section, I review the discussions from Bruce *et al.* and show how OREI fits into the picture.

The four abstraction-oriented object encodings can be arranged in order of the degree to which they specify the form that the "state" of an object must take, (as shown in Figure 3):

1. The simplest of the four is OE, which views an object as a pair, consisting of a state having an arbitrary type and a collection of methods operating on states. Methods that functionally update an object do so by returning a new state. Since that new state is only part of the complete object, the caller has the responsibility of repackaging it to form a complete object by pairing it with its methods and existentially sealing the pair.

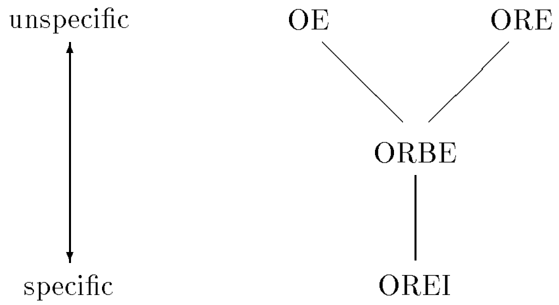

Figure 3: Object Encoding Relationships

2. The ORE encoding is like the OE encoding, except that it moves the burden of repackaging the state from the caller to the method. Consequently, method types must mention the full type of the object, and accordingly objects are given recursive types. Despite this difference, the type of the state itself in the ORE in still completely unspecified, as in the OE encoding.

3. The ORBE encoding unifies the OE and ORE encodings by requiring that an object's state be an object itself, thereby eliminating the distinction between returning a state and returning an object. This requirement is imposed in the type by using bounded quantification to indicate that the type of the state is a subtype of the full object's type.

4. The OREI encoding goes one step further than the ORBE encoding, specifying that the object's state is not just any object, but is the selfsame object itself. This makes OREI the most specific of the four, entirely specifying the object's state.

A shallow examination of the type encodings would suggest that OREI bears the greatest resemblance to the ORE encoding, since they differ only in the type operator used to join the methods, product or intersection. However, the preceding discussion reveals that the similarity is deceptive; operationally the two encodings are very different. For example, object states in ORE are entirely unspecified, while object states in OREI are entirely specified.

Operationally, OREI is most similar to the ORBE encoding. Certainly ORBE is closest in the specificity spectrum, but more importantly, ORBE is closest in expressiveness, such as the ability to support method update. The principal difference between the two is the one discussed above: ORBE's type allows its state to be any object. Consequently, even though an object's state will be the object itself in common usage (at least in a noncoercive interpretation of subtyping), the possibility that it could be another object makes it impossible to take advantage of that fact. As discussed in the introduction, this means that the object must use an extra word to store the state pointer, and for every method invocation must perform and extra dereference to obtain that state pointer.

4.1 Closure Conversion

A variant of the Abadi, *et al.* encoding proposed by Viswanathan [24, 2] hearkens back to the recursive record encodings [9, 11, 8] by hiding the state within method functions, but uses dedicated update methods to support method udpate. At high-level phases of a compiler, the recursive record encodings and Viswanathan's encoding appear to eliminate the extra state pointer, but a consideration of function closures reveals that this is not so.

A function having free variables is implemented by transforming it into a closure, which is a pair the first component of which is an environment containing the function's free variables, and the second component of which is the function's code (abstracted over the environment). Hiding the extra state pointer with a method function merely places it within the environment, where it still uses an word and an extra dereference is still required to obtain it.

Moreover, an obvious optimization to perform after closure conversion is to merge methods' environments into the object, thereby eliminating an extra level of indirection. By appending methods' free variables to the end of an object, we can (in most cases) eliminate the need to allocate closures for methods. The extra fields can then be forgotten using subtyping. With such an optimization in play, the extra state pointer previously hidden within the function now appears in the object again.

4.2 Full Abstraction

The main purpose to Viswanathan's encoding is as a step on the way to a fully abstract object encoding, a property not enjoyed by ORBE. Full abstraction in compilation is not only of theoretical interest; in systems where programmers may write code in lower-level intermediate languages, it is desirable that abstraction properties in the source language be protected in the lower-level intermediate languages as well [1]. An interesting question, then, is whether the OREI encoding is fully abstract.

