An Efficient Class and Object Encoding *

Neal Glew
Department of Computer Science
Cornell University
aglew@acm.org

ABSTRACT

An object encoding translates a language with object primitives to one without. Similarly, a class encoding translates classes into other primitives. Both are important theoretically for comparing the expressive power of languages and for transferring results from traditional languages to those with objects and classes. Both are also important foundations for the implementation of object-oriented languages as compilers typically include a phase that performs these translations.

This paper describes a language with a primitive notion of classes and objects and presents an encoding of this language into one with records and functions. The encoding uses two techniques often used in compilers for single-inheritance class-based object-oriented languages: the self-application semantics and the method-table technique. To type the output of the encoding, the encoding uses a new formulation of self quantifiers that is more powerful than previous approaches.

1. INTRODUCTION

An object encoding is a translation from a language with a primitive notion of objects to one without, typically one that has functions and records instead. Object encodings are important theoretically for comparing the expressive power of object-oriented languages versus functional languages and for transferring results proven about functional languages to object-oriented languages. Object encodings are also important for building solid foundations for the implementation of object-oriented languages as known implementation techniques typically involve a phase that turns objects into records and functions.

*This paper is based on work supported in part by the NSF grant CCR-9708915, AFOSR grant F49620-97-1-0013, and ARPA/RADC grant F30602-1-0317. 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.

OOPSLA 2000, Minneapolis, MN, USA

A typical implementation of an object-oriented language uses a translation that corresponds to the *self-application semantics* [Kam88] (this paper uses the term self-application semantics to refer to both the implementation technique and the semantics). In the self-application semantics, an object becomes a record with entries for the object's fields and methods. Methods become functions which take an extra argument, and the object itself is always passed as this extra argument—because part of the object is applied to the object itself, the terminology "self application" is used.

Similarly, a class encoding is a translation from a language with a primitive notion of classes into one without. A class encoding might target a pure object language or might be combined with an object encoding to produce records and functions. Class encodings are important for the same reason as object encodings: they illuminate the additional expressiveness of classes and provide foundations for the implementation of class-based languages.

A typical implementation of a class-based language uses the *method-table technique* in addition to the self-application semantics. In this refinement, objects become records with entries for the object's fields and an entry for a method table, which is another record with entries for the object's methods. Only one method table is constructed per class and is shared amongst all the class's instances.

It is important to distinguish between typed and untyped encodings. An untyped encoding translates into a language without a static typing discipline. A typed encoding translates a statically typed language into a statically typed language and must, if given type-correct input, produce type-correct output. Typed encodings provide foundations for type-directed compilers, which in addition to translating code also maintain and translate type information. There are several benefits to type-directed compilers: They can be debugged by turning on intermediate language type checkers (*e.g.*, [MTC+96]). They can use type information to guide or enable optimisation or better run-time systems (*e.g.*, [TMC+96]). They can produce certificates of safety for target code [NL98, MCG+99]. Typed object and class encodings describe how to produce the necessary type information for an intermediate language based on records and functions.

The search for object and class encodings is also an attempt to tease apart the various aspects of objects: packaging of code and data, self reference, information hiding, *et cetera*. Teasing these aspects apart breaks the object construct into a number of separate concepts. Many of these concepts match well with traditional records, functions, and

type systems. Indeed, the self-application semantics and method-tables are easily expressed as untyped object encodings. But one concept does not: the type of `self`, and this mismatch makes it difficult to extend these encodings to produce type correct output in some typed language. Typed encodings must somehow capture the type of `self` using existing type constructors or by inventing new ones.

The extensive previous work on typed object and class encodings, is discussed in detail with references in Section 5. These encodings have used a combination of recursive types and some flavour of existential types to capture the type of `self`. Recursive types are used because objects can refer to themselves. Existential types are used to try to capture the information hiding aspect of objects. Unfortunately, these combinations of recursive and existential types do not quite capture `self`'s type, forcing these encodings to include work arounds: extra data structures, extra indirections, and extra operations. For theoretical purposes, these work arounds are no problem, and these encodings answer important theoretical questions such as the comparative expressiveness of object-oriented languages versus functional languages and how to transfer results about functional languages to object-oriented languages. But for practical purposes, implementations based directly on these encodings are not efficient, and so these encodings cannot be considered foundations for language implementation.

This paper tackles the type of `self` directly by introducing a new type constructor, the *self quantifier*, to capture it. To explain what a self quantifier does, consider some object and the static types assigned to it at various program points. In general the static types will not be the actual run-time type of the object, but rather a supertype of the run-time type. Self quantifiers allow the supertype to refer to the actual, but unknown, run-time type of the object. This paper uses self quantifiers to devise a new typed object and class encoding that is faithful to the self-application semantics and method-table techniques. In particular, methods become functions which require an argument of the actual run-time type, and self quantifiers are used to capture this requirement.

The main contribution of this paper is a new typed object and class encoding based on a new formulation of self quantifiers. The key to understanding object encodings is the typing of `self`, and self quantifiers naturally capture this type. The new formulation of self quantifiers avoids certain problems with previous typed encodings and allows the self-application semantics and method-table techniques to be used as the term translation. Since these techniques are used in real compilers, the new encoding provides a formal foundation for implementations. A minor contribution of this paper is a new object language with primitive notions of classes and objects.

1.1 Roles of Classes

In typical class-based object-oriented languages, classes play a number of roles. Primarily, a class provides a template for the creation of objects, specifying which fields the objects will have and how they will respond to methods. But classes play other roles as well. For example, in Java [GJS96] a class provides a template, a named type, an explicit subtyping relationship between this named type and the named type of the superclass, constructors that allow for the controlled creation and initialisation of objects, and the ability to downcast objects based on the class from which they were created. This paper just concentrates on the role of classes as templates for objects. The role of classes in downcasting is addressed in another paper [Gle99b]. The other roles, while important, are beyond the scope of this paper and are left to future work.

```
class Window {
  Rectangle extent;
  boolean handleEvent(Event) {···};
  boolean contains(Point) {···};
}
class ContainerWindow extends Window {
  Window[] children;
  boolean handleEvent(Event) {···};
  void addChild(Window) {···};
}
```

Figure 1: Example Class Hierarchy

2. OBJECT TEMPLATE LANGUAGE

To begin, this section defines a language with primitive notions of objects and object templates. Object templates capture the role of classes as templates for objects. Consider the example class hierarchy shown in Figure 1 that includes classes from a hypothetical GUI toolkit. The class `Window` represents windows on the user's screen and the class `ContainerWindow` represents windows that are the composition of a number of child windows. Class `Window` has a field containing the current position and size of the window and methods for handling user events and for determining if a pixel is within the window. Class `ContainerWindow` has a field containing the current children and a new method for adding children. Additionally, it overrides `handleEvent` perhaps to distribute the event to one of children according to screen location.

Class `Window` provides a template for objects with a field `extent` and methods `handleEvent` and `contains`. Similarly, `ContainerWindow` provides a template for objects with fields `extent` and `children` and methods `handleEvent`, `contains`, and `addChild`. One way to construct these templates is to start with the superclass's template and then apply operations that add fields, add methods, and override methods. Since adding a field is not dependent upon the other fields or methods, adding a single field could be a basic operation. Adding or overriding methods, however, is not independent, because the new methods may refer to other methods being added. Therefore, the addition and overriding of several methods is the most basic operation. Using et to denote the empty template, $+$ to denote field addition, and $\leftarrow\!\!+$ to denote method addition and overriding, templates for `Window` and `ContainerWindow` are constructed as follows:

$$\begin{array}{l}\mathsf{let}\ \mathtt{Window}_t = \\ \quad \mathsf{et} + \mathtt{extent} : \mathtt{Rectangle} \\ \qquad \leftarrow\!\!+ [\mathtt{handleEvent} = \cdots, \mathtt{contains} = \cdots]\ \mathsf{in} \\ \mathsf{let}\ \mathtt{ContainerWindow}_t = \\ \quad \mathtt{Window}_t + \mathtt{children} : \mathsf{array}(\mathtt{Window}) \\ \qquad\qquad \leftarrow\!\!+ [\mathtt{handleEvent} = \cdots, \mathtt{addChild} = \cdots]\ \mathsf{in}\end{array}$$

Objects are created by instantiating classes, or more precisely, their templates. The new object will have all the fields and methods in the template and will respond to the

methods according to the code given in the template. The instantiation operation must provide initial values for the fields as the template only lists the fields and their types. (Most object-oriented languages provide more sophisticated creation and initialisation mechanisms; these mechanisms are beyond the scope of this paper.) Writing instantiation as new *template*[*field* = *initial value*], objects from `Window` and `ContainerWindow` are created as follows:

$$\begin{array}{l}\mathsf{let}\ w1 = \mathsf{new}\ \mathtt{Window_t}[\mathtt{extent} = r1]\ \mathsf{in}\\ \mathsf{let}\ w2 = \mathsf{new}\ \mathtt{ContainerWindow_t}\\ \qquad\qquad [\mathtt{extent} = r2, \mathtt{children} = \mathsf{array}()]\ \mathsf{in}\end{array}$$

where $r1$ and $r2$ are rectangles.

Objects are manipulated using the operations of method invocation, field selection, and field update. For example, the operation $w2.\mathtt{addChild}(w1)$ adds $w1$ as a child of $w2$.

The language informally described so far captures the key features of a single-inheritance class-based language such as Java. The rest of this section formalises an object template language called O, and the next section shows how to encode it into a language with records and functions. The term language of O is:

$$\begin{array}{llll}\text{Expressions} & e & ::= & x \mid \\ & & & \mathsf{et} \mid e + f : \sigma \mid e \leftarrowtail [m_i = M_i]_{i\in I} \mid \\ & & & \mathsf{new}\ e[f_j = e_j]_{j\in J} \mid \\ & & & e.m \mid e.f \mid e_1.f := e_2 \\ \text{Methods} & M & ::= & x.e{:}\tau\end{array}$$

Metavariable x ranges over term variables, m over method names, and f over field names.

In O, templates are values distinct from the objects that are created from them, and are built from the empty template by the operations of field addition and method addition/override. The empty template is written et; its instances have no fields and no methods. The operation $e+f : \sigma$ adds field f with type σ to template e producing a new template. The template e must not have field f, and the new template has all the methods, method implementations, and fields of e, as well as field f with type σ. The operation $e \leftarrowtail [m_i = M_i]_{i\in I}$ adds or overrides methods m_i of template e with implementations M_i, producing a new template. The new template will have methods m_i with implementation M_i plus all the methods in e not in $\{m_i | i \in I\}$, as well as all fields of e. A method implementation M has the form $x.e{:}\tau$. Unlike Java, O does not have an explicit keyword for `self`, but instead the programmer chooses a variable x and uses this variable to refer to `self` in the method body. In $x.e{:}\tau$, x is the variable chosen to stand for `self`, e is the method body, and τ is the return type. If object o has $x.e{:}\tau$ as its implementation of method m then the method invocation $o.m$ will result in the execution of e with x replaced by o. Note that methods are parameterless, Section 4 describes extensions of O with method parameters.

Objects are created by instantiating templates and can be manipulated by method invocation, field selection, and field update. Instantiation is written $\mathsf{new}\ e[f_j = e_j]_{j\in J}$, where e is the template to be instantiated and e_j is the initial value of field f_j. The new object will respond to methods as dictated by template e. Method invocation is written $e.m$, field selection $e.f$, and field update $e_1.f := e_2$.

The type language of O is:

$$\begin{array}{llll}\text{Types} & \tau, \sigma & ::= & \mathsf{tempt}\ r \mid \mathsf{objt}\ r \\ \text{Rows} & r & ::= & [m_i : s_i; f_j : \sigma_j]_{i\in I, j\in J} \\ \text{Signatures} & s & ::= & \tau\end{array}$$

As templates are distinct from objects, they have their own types, which are different from the types for objects. Templates are given the type $\mathsf{tempt}\ r$ where r, called a row, describes the objects that result from instantiating the template. The row $[m_i{:}s_i; f_j{:}\sigma_j]_{i\in I, j\in J}$ describes objects with methods m_i of signature s_i and fields f_j of type σ_j. The order of the methods and fields matters,[1] so I and J can be thought of as ordered index sets. Methods in O are parameterless and just compute a result, so a signature is a type, the type of the result. Section 4 describes extensions of O with more complicated signatures. There is no interesting subtyping for template types as the operations on templates have conflicting subtyping requirements.

Objects have type $\mathsf{objt}\ r$ where r is a row as above. Object types have right-extension breadth subtyping: an object type with more methods on the right end and more fields on the right end is a subtype of an object type with less. Objects types also have depth subtyping for methods: methods of the subtype may have subsignatures of the methods of the supertype. Because fields are mutable, they have no depth subtyping. The following rule captures these properties:

$$\frac{k \in I_2 : \quad \vdash_{\mathcal{O}} s_k \le s'_k}{\vdash_{\mathcal{O}} \mathsf{objt}[m_i{:}s_i; f_j{:}\sigma_j]_{i\in I_1, j\in J_1} \le \mathsf{objt}[m_i{:}s'_i; f_j{:}\sigma_j]_{i\in I_2, j\in J_2}}$$

where I_2 is a prefix of I_1 and J_2 is a prefix of J_1.