A formal proof is left as future work, but we may conjecture that OREI is fully abstract for the object calculus without pre-objects, but not with pre-objects. This is based on the following observations: We expect that the encoding will be fully abstract if no "useful" operations can be performed on encoded terms, that cannot be performed on the original terms.

- The action of the encoding on integers, records and functions is trivial, so clearly no additional actions are made possible in those cases. The interesting cases are objects and pre-objects.
- The sole operation available on objects is method invocation. We wish to argue that nothing can be done with an encoded object but invoke methods. An encoded object provides two things (actually, one thing viewable two ways): a member of an abstract type β, and a collection of functions with domain β. A member of abstract type by itself is useless. The functions, on the other hand, can be called, but only by providing a member of β as an argument. The object itself is the only available member of β, and that function call is precisely what is meant by method invocation.
- The operations available on pre-objects are method update and method invocation. However, in an encoded object, the method functions can be extracted, and from there many things are possible. Therefore, for the encoding to be fully abstract, it must be possible to extract a method function from an unencoded pre-objects. Following Viswanathan [24], we can very nearly create a function with identical behavior to a method function. That function takes

a new pre-object argument, updates all the methods of the old pre-object with the new pre-object's methods, then invokes the method in question and returns its result.

$$out(o \cdot \ell) \stackrel{\text{def}}{=} \lambda x.\, ((o \cdot \ell_1 \leftleftharpoons out(x \cdot \ell_1)) \cdot \ell_2 \leftleftharpoons out(x \cdot \ell_2) \cdots) \triangleleft \ell$$

The problem is whether to update the method being extracted. If it is not updated, the object created within *out* does not have quite the right behavior. If it *is* updated, then the field of interest is obliterated.

It appears that this problem could be solved using the same device as Viswanathan [24], to alter the type of pre-objects so that methods cannot depend on their own field. With such a change, the function *out* works as intended, and the modified encoding appears to be fully abstract.

It is worth pointing out that, although pre-objects provide additional functionality over the object calculus used by Viswanathan (*i.e.,* non-uniform method update), the (conjectured) full abstraction of OREI minus pre-objects is not a comparable result to Viswanathan's because of a difference in the treatment of method update in the source calculi. In the OREI object calculus, dedicated update methods are just ordinary methods written to perform method update; such fields can be filled with other non-updating methods so long as the types are the same. In contrast, in Viswanathan's calculus, dedicated update methods are built in and only do update, and consequently a fully abstract encoding must ensure that such methods actually do update. Preventing spurious update methods is the primary issue Viswanathan's encoding addresses.

5 Conclusion

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.

Unlike other object encodings that use intersection types [10, 15], the OREI encoding makes no use of the usual subtyping behavior of intersection types, that $\tau \wedge \tau' \leq \tau$ (in the F_C formalism that $\pi_1 : \tau \wedge \tau' \Rightarrow \tau$). What is used by the OREI encoding is the import of intersection types for controlled information hiding. Existential types are used to hide type information by replacing the information to be hidden by 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.

Although the axiomatization of F_C allows the intersection type to enjoy a considerably simpler metatheory, one should not conclude that F_C's intersection type is the same as a product type. One can give F_C a semantics in which coercions are ordinary functions, and in such a semantics the intersection and product types are indeed the same. However, the preferred semantics is a type-erasure operational semantics in which coercions are merely retyping operators with no run-time effect whatsoever. In that semantics, intersection and product types are clearly different. Moreover, it is in that semantics that the efficiency goals of this work are realized. Fortunately, it is also that semantics that is most easily implemented by a compiler.

References

[1] Martín Abadi. Protection in programming-language translations. In *Twenty-Fifth International Colloquium on Automata, Languages, and Programming*, July 1998.

[2] Martín Abadi and Luca Cardelli. *A Theory of Objects*, chapter 18.3.5. 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] Roberto Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

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

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

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

[8] 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.