Type checking terms is separated into two judgements: one for expressions and one for methods. (This separation along with the distinct syntactic classes for signatures and methods will facilitate later extensions of O with method arguments, type arguments, and self types.) Judgement $\Gamma \vdash^{\mathsf{M}}_{\mathcal{O}} M : \tau \rhd s$ asserts that method implementation M has signature s when `self` has type τ or one of its subtypes.

Type checking the template operations is fairly straightforward. The empty template has the empty template type $\mathsf{tempt}[;]$. The operation $e+f{:}\sigma$ requires e to have a template type without field f, and the result type is the same template type but with f added. Method addition/override is more complicated. This operation is type checked by first computing the row for the new template, then checking that the method implementations are correct assuming `self` has the new row, and finally checking that any overridden methods have compatible signatures, that is, the overriding method has a subsignature of the overridden method. Formally, the typing rule is:

$$\frac{\begin{array}{ll}& \Gamma \vdash_{\mathcal{O}} e : \mathsf{tempt}[m_i{:}s_i; f_j{:}\sigma_j]_{i\in I, j\in J} \\ k \in K : & \Gamma \vdash^{\mathsf{M}}_{\mathcal{O}} M_k : \mathsf{objt}\ r' \rhd s'_k \\ k \in I \cap K : & \vdash_{\mathcal{O}} s'_k \le s_k\end{array}}{\Gamma \vdash_{\mathcal{O}} e \leftarrowtail [m_i = M_i]_{i\in K} : \mathsf{tempt}\ r'}$$

where $r' = [m_i{:}s''_i; f_j{:}\sigma_j]_{i\in(I, K-I), j\in J}$, $s''_i = s_i$ if $i \in I - K$, and $s''_i = s'_i$ if $i \in K$.

Instantiation $\mathsf{new}\ e[f_j = e_j]_{j\in J}$ requires e to have a template type, $\{f_j | j \in J\}$ to be exactly the fields in e's type,

[1] The encoding is also correct if the order of methods and fields is unimportant, so long as the order of record fields in the target language is unimportant.

and e_j to have the type of field f_j:

$$\frac{\Gamma \vdash_{\mathcal{O}} e : \mathsf{tempt}\ r \quad \Gamma \vdash_{\mathcal{O}} e_j : \sigma_j}{\Gamma \vdash_{\mathcal{O}} \mathsf{new}\ e[f_j = e_j]_{j\in J} : \mathsf{objt}\ r}\ (r = [m_i{:}s_i; f_j{:}\sigma_j]_{i\in I, j\in J})$$

Method invocation $e.m$ requires e to have an object type with method m and the result type is m's signature. Field selection $e.f$ requires e to have an object type with field f and the result type is f's type. Field update $e_1.f := e_2$ requires e_1 to have an object type with field f, e_2 to have f's type, and the result type is e_1's type.

Appendix A gives an operational semantics and a full set of typing rules for O. O could be given either a functional or an imperative semantics, and so long as the target language in the next section has the same kind of semantics, the encoding in the next section will be correct. Since a functional semantics is more concise, it is used in this paper. The typing rules are sound with respect to the operational semantics [Gle00a].

3. ENCODING O

This section presents a typed encoding of O into a language with records and functions, using the self-application semantics and the method-table technique. Before getting into formal details, this part of the section informally spells out the self-application semantics and method-table technique in more detail and discusses the issues that arise in trying to type them. These typing issues motivate a new type constructor, the self quantifier, as well as other features needed in the target language. Section 3.1 formalises the target language and Section 3.2 formalises the encoding.

This section informally describes the encoding, and sometimes uses the the formal translation syntax to refer to other parts of the encoding. Unfortunately, it is necessary to refer to some parts of encoding before they are defined. To ease the burden, here is a summary of the parts of the encoding, the formal translation syntax, and their intended meanings.

$[\![\tau]\!]_{\text{type}}$	The translation of type τ
$[\![r]\!]_{\text{mt}}(\tau)$	The record type of the method table of a translated object where τ is the type of `self`
$[\![r]\!]_{\text{full}}(\tau)$	The record type of the translation of an object where τ is the type of `self`
$[\![s]\!]_{\text{sig}}(\tau)$	The translation of signature s
$[\![e]\!]_{\text{exp}}$	The translation of expression e
$[\![M]\!]_{\text{mth}}(\tau)$	The translation of method body M where τ is the type of `self`

Two of these deserve a little elaboration. Objects are translated into records, one of whose fields is the method table, also a record. The types $[\![r]\!]_{\text{full}}(\tau)$ and $[\![r]\!]_{\text{mt}}(\tau)$ are the respective record types for these records where τ is the type of `self`. The translation of object and template types use these types, but also have quantifiers to introduce `self`'s type as described later.

Under the self-application semantics, a method is compiled into a function taking an extra argument, and during method invocation the object itself is always passed as the extra argument. Thus, the method `handleEvent` in the class `Window` is compiled to a function of the following form, named say `Window::handleEvent`:

$$\lambda(x{:}\alpha, y{:}\texttt{Event}).b$$

where x is the extra self parameter, b is the body of the method, and α is, for now, a type variable that stands for the type of self. This function has type $(\alpha, \texttt{Event}) \to \mathsf{bool}$.

In a class-based language, all instances of a class have the same methods. In order to save space, objects share a structure with other instances of the class, the method table. A method table is a record with one field for each method the object responds to. For example, the `Window` class has a method table, named say `Window::mt`:

$$\langle \texttt{handleEvent} = \texttt{Window::handleEvent},\ \texttt{contains} = \texttt{Window::contains}\rangle$$

and `ContainerWindow` has method table:

$$\langle \texttt{handleEvent} = \texttt{ContainerWindow::handleEvent},\ \texttt{contains} = \texttt{Window::contains},\ \texttt{addChild} = \texttt{ContainerWindow::addChild}\rangle$$

Using the suggested typing of `Window::handleEvent`, the method table `Window::mt` has type:

$$[\![\texttt{Window}]\!]_{\text{mt}}(\alpha) \quad = \quad \langle \texttt{handleEvent}{:}(\alpha, \texttt{Event}) \to \mathsf{bool},\ \texttt{contains}{:}(\alpha, \texttt{Point}) \to \mathsf{bool}\rangle \quad (1)$$

However, this type has a free α, and the encoding must somehow introduce this α. Abadi and Cardelli [AC96] observe that the methods in these method tables are polymorphic in the final object type, and so can be given an F-bounded polymorphic type [CCH+89]. Using the $[\![r]\!]_{\text{full}}(\alpha)$ type, `Window`'s method table gets type:

$$\langle \texttt{handleEvent} : \forall\alpha \le [\![\texttt{Window}]\!]_{\text{full}}(\alpha).(\alpha, \texttt{Event}) \to \mathsf{bool},\ \texttt{contains} : \forall\alpha \le [\![\texttt{Window}]\!]_{\text{full}}(\alpha).(\alpha, \texttt{Point}) \to \mathsf{bool}\rangle$$