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

[10] Adriana B. Compagnoni and Benjamin C. Pierce. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, October 1996.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Jason J. Hickey. A semantics of objects in type theory. Unpublished manuscript, 1997.

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

[17] 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.

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

[20] Benjamin C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, April 1997.

[21] Benjamin C. Pierce. Personal communication, 1998.

[22] 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.

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

[24] Ramesh Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Thirteenth IEEE Symposium on Logic in Computer Science*, Indianapolis, June 1998.

A The Coercion Calculus

$$
\begin{array}{llll}
\text{types} & \tau & ::= & \alpha \mid \mathtt{int} \mid \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau \mid \exists\alpha.\tau \mid \\
 & & & \mu\alpha.\tau \mid \tau_1 \wedge \tau_2 \\
\text{terms} & e & ::= & x \mid i \mid \{\ell_1 = e_1, \ldots, \ell_n = e_n\} \mid e.\ell \mid \lambda x{:}\tau.e \mid e_1 e_2 \mid \\
 & & & \Lambda\alpha.e \mid \mathit{unpack}\,[\alpha, x] = e_1 \mathit{\ in\ } e_2 \mid c\, e \\
\text{coercions} & c & ::= & \mathit{id}[\tau] \mid c_1 \circ c_2 \mid \{\ell_1 : c_1, \ldots, \ell_n : c_n\} \mid c_1 \to c_2 \mid \forall\alpha.c \mid \\
 & & & \exists\alpha.c \mid \mu\alpha.c \mid \mathit{apply}\,\tau_1 \mathit{\ in\ } \forall\alpha.\tau_2 \mid \mathit{hide}\,\tau_1 \mathit{\ in\ } \exists\alpha.\tau_2 \mid \\
 & & & \mathit{fold}[\mu\alpha.\tau] \mid \mathit{unfold}[\mu\alpha.\tau] \mid \langle c_1, c_2\rangle \mid \pi_i[\tau_1 \wedge \tau_2] \mid \\
 & & & \mathit{drop}\,\{\ell_1 : \tau_1, \ldots, \ell_m : \tau_m \mid \ell_{m+1} : \tau_{m+1}, \ldots, \ell_n : \tau_n\} \\
\text{contexts} & \Gamma & ::= & \epsilon \mid \Gamma[\alpha] \mid \Gamma[x : \tau]
\end{array}
$$

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

$$\frac{}{\Gamma \vdash_T \alpha \ \text{type}} \ (\alpha \in \Gamma) \qquad \frac{}{\Gamma \vdash_T \texttt{int} \ \text{type}}$$

$$\frac{\Gamma \vdash_T \tau_i \ \text{type} \quad (\text{for } 1 \leq i \leq n)}{\Gamma \vdash_T \{\ell_i : \tau_i{}^{[i=1\ldots n]}\} \ \text{type}} \qquad \frac{\Gamma \vdash_T \tau_1 \ \text{type} \quad \Gamma \vdash_T \tau_2 \ \text{type}}{\Gamma \vdash_T \tau_1 \to \tau_2 \ \text{type}}$$

$$\frac{\Gamma[\alpha] \vdash_T \tau \ \text{type}}{\Gamma \vdash_T \forall \alpha.\tau \ \text{type}} \ (\alpha \notin \Gamma) \qquad \frac{\Gamma[\alpha] \vdash_T \tau \ \text{type}}{\Gamma \vdash_T \exists \alpha.\tau \ \text{type}} \ (\alpha \notin \Gamma)$$

$$\frac{\Gamma[\alpha] \vdash_T \tau \ \text{type}}{\Gamma \vdash_T \mu\alpha.\tau \ \text{type}} \ \left(\begin{array}{l}\alpha \notin \Gamma, \ \alpha \text{ appears} \\ \text{only positively in } \tau\end{array}\right) \qquad \frac{\Gamma \vdash_T \tau_1 \ \text{type} \quad \Gamma \vdash_T \tau_2 \ \text{type}}{\Gamma \vdash_T \tau_1 \wedge \tau_2 \ \text{type}}$$