My encoding will use this idea with one twist. Instead of polymorphic methods, the method table itself is polymorphic. Thus `Window`'s method table has type:

$$\forall\alpha \le [\![\texttt{Window}]\!]_{\text{full}}(\alpha).\langle \texttt{handleEvent} : (\alpha, \texttt{Event}) \to \mathsf{bool},\ \texttt{contains} : (\alpha, \texttt{Point}) \to \mathsf{bool}\rangle$$

This means that a method table can be installed into an object simply by instantiating it at an appropriate type. In general $[\![\mathsf{temp}\ r]\!]_{\text{type}} = \forall\alpha \le [\![r]\!]_{\text{full}}(\alpha)\ .\ [\![r]\!]_{\text{mt}}(\alpha)$ and $[\![r]\!]_{\text{mt}}(\alpha)$ is a record type with one entry for each method in r, which is a function taking an α to the result of that method:

$$[\![[m_i{:}\alpha.\tau_i; f_j{:}\sigma_j]_{i\in I, j\in I}]\!]_{\text{mt}}(\alpha) = \langle m_i{:}\alpha \to [\![\tau_i]\!]_{\text{type}}\rangle_{i\in I}$$

An object is a record with an entry for its class's method table and an entry for each of its fields. For example, instances of `Window` and `ContainerWindow` would have the forms

$$\langle \mathsf{mt}{=}\texttt{Window::mt}, \texttt{extent}{=}r_1\rangle$$
$$\langle \mathsf{mt}{=}\texttt{ContainerWindow::mt}, \texttt{extent}{=}r_2, \texttt{children}{=}a\rangle$$

respectively, where r_1 and r_2 are some rectangles and a is some array of `Window`s.

The type of an instance of `Window` has the form:

$$[\![\texttt{Window}]\!]_{\text{full}}(\alpha) = \langle \mathsf{mt} : [\![\texttt{Window}]\!]_{\text{mt}}(\alpha),\ \texttt{extent} : \texttt{Rectangle}\rangle$$

Again the issue is how to introduce α. Naively, this is the object type itself, so a recursive type should be used:

$$\mathsf{rec}\ \alpha.\langle \mathsf{mt} : [\![\texttt{Window}]\!]_{\text{mt}}(\alpha), \texttt{extent} : \texttt{Rectangle}\rangle \quad (2)$$

Unfortunately this does not work for the following reason. Consider a `ContainerWindow` instance, which also has type `Window`, it would have the following target type:

$$\mathsf{rec}\ \alpha.\langle \mathsf{mt} : [\![\texttt{ContainerWindow}]\!]_{\mathrm{mt}}(\alpha), \texttt{extent} : \texttt{Rectangle}, \texttt{children} : \mathsf{array}(\texttt{Window})\rangle$$

Since `ContainerWindow` is a subtype of `Window`, its translation must be a subtype of `Window`'s translation. This means that the above type should be a subtype of (2), but it is not. Type (2)'s body has a contravariant occurance of α (see (1)), so has no subtypes other than itself.

The problem is that the recursive type makes α, the type of `self`, equal to `Window` instead of the actual run-time type of the object. The solution is to somehow make α refer to this actual run-time type. To achieve this, I introduce a new[2] type constructor, a self quantifier, instead of the recursive quantifier.

A self quantifier allows a type to refer to the actual run-time type of the value inhabiting it. The type $\mathsf{self}\ \alpha.\tau$ contains values v of type τ where α is the actual type of v. Using this quantifier, `Window`'s instances have type:

$$\mathsf{self}\ \alpha.\langle \mathsf{mt} : [\![\texttt{Window}]\!]_{\mathrm{mt}}(\alpha), \texttt{extent} : \texttt{Rectangle}\rangle$$

In general $[\![\mathsf{obj}\ r]\!]_{\mathrm{type}} = \mathsf{self}\ \alpha.[\![r]\!]_{\mathrm{full}}(\alpha)$ and $[\![r]\!]_{\mathrm{full}}(\alpha)$ is a record type with an entry for the method table and an entry for each field of r:

$$[\![r]\!]_{\mathrm{full}}(\alpha) = \langle \mathsf{mt}{:}[\![r]\!]_{\mathrm{mt}}(\alpha), f_j{:}[\![\sigma_j]\!]_{\mathrm{type}}\rangle_{j\in J} \quad \text{where} \quad r = [m_i{:}\tau_i; f_j{:}\sigma_j]_{i\in I, j\in I}$$

The only remaining issue is formalising self quantifiers. Abadi and Cardelli [AC96] provide a formulation of self quantifiers, but their formulation leads to the need for "recoup" fields and the inefficiencies of an extra field and an extra projection.[3] The encoding needs a new formulation of self quantifiers that avoids the problems of recoup fields.

Abadi and Cardelli's formulation involves two operations: one to introduce self quantifiers and one to eliminate them. The introduction form is $\mathsf{pack}\ e, \sigma\ \mathsf{as}\ \mathsf{self}\ \alpha.\tau$ (they call it "wrap"), and it produces an expression e packaged up with its actual self type σ. The typing rule is:

$$\frac{\Delta; B; \Gamma \vdash_{\mathcal{F}} e : \sigma \quad \Delta; B \vdash_{\mathcal{F}} \sigma \le \tau\{\alpha := \sigma\}}{\Delta; B; \Gamma \vdash_{\mathcal{F}} \mathsf{pack}\ e, \sigma\ \mathsf{as}\ \mathsf{self}\ \alpha.\tau : \mathsf{self}\ \alpha.\tau}$$

where capture avoiding substitution of x for y in z is written $z\{y := x\}$. For σ to actually be e's self type, e must have type σ. In addition e also must have type τ with α replaced by e's self type, that is, e must have type $\tau\{\alpha := \sigma\}$. The latter is achieved by requiring that $\sigma \le \tau\{\alpha := \sigma\}$. Using `pack`, the translation of $\mathsf{new}\ \texttt{Window}_{\mathsf{t}}[\texttt{extent} = r1]$ is:

$$\mathsf{pack}\ \langle \mathsf{mt} = \texttt{Window} :: \mathsf{mt}, \mathsf{extend} = [\![r1]\!]_{\mathrm{exp}}\rangle, \mathsf{rec}\ \alpha.[\![\texttt{Window}]\!]_{\mathrm{full}}(\alpha)\ \mathsf{as}\ \mathsf{self}\ \alpha.[\![\texttt{Window}]\!]_{\mathrm{full}}(\alpha)$$

Here, the object's run-time is $\mathsf{rec}\ \alpha.[\![\texttt{Window}]\!]_{\mathrm{full}}(\alpha)$. For this type to satisfy the requirements for packing into a self type, the condition $\Delta; B \vdash_{\mathcal{F}} \sigma \le \tau\{\alpha := \sigma\}$ above, it must be the case that $\mathsf{rec}\ \alpha.\tau \le \tau\{\alpha := \mathsf{rec}\ \alpha.\tau\}$. This is true under an equirecursive interpretation of recursive types, that is, where $\mathsf{rec}\ \alpha.\tau = \tau\{\alpha := \mathsf{rec}\ \alpha.\tau\}$. The target language has this interpretation.