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

$$\frac{}{\Gamma \vdash_T x : \tau} \ (\Gamma(x) = \tau) \qquad \frac{}{\Gamma \vdash_T i : \texttt{int}}$$

$$\frac{\Gamma \vdash_T e_i : \tau_i \quad (\text{for } i \leq i \leq n)}{\Gamma \vdash_T \{\ell_i = e_i{}^{[i=1\ldots n]}\} : \{\ell_i : \tau_i{}^{[i=1\ldots n]}\}} \qquad \frac{\Gamma \vdash_T e : \{\ell_i : \tau_i{}^{[i=1\ldots n]}\}}{\Gamma \vdash_T e.\ell_j : \tau_j} \ (1 \leq j \leq n)$$

$$\frac{\Gamma[x : \tau_1] \vdash_T e : \tau_2 \quad \Gamma \vdash_T \tau_1 \ \text{type}}{\Gamma \vdash_T \lambda x{:}\tau_1.e : \tau_1 \to \tau_2} \ (x \notin \Gamma) \qquad \frac{\Gamma \vdash_T e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash_T e_2 : \tau_2}{\Gamma \vdash_T e_1 e_2 : \tau_2}$$

$$\frac{\Gamma[\alpha] \vdash_T e : \tau}{\Gamma \vdash_T \Lambda\alpha.e : \forall\alpha.\tau} \ (\alpha \notin \Gamma) \qquad \frac{\Gamma \vdash_T e_1 : \exists\alpha.\tau \quad \Gamma[\alpha][x : \tau] \vdash_T e_2 : \tau'}{\Gamma \vdash_T \mathit{unpack}\,[\alpha, x] = e_1 \ \mathit{in} \ e_2 : \tau'} \ (\alpha \notin \Gamma, x \notin \Gamma, \alpha \notin \tau')$$

$$\frac{\Gamma \vdash_T c : \tau_1 \Rightarrow \tau_2 \quad \Gamma \vdash_T e : \tau_1}{\Gamma \vdash_T c\, e : \tau_2}$$

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

$$\frac{\Gamma \vdash_T \tau \ \text{type}}{\Gamma \vdash_T \mathit{id}[\tau] : \tau \Rightarrow \tau} \qquad \frac{\Gamma \vdash_T c_1 : \tau_1 \Rightarrow \tau_2 \quad \Gamma \vdash_T c_2 : \tau_2 \Rightarrow \tau_3}{\Gamma \vdash_T c_1 \circ c_2 : \tau_1 \Rightarrow \tau_3}$$

$$\frac{\Gamma \vdash_T c_i : \tau_i \Rightarrow \tau_i' \quad (\text{for } 1 \leq i \leq n)}{\Gamma \vdash_T \{\ell_i : c_i{}^{[i=1\ldots n]}\} : \{\ell_i : \tau_i{}^{[i=1\ldots n]}\} \Rightarrow \{\ell_i : \tau_i'{}^{[i=1\ldots n]}\}} \qquad \frac{\Gamma \vdash_T c_1 : \tau_1' \Rightarrow \tau_1 \quad \Gamma \vdash_T c_2 : \tau_2 \Rightarrow \tau_2'}{\Gamma \vdash_T c_1 \to c_2 : (\tau_1 \to \tau_2) \Rightarrow (\tau_1' \to \tau_2')}$$

$$\frac{\Gamma[\alpha] \vdash_T c : \tau_1 \Rightarrow \tau_2}{\Gamma \vdash_T \forall\alpha.c : \forall\alpha.\tau_1 \Rightarrow \forall\alpha.\tau_2}\ (\alpha \notin \Gamma) \qquad \frac{\Gamma[\alpha] \vdash_T c : \tau_1 \Rightarrow \tau_2}{\Gamma \vdash_T \exists\alpha.c : \exists\alpha.\tau_1 \Rightarrow \exists\alpha.\tau_2}\ (\alpha \notin \Gamma)$$