The elimination form is $\mathsf{unpack}\ \alpha, x = e_1\ \mathsf{in}\ e_2$ (they call it "use as"). Intuitively, the expression e_1 is a value packaged with its self type, and `unpack` unpacks the value into x and the self type into α, and executes e_2. The typing rule is:

$$\frac{\Delta; B; \Gamma \vdash_{\mathcal{F}} e_1 : \mathsf{self}\ \alpha.\tau_1 \quad \Delta, \alpha; B, \alpha \le \mathsf{self}\ \alpha.\tau_1; \Gamma, x : \tau_1 \vdash_{\mathcal{F}} e_2 : \tau_2 \quad \Delta \vdash_{\mathcal{F}} \tau_2}{\Delta; B; \Gamma \vdash_{\mathcal{F}} \mathsf{unpack}\ \alpha, x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

Notice that x is assumed to have type τ_1, but will be bound to a value whose actual run-time type could be a strict subtype of τ_1. For this reason, this unpack typing rule is too weak to type check method invocation. Consider the method invocation $e.m$ where e has type $\mathsf{obj}\ r$ and m has signature τ in r. It is translated into $\mathsf{unpack}\ \alpha, x = [\![e_1]\!]_{\mathrm{exp}}\ \mathsf{in}\ x.\mathsf{mt}.m\ x$. Under the above rule, $x.\mathsf{mt}.m$ has type $\alpha \to [\![\tau]\!]_{\mathrm{type}}$, but x has type $[\![r]\!]_{\mathrm{full}}(\alpha)$, which is a strict supertype of α.

The solution is to make a stronger assumption about x—that it has type α. This is sound because α is bound to the actual run-time type of the value bound to x. This stronger assumption leads to the rule:

$$\frac{\Delta; B; \Gamma \vdash_{\mathcal{F}} e_1 : \mathsf{self}\ \alpha.\tau_1 \quad \Delta, \alpha; B, \alpha \le \tau_1; \Gamma, x : \alpha \vdash_{\mathcal{F}} e_2 : \tau_2 \quad \Delta \vdash_{\mathcal{F}} \tau_2}{\Delta; B; \Gamma \vdash_{\mathcal{F}} \mathsf{unpack}\ \alpha, x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

Note, however, that the bound $\alpha \le \tau_1$ has α on both the left and the right sides, that is, it is an F bound rather than an ordinary bound. Since the system must already deal with F-bounded polymorphism, these F bounds add no additional complexity. In fact, this use of F bounds brings a nice symmetry to the system, as F bounds are used in the typing of methods, and F bounds are used in the typing of method invocation.

Using this new rule, reconsider the translation of $e.m$:

$$\mathsf{unpack}\ \alpha, x = [\![e]\!]_{\mathrm{exp}}\ \mathsf{in}\ x.\mathsf{mt}.m\ x$$

During type checking of $x.\mathsf{mt}.m\ x$, x has type α and α has bound $[\![r]\!]_{\mathrm{full}}(\alpha)$. Thus $x.\mathsf{mt}.m$ type checks and has type $\alpha \to [\![\tau]\!]_{\mathrm{type}}$. Since x has type α, the application type checks.

Finally consider method addition/override. The translation of this operation needs to create a new method table that is a combination of an old method table and some new method implementations. There are two approaches: create a new record and copy the relevant entries from the old record, or have record operations for updating a field and for extending a record with a new field. Either approach would work, but I have chosen the second approach. Overridden methods translate into a record update operation that needs to produce a new record. Field update also translates into a record update operation. If the source language has applicative field update then this operation needs to produce a new record. If the source language has imperative field update then this operation needs to update in place. Even if the source language has applicative field update, the typing requirements for the translation of field update and the translation of method override are different and require different record update operations. Therefore, the target language

[2] Strictly speaking, self quantifiers were introduced by Abadi and Cardelli [AC96], but, as explained in this section, their formulation of self quantifiers is insufficient for the purposes of this paper, so a new formulation is needed.

[3] Note that the translation of objects given in this section under the interpretation of self quantifiers used by Abadi and Cardelli leads directly to Abadi, Cardelli, and Viswanathan's encoding for an imperative calculus.

has record extension and two record update operations in addition to projection and record formation.

Most class-based object-oriented language, unlike O, do not have first-class templates. In these languages, a method table is completely determined at compile time (link time in dynamic languages like Java) and the method tables can be built by the compiler and included in the static-data segment. In O this amounts to statically reducing template expressions to template values, which can then be translated into statically determined records and functions. Since O has first-class values, the translation presented in this paper treats the more general case.

3.1 Target Language

The target language, $\mathcal{F}_{\mathsf{self}}$, is a variant of the second-order typed lambda calculus with records, F-bounded polymorphism, self quantifiers, and recursive types. The syntax is:

$$
\begin{array}{lllll}
\text{Types} & \tau, \sigma & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \langle \ell_i : \tau_i^{\phi_i} \rangle_{i \in I}^{\varphi} \mid \\
 & & & \forall \alpha \leq \tau_1 . \tau_2 \mid \mathsf{self}\ \alpha.\tau \mid \mathsf{rec}\ \alpha.\tau \\
\text{Variances} & \phi & ::= & + \mid \circ \\
 & \varphi & ::= & \circ \mid \to \\
\text{Expressions} & e & ::= & x \mid \lambda x : \tau . e \mid e_1\ e_2 \mid \\
 & & & \langle \ell_i = e_i \rangle_{i \in I} \mid e.\ell \mid e_1.\ell \leftarrow e_2 \mid \\
 & & & e_1.\ell := e_2 \mid e_1 + \ell = e_2 \mid \\
 & & & \Lambda \alpha \leq \tau . e \mid e[\tau] \mid \\
 & & & \mathsf{pack}\ e, \tau\ \mathsf{as\ self}\ \alpha.\sigma \mid \\
 & & & \mathsf{unpack}\ \alpha, x = e_1\ \mathsf{in}\ e_2
\end{array}
$$

The unusual features of $\mathcal{F}_{\mathsf{self}}$ are its records, F-bounded polymorphism, self quantifiers, and equirecursive types. $\mathcal{F}_{\mathsf{self}}$ contains an extensive set of record operations including projection, update, and extension. In order to have all these operations as well as breadth and depth subtyping, which is necessary for the encoding, record types must have a number of variances to keep everything straight. A record type $\langle \ell_i : \tau_i^{\phi_i} \rangle_{i \in I}^{\varphi}$ contains records with fields ℓ_i of type τ_i. The variance ϕ_i specifies the allowable operations on that field, $+$ means projection only and $\circ$ allows both projection and update. The record variance φ specifies whether the type lists all of the fields of the value or just some of them. A record is in the type $\langle \ell_i : \tau_i^{\phi_i} \rangle_{i \in I}^{\circ}$ only when it has exactly the fields $\ell_{i \in I}$, but is in the type $\langle \ell_i : \tau_i^{\phi_i} \rangle_{i \in I}^{\to}$ when it has at least the fields $\ell_{i \in I}$ and possibly more.