$$\frac{\Gamma[\alpha] \vdash_T c : \tau_1 \Rightarrow \tau_2}{\Gamma \vdash_T \mu\alpha.c : \mu\alpha.\tau_1 \Rightarrow \mu\alpha.\tau_2}\ \left(\begin{array}{l}\alpha \notin \Gamma,\ \alpha \text{ appears}\\ \text{only positively in } c\end{array}\right)$$

$$\frac{\Gamma \vdash_T \forall\alpha.\tau_2 \text{ type} \quad \Gamma \vdash_T \tau_1 \text{ type}}{\Gamma \vdash_T \mathit{apply}\ \tau_1\ \mathit{in}\ \forall\alpha.\tau_2 : \forall\alpha.\tau_2 \Rightarrow \tau_2[\tau_1/\alpha]}$$

$$\frac{\Gamma \vdash_T \exists\alpha.\tau_2 \text{ type} \quad \Gamma \vdash_T \tau_1 \text{ type}}{\Gamma \vdash_T \mathit{hide}\ \tau_1\ \mathit{in}\ \exists\alpha.\tau_2 : \tau_2[\tau_1/\alpha] \Rightarrow \exists\alpha.\tau_2}$$

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

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

$$\frac{\Gamma \vdash_T \tau_i \text{ type} \quad (\text{for } 1 \le i \le n)}{\Gamma \vdash_T \mathit{drop}\ \{\ell_i : \tau_i{}^{[i=1\ldots m]} \mid \ell_i : \tau_i{}^{[i=m+1\ldots n]}\} : \{\ell_i : \tau_i{}^{[i=1\ldots n]}\} \Rightarrow \{\ell_i : \tau_i{}^{[i=1\ldots m]}\}}\ (m \le n)$$

The rules of F_C do not specify whether records are taken to be ordered or unordered. If records are to be unordered, we use the same typing rules and take records and record types to be syntactically identical to ones with permuted fields. All the results of this paper apply without modification in either version of F_C. However, the efficiency advantages of the self-application semantics are likely to be most telling in the version with ordered fields. If fields are unordered, one must either view records as association lists of labels and data, or adopt a coercive interpretation of subtyping [5]. In either case, the efficiency advantages of self-application will stand, but will likely be dwarfed by the costs of supporting unordered fields.

B The Object Encoding

Static Semantics and Translation Rules

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

$$\frac{\mathit{FreeTypeVariables}(\tau) \subseteq \Gamma}{\Gamma \vdash_S \tau \text{ type}}$$

$\boxed{\Gamma \vdash_S \tau_1 \leq \tau_2 \rightsquigarrow c}$

$$\frac{\Gamma \vdash_S \tau \ \text{type}}{\Gamma \vdash_S \tau \leq \tau \rightsquigarrow id[|\tau|]} \qquad \frac{\Gamma \vdash_S \tau_1 \leq \tau_2 \rightsquigarrow c_1 \quad \Gamma \vdash_S \tau_2 \leq \tau_3 \rightsquigarrow c_2}{\Gamma \vdash_S \tau_1 \leq \tau_3 \rightsquigarrow c_2 \circ c_1}$$

$$\frac{\Gamma \vdash_S \tau_i \leq \tau_i' \rightsquigarrow c_i \quad (\text{for } 1 \leq i \leq n) \quad \Gamma \vdash_S \tau_i \ \text{type} \quad (\text{for } n < i \leq m)}{\begin{array}{l}\Gamma \vdash_S \{\ell_i : \tau_i{}^{[i=1\ldots m]}\} \leq \{\ell_i : \tau_i'{}^{[i=1\ldots n]}\} \rightsquigarrow \\ \qquad \{\ell_i : c_i{}^{[i=1\ldots n]}\} \circ \mathit{drop}\, \{\ell_i : |\tau_i|^{[i=1\ldots n]} \mid \ell_i : |\tau_i|^{[i=n+1\ldots m]}\}\end{array}} \ (m \geq n)$$

$$\frac{\Gamma \vdash_S \tau_1' \leq \tau_1 \rightsquigarrow c_1 \quad \Gamma \vdash_S \tau_2 \leq \tau_2' \rightsquigarrow c_2}{\Gamma \vdash_S \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' \rightsquigarrow c_1 \to c_2} \qquad \frac{\Gamma[\alpha] \vdash_S \tau_1 \leq \tau_2 \rightsquigarrow c}{\Gamma \vdash_S \forall\alpha.\tau_1 \leq \forall\alpha.\tau_2 \rightsquigarrow \forall\alpha.c} \ (\alpha \notin \Gamma)$$

$$\frac{\Gamma[\alpha] \vdash_S \tau_i \leq \tau_i' \rightsquigarrow c_i \quad (\text{for } 1 \leq i \leq n) \quad \Gamma[\alpha] \vdash_S \tau_i \ \text{type} \quad (\text{for } n < i \leq m)}{\begin{array}{l}\Gamma \vdash_S \{\!|\, \ell_i : \tau_i{}^{[i=1\ldots m]} \,|\!\}\ \text{as}\ \alpha \leq \{\!|\, \ell_i : \tau_i'{}^{[i=1\ldots n]} \,|\!\}\ \text{as}\ \alpha \rightsquigarrow \\ \quad \mu\alpha.\, \exists\beta.\, id[\beta] \wedge (\{\ell_i : id[\beta] \to c_i{}^{[i=1\ldots n]}\} \circ \\ \qquad\qquad \mathit{drop}\, \{\ell_i : \beta \to |\tau_i|^{[i=1\ldots n]} \mid \\ \qquad\qquad\quad \ell_i : \beta \to |\tau_i|^{[i=n+1\ldots m]}\})\end{array}} \ \begin{pmatrix}\alpha \notin \Gamma, \beta \notin \tau_i, m \geq n, \\ \alpha \text{ appears only} \\ \text{positively in } \tau_i\end{pmatrix}$$

$$\frac{\Gamma[\alpha] \vdash_S \tau_i \ \text{type} \quad (\text{for } 1 \leq i \leq n)}{\begin{array}{l}\Gamma \vdash_S !\{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\}\ \text{as}\ \alpha \leq \{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\}\ \text{as}\ \alpha \rightsquigarrow \\ \qquad \mathit{fold}[\overline{\tau}] \circ \\ \qquad \mathit{hide}\ \tau^*\ \mathit{in}\ \exists\beta.\, \beta \wedge \{\ell_i : \beta \to |\tau_i|[\overline{\tau}/\alpha]^{[i=1\ldots n]}\} \circ \\ \qquad \langle id[\tau^*], \mathit{unfold}[\tau^*]\rangle \\ \text{where } \overline{\tau} = |\{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\}\ \text{as}\ \alpha| \\ \text{and } \tau^* = |!\{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\}\ \text{as}\ \alpha|\end{array}} \ (\alpha \notin \Gamma, \beta \notin \tau_i)$$

$\boxed{\Gamma \vdash_S e : \tau \rightsquigarrow e'}$

$$\frac{\Gamma \vdash_S e : \tau_1 \rightsquigarrow e' \quad \Gamma \vdash_S \tau_1 \leq \tau_2 \rightsquigarrow c}{\Gamma \vdash_S e : \tau_2 \rightsquigarrow c\, e'} \qquad \frac{}{\Gamma \vdash_S x : \tau \rightsquigarrow x} \ (\Gamma(x) = \tau) \qquad \frac{}{\Gamma \vdash_S i : \texttt{int} \rightsquigarrow i}$$

$$\frac{\Gamma \vdash_S e_i : \tau_i \rightsquigarrow e_i' \quad (\text{for } 1 \leq i \leq n)}{\Gamma \vdash_S \{\ell_i = e_i{}^{[i=1\ldots n]}\} : \{\ell_i : \tau_i{}^{[i=1\ldots n]}\} \rightsquigarrow \{\ell_i = e_i'{}^{[i=1\ldots n]}\}}$$

$$\frac{\Gamma \vdash_S e : \{\ell_i : \tau_i{}^{[i=1\ldots n]}\} \rightsquigarrow e'}{\Gamma \vdash_S e.\ell_j : \tau_j \rightsquigarrow e'.\ell_j} \ (1 \leq j \leq n)$$