There are two record update operations. The operation $e_1.\ell := e_2$ can be interpreted imperatively or applicatively depending upon whether the source language has an imperative or applicative field update. Its typing rule is structural [AC96] (see also [HP98]), that is, the type of the result is the same as the type of e_1, which is required to be a subtype of a record type with a field ℓ that is mutable and e_2 must have the type corresponding to ℓ:

$$
\frac{\begin{array}{c}\Delta; B; \Gamma \vdash_{\mathcal{F}} e_1 : \sigma_1 \\ \Delta; B \vdash_{\mathcal{F}} \sigma_1 \leq \langle \ell_i : \tau_i^{\phi_i} \rangle_{i \in I}^{\varphi} \\ \Delta; B; \Gamma \vdash_{\mathcal{F}} e_2 : \tau_k\end{array}}{\Delta; B; \Gamma \vdash_{\mathcal{F}} e_1.\ell_k := e_2 : \sigma_1} \ (k \in I; \phi_k = \circ)
$$

The operation $e_1.\ell \leftarrow e_2$, on the other hand, always produces a new record, which is a copy of e_1 with the ℓ field replaced by e_2. Its typing rule ignores the old type and variance of ℓ and the result type is a record type obtained from e_1's by replacing the field ℓ with the type of e_2:

$$
\frac{\Delta; B; \Gamma \vdash_{\mathcal{F}} e_1 : \langle \ell_i : \tau_i^{\phi_i} \rangle_{i \in I}^{\varphi} \quad \Delta; B; \Gamma \vdash_{\mathcal{F}} e_2 : \sigma}{\Delta; B; \Gamma \vdash_{\mathcal{F}} e_1.\ell_k \leftarrow e_2 : \langle \ell_i : \tau_i'^{\phi_i'} \rangle_{i \in I}^{\varphi}} \ (k \in I)
$$

where $\tau_i'^{\phi_i'} = \tau_i^{\phi_i}$ if $i \neq k$ and $\tau_k'^{\phi_k'} = \sigma^{\circ}$. Finally, the operation $e_1 + \ell = e_2$ adds a new field ℓ with initial value e_2 to record e_1. Record e_1 must not contain a field ℓ, and the typing rule ensures this by using the exact record variance $\circ$:

$$
\frac{\Delta; B; \Gamma \vdash_{\mathcal{F}} e_1 : \langle \ell_i : \tau_i^{\phi_i} \rangle_{i \in I}^{\circ} \quad \Delta; B; \Gamma \vdash_{\mathcal{F}} e_2 : \sigma}{\Delta; B; \Gamma \vdash_{\mathcal{F}} e_1 + \ell = e_2 : \langle \ell_i : \tau_i^{\phi_i}, \ell : \sigma^{\circ} \rangle_{i \in I}^{\circ}} \ (\ell \notin \ell_{i \in I})
$$

Polymorphic types $\forall \alpha \leq \tau_1 . \tau_2$ are F bounded [CCH$^+$89]. This means that α binds in both τ_1 and τ_2, and that a type σ satisfies the bound if σ is a subtype of $\tau_1\{\alpha := \sigma\}$. Otherwise, they follow the standard rules for polymorphic types with the kernel-fun subtyping rule:

$$
\frac{\Delta, \alpha \vdash_{\mathcal{F}} \tau \quad \Delta, \alpha; B, \alpha \leq \tau; \Gamma \vdash_{\mathcal{F}} e : \sigma}{\Delta; B; \Gamma \vdash_{\mathcal{F}} \Lambda \alpha \leq \tau . e : \forall \alpha \leq \tau . \sigma}
$$

$$
\frac{\Delta; B; \Gamma \vdash_{\mathcal{F}} e : \forall \alpha \leq \tau_1 . \tau_2 \quad \Delta; B \vdash_{\mathcal{F}} \sigma \leq \tau_1\{\alpha := \sigma\}}{\Delta; B; \Gamma \vdash_{\mathcal{F}} e[\sigma] : \tau_2\{\alpha := \sigma\}}
$$

Recursive types $\mathsf{rec}\ \alpha.\tau$ contain values that have type τ where α refers to the recursive type $\mathsf{rec}\ \alpha.\tau$. This is formalised by making a recursive type equal to its unrolling:

$$
\frac{}{\Delta \vdash_{\mathcal{F}} \mathsf{rec}\ \alpha.\tau = \tau\{\alpha := \mathsf{rec}\ \alpha.\tau\}}
$$

The most novel aspect of $\mathcal{F}_{\mathsf{self}}$ is its self quantifiers. A self quantified type is written $\mathsf{self}\ \alpha.\tau$, and has a covariant subtyping rule:

$$
\frac{\Delta, \alpha; B \vdash_{\mathcal{F}} \tau \leq \sigma}{\Delta; B \vdash_{\mathcal{F}} \mathsf{self}\ \alpha.\tau \leq \mathsf{self}\ \alpha.\sigma}
$$

Self quantified types are introduced by the pack operation and eliminated by the unpack operation. These operations have the semantics and typing rules discussed above.

Appendix B gives an operational semantics and a full set of typing rules for $\mathcal{F}_{\mathsf{self}}$. The typing rules are sound with respect to the operational semantics [Gle00a]. An alternative formulation of the target language with explicit coercions for recursive types is described in my dissertation [Gle00b].

Equirecursive types and F-bounds make decision procedures for subtyping far from obvious. However, there are algorithms for first and second-order systems with recursive types [AC93, KPS95, CG99]. I believe these results can be extended to $\mathcal{F}_{\mathsf{self}}$, but am still working out the details. Alternatively, a variation of $\mathcal{F}_{\mathsf{self}}$ where recursive types and F-bounds are mediated by explicit coercions [Gle00b] is definitely decidable and practical. Also, the calculus of coercions [Cra99] could also be used to get a decidable version of $\mathcal{F}_{\mathsf{self}}$. $\mathcal{F}_{\mathsf{self}}$ does not have a minimal types property because of the variances on fields. If the introduction form for records were $\langle \ell_i = e_i : \tau_i \rangle_{i \in I}$ with a rule that required e_i to have type τ_i then $\mathcal{F}_{\mathsf{self}}$ would have a minimal types property. The self quantifier developed in this paper could also be used in lower-level typed languages including typed target languages such as TAL [MWCG99, MCG$^+$99].

3.2 The Translation

The translation appears in Figure 2. Technically it is a type-directed translation and is defined by induction over an

$$
\begin{array}{lcl}
[\![\mathsf{tempt}\ r]\!]_{\mathrm{type}} & = & \forall\alpha \leq [\![r]\!]_{\mathrm{full}}(\alpha).[\![r]\!]_{\mathrm{mt}}(\alpha,\circ) \\
[\![\mathsf{objt}\ r]\!]_{\mathrm{type}} & = & \mathsf{self}\ \alpha.[\![r]\!]_{\mathrm{full}}(\alpha) \\
[\![r]\!]_{\mathrm{mt}}(\tau,\varphi) & = & \langle m_i{:}[\![s_i]\!]_{\mathrm{sig}}(\tau)^{+}\rangle^{\varphi}_{i\in I} \\
[\![r]\!]_{\mathrm{full}}(\tau) & = & \langle \mathsf{mt}{:}[\![r]\!]_{\mathrm{mt}}(\tau,\rightarrow)^{+}, f_j{:}[\![\sigma_j]\!]^{\circ}_{\mathrm{type}}\rangle^{\rightarrow}_{j\in J} \\
[\![\sigma]\!]_{\mathrm{sig}}(\tau) & = & \tau \rightarrow \sigma \\
 & & \\
[\![x]\!]_{\mathrm{exp}} & = & x \\
[\![\mathsf{et}]\!]_{\mathrm{exp}} & = & \forall\alpha \leq \langle \mathsf{mt}{:}\langle\rangle^{\rightarrow+}\rangle^{\rightarrow}.\langle\rangle \\
[\![e + f{:}\tau]\!]_{\mathrm{exp}} & = & \forall\alpha \leq [\![r']\!]_{\mathrm{full}}(\alpha).[\![e]\!]_{\mathrm{exp}}[\alpha] \\
 & & \text{where } e \text{ has type } \mathsf{tempt}[m_i{:}s_i; f_j{:}\sigma_j]_{i\in I, j\in J} \text{ and } r' \text{ is } [m_i{:}s_i; f_j{:}\sigma_j, f{:}\tau]_{i\in I, j\in J} \\
[\![e \leftarrow\!\!+ [m_i = M_i]_{i\in K}]\!]_{\mathrm{exp}} & = & \forall\alpha \leq [\![r']\!]_{\mathrm{full}}(\alpha).([\![e]\!]_{\mathrm{exp}}[\alpha].m_i \leftarrow_{i\in I\cap K} [\![M_i]\!]_{\mathrm{mth}}(\alpha) +_{i\in K-I} m_i = [\![M_i]\!]_{\mathrm{mth}}(\alpha)) \\
 & & \text{where } e \text{ has type } \mathsf{tempt}[m_i{:}s_i; f_j{:}\sigma_j]_{i\in I, j\in J},\ r' \text{ is } [m_i{:}s_i'', f_j{:}\sigma_j]_{i\in (I,K-I), j\in J}, \\
 & & \qquad s_i'' = s_i' \text{ if } i \in K, \text{ and } s_i'' = s_i \text{ if } i \notin K \\
[\![\mathsf{new}\ e[f_j = e_j]_{j\in J}]\!]_{\mathrm{exp}} & = & \mathsf{pack}\ \langle \mathsf{mt} = [\![e]\!]_{\mathrm{exp}}[\mathsf{rec}\ \alpha.[\![r]\!]_{\mathrm{full}}(\alpha)], f_j = [\![e_j]\!]_{\mathrm{exp}}\rangle_{j\in J}, \mathsf{rec}\ \alpha.[\![r]\!]_{\mathrm{full}}(\alpha)\ \mathsf{as}\ [\![\mathsf{objt}\ r]\!]_{\mathrm{type}} \\
 & & \text{where } e \text{ has type } \mathsf{tempt}\ r \\
[\![e.m]\!]_{\mathrm{exp}} & = & \mathsf{unpack}\ \alpha, x = [\![e]\!]_{\mathrm{exp}}\ \mathsf{in}\ x.\mathsf{mt}.m\ x \\
 & & \text{where } \alpha \text{ and } x \text{ are fresh} \\
[\![e.f]\!]_{\mathrm{exp}} & = & \mathsf{unpack}\ \alpha, x = [\![e]\!]_{\mathrm{exp}}\ \mathsf{in}\ x.f \\
 & & \text{where } \alpha \text{ and } x \text{ are fresh} \\
[\![e_1.f := e_1]\!]_{\mathrm{exp}} & = & \mathsf{unpack}\ \alpha, x = [\![e_1]\!]_{\mathrm{exp}}\ \mathsf{in}\ \mathsf{pack}\ x.f := [\![e_2]\!]_{\mathrm{exp}}, \alpha\ \mathsf{as}\ [\![\mathsf{objt}\ r]\!]_{\mathrm{type}} \\
 & & \text{where } e_1 \text{ has type } \mathsf{objt}\ r \text{ and } \alpha \text{ and } x \text{ are fresh} \\
[\![x.e{:}\sigma]\!]_{\mathrm{mth}}(\tau) & = & \lambda x'{:}\tau.\mathsf{let}\ x = \mathsf{pack}\ x', \tau\ \mathsf{as}\ \tau'\ \mathsf{in}\ [\![e]\!]_{\mathrm{exp}} \\
 & & \text{where } x.e{:}\sigma \text{ has type } \tau' \rhd \sigma \text{ and } x' \text{ is fresh}
\end{array}
$$

Figure 2: The Translation

O typing derivation. However, I present it as a function of O expression syntax and indicate the typing assumptions. The proof of correctness contains a kind of coherence argument for the translation [Gle00a].

The translation uses the ideas developed in the introduction of this section. For a row r there are two important target types: $[\![r]\!]_{\mathrm{mt}}(\tau,\varphi)$ for method tables, and $[\![r]\!]_{\mathrm{full}}(\tau)$ for the objects. The record type of a method table is $[\![r]\!]_{\mathrm{mt}}(\tau,\varphi)$ where τ is the type of self and φ is the desired record variance. Exact record variance is used for object templates, because adding methods requires using the record extension operation, which requires exact record variance. Extensible record variance is used for objects in order to get depth subtyping. The record type of an object is $[\![r]\!]_{\mathrm{full}}(\tau)$ where τ is the type of self. As discussed, a template type is polymorphic over self, so is translated to $\forall\alpha \leq [\![r]\!]_{\mathrm{full}}(\alpha).[\![r]\!]_{\mathrm{mt}}(\alpha,\circ)$. An object type uses a self quantifier, so is translated to $\mathsf{self}\ \alpha.[\![r]\!]_{\mathrm{full}}(\alpha)$.

At the term level, notice that field extension translates into a term with no operational effect. The effect of the type abstraction and application is to change the type to reflect the new bound on the self type. Method update and addition translates into a series of record extensions and $\leftarrow$ updates. Note that these are done sequentially, and the mutual dependencies in the source language are resolved by the type application $[\![e]\!]_{\mathrm{exp}}[\alpha]$, where the new bound for α has the necessary type information about the other methods. Template instantiation is translated into record formation followed by packing into the appropriate self type. The method table is installed by instantiating it with a recursive type for the actual self type. The method invocation, field selection, and field update operations use `unpack` to open the self quantifier and then apply the appropriate record operations. Note that field update unpacks the object, updates, and then repacks it back into the appropriate self type.

The translation is both type preserving and operationally correct [Gle00a].

4. EXTENSIONS

The previous section presented an object and class encoding for a very simple object template language. This allowed the key ideas to be presented without being cluttered by extraneous source-language features. However, it begs the question of whether the ideas generalise to more advanced object-oriented constructs. In fact, they do and, except for self types, they generalise without requiring any new ideas. This section will demonstrate this by sketching a number of extensions.

It is worth noting that the encoding, as presented, works for either imperative or functional objects so long as the operation $:=$ in $\mathcal{F}_{\mathsf{self}}$ is interpreted in the same way. Also note that any number of nonobject-oriented constructs could be added to the source and target languages and the translation extended to deal with them. Since the target language already has F-bounded polymorphism and recursive types, these could also be added to the source language and translation.

Now consider more object-oriented constructs. Method parameters could be added to O as follows:

$$
\begin{array}{lcl}
s & ::= & (\tau_1, \ldots, \tau_n) \rightarrow \tau \\
e & ::= & \cdots \mid e.m(e_1, \ldots, e_n) \\
M & ::= & x(x_1 : \tau_1, \ldots, x_n : \tau_n).b : \tau
\end{array}
$$

To encode these method parameters $\mathcal{F}_{\mathsf{self}}$ is extended with multiargument functions.[4] Using $(\tau_1, \ldots, \tau_n) \rightarrow \tau$ for mul-

[4]Multiargument functions are theoretically equivalent to curried functions, but the implementations are vastly different, especially if closure conversion is done earlier than object encoding as I have suggested elsewhere [Gle99a].

tiargument function types, $\lambda(x_1 : \tau_1, \ldots, x_n : \tau_n).b$ for multiargument functions, and $e(e_1, \ldots, e_n)$ for multiargument application the revised encoding is:

$$
\begin{array}{l}
[\![(\tau_1, \ldots, \tau_n) \to \tau]\!]_{\mathrm{sig}}(\sigma) = \\
\quad (\sigma, [\![\tau_1]\!]_{\mathrm{type}}, \ldots, [\![\tau_n]\!]_{\mathrm{type}}) \to [\![\tau]\!]_{\mathrm{type}} \\
[\![e.m(e_1, \ldots, e_n)]\!]_{\mathrm{exp}} = \\
\quad \mathsf{unpack}\ \alpha, x = [\![e]\!]_{\mathrm{exp}}\ \mathsf{in} \\
\quad x.\mathsf{mt}.m(x, [\![e_1]\!]_{\mathrm{exp}}, \ldots, [\![e_n]\!]_{\mathrm{exp}}) \\
[\![x(x_1 : \tau_1, \ldots, x_n : \tau_n).b : \tau]\!]_{\mathrm{mth}}(\sigma) = \\
\quad \lambda(x' : \sigma, x_1 : [\![\tau_1]\!]_{\mathrm{type}}, \ldots, x_n : [\![\tau_n]\!]_{\mathrm{type}}). \\
\quad \mathsf{let}\ x = \mathsf{pack}\ x', \sigma\ \mathsf{as}\ \tau'\ \mathsf{in}\ [\![b]\!]_{\mathrm{exp}}
\end{array}
$$

Type parameters can also be incorporated by encoding methods as polymorphic functions:

$$
\begin{array}{lcl}
s & ::= & [\alpha_1 \le \tau_1, \ldots, \alpha_n \le \tau_n]\tau \\
e & ::= & \cdots \mid e.m[\tau_1, \ldots, \tau_n] \\
M & ::= & x[\alpha_1 \le \tau_1, \ldots, \alpha_n \le \tau_n].b : \tau
\end{array}
$$

$$
\begin{array}{l}
[\![[\alpha_1 \le \tau_1, \ldots, \alpha_n \le \tau_n]\tau]\!]_{\mathrm{sig}}(\sigma) = \\
\quad \forall\alpha_1 \le [\![\tau_1]\!]_{\mathrm{type}} \ldots \forall\alpha_n \le [\![\tau_n]\!]_{\mathrm{type}}.\sigma \to \tau \\
[\![e.m[\tau_1, \ldots, \tau_n]]\!]_{\mathrm{exp}} = \\
\quad \mathsf{unpack}\ \alpha, x = [\![e]\!]_{\mathrm{exp}}\ \mathsf{in} \\
\quad x.\mathsf{mt}.m[[\![\tau_1]\!]_{\mathrm{type}}] \cdots [[\![\tau_n]\!]_{\mathrm{type}}]\ x \\
[\![x[\alpha_1 \le \tau_1, \ldots, \alpha_n \le \tau_n].b : \tau]\!]_{\mathrm{mth}}(\sigma) = \\
\quad \Lambda\alpha_1 \le [\![\tau_1]\!]_{\mathrm{type}} \ldots \Lambda\alpha_n \le [\![\tau_n]\!]_{\mathrm{type}}.\lambda x' : \sigma. \\
\quad \mathsf{let}\ x = \mathsf{pack}\ x', \sigma\ \mathsf{as}\ \tau'\ \mathsf{in}\ [\![b]\!]_{\mathrm{exp}}
\end{array}
$$

Covariant self types could be added to O by allowing signatures of the form $\alpha.\tau$ where α may occur only positively in τ. Method bodies would have the form $\alpha, x.b : \tau$ where α is a type variable that binds in b and stands for the type of self. The rule for method invocation would become:

$$
\frac{\Delta; B \vdash_{\mathcal{O}} e : \tau}{\Delta; B \vdash_{\mathcal{O}} e.m_k : \tau_k\{\alpha := \tau\}}
$$

where $\tau = \mathsf{objt}[m_i{:}s_i; f_j{:}\sigma_j]_{i\in I, j\in J}$, $k \in I$, and $s_k = \alpha_k.\tau_k$. To encode