$$\frac{\Gamma[x:\tau_1] \vdash_S e : \tau_2 \leadsto e' \quad \Gamma \vdash_S \tau_1 \ \text{type}}{\Gamma \vdash_S \lambda x{:}\tau_1.e : \tau_1 \to \tau_2 \leadsto \lambda x{:}|\tau_1|.e'} \ (x \notin \Gamma) \qquad \frac{\Gamma \vdash_S e_1 : \tau_1 \to \tau_2 \leadsto e_1' \quad \Gamma \vdash_S e_2 : \tau_2 \leadsto e_2'}{\Gamma \vdash_S e_1 e_2 : \tau_2 \leadsto e_1' e_2'}$$

$$\frac{\Gamma[\alpha] \vdash_S e : \tau \leadsto e'}{\Gamma \vdash_S \Lambda\alpha.e : \forall\alpha.\tau \leadsto \Lambda\alpha.e'} \ (\alpha \notin \Gamma) \qquad \frac{\Gamma \vdash_S e : \forall\alpha.\tau' \leadsto e' \quad \Gamma \vdash_S \tau \ \text{type}}{\Gamma \vdash_S e[\tau] : \tau'[\tau/\alpha] \leadsto e'[|\tau|]}$$

$$\frac{\Gamma[\alpha] \vdash_S e_i : !\{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\} \text{ as } \alpha \to \tau_i[(\{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\} \text{ as } \alpha)/\alpha] \leadsto e_i'}{\begin{array}{l}\Gamma \vdash_S \{\!|\, \ell_i = e_i{}^{[i=1\ldots n]} \,|\!\} : !\{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\} \text{ as } \alpha \leadsto \\ \qquad (\mathit{fold}[|!\{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\} \text{ as } \alpha|])\{\ell = e_i'{}^{[i=1\ldots n]}\}\end{array}} \ (\alpha \notin \Gamma, 1 \leq i \leq n)$$

$$\frac{\Gamma \vdash_S e : \{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\} \text{ as } \alpha \leadsto e'}{\begin{array}{l}\Gamma \vdash_S e \lhd \ell_i : \tau_i[(\{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\} \text{ as } \alpha)/\alpha] \leadsto \\ \qquad \mathit{unpack}\ [\beta, x] = \mathit{unfold}[\overline{\tau}]\ e'\ \mathit{in}\ (\pi_2[\tau']\,x).\ell\,(\pi_1[\tau']\,x) \\ \text{where } \tau' = \beta \wedge \{\ell_i : \beta \to |\tau_i|[\overline{\tau}/\alpha]^{[i=1\ldots n]}\} \\ \text{and } \overline{\tau} = |\{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\} \text{ as } \alpha|\end{array}} \ (1 \leq i \leq n, \beta \notin \tau_i)$$

$$\frac{\begin{array}{l}\Gamma \vdash_S e_1 : !\{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\} \text{ as } \alpha \leadsto e_1' \\ \Gamma \vdash_S e_2 : !\{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\} \text{ as } \alpha \to \tau_j[(\{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\} \text{ as } \alpha)/\alpha] \leadsto e_2'\end{array}}{\begin{array}{l}\Gamma \vdash_S e_1 \cdot \ell_j \leftleftarrows e_2 : !\{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\} \text{ as } \alpha \leadsto \\ \qquad \mathit{let}\ \ x = \mathit{unfold}[|!\{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\} \text{ as } \alpha|]\ e_1' \\ \qquad \mathit{in} \\ \qquad \mathit{fold}[|!\{\!|\, \ell_i : \tau_i{}^{[i=1\ldots n]} \,|\!\} \text{ as } \alpha|] \\ \qquad\qquad \{\ell_i = x.\ell_i{}^{[i=1\ldots j-1]}, \ell_j = e_2', \ell_i = x.\ell_i{}^{[i=j+1\ldots n]}\}\end{array}} \ (1 \leq j \leq n, x \notin e_1)$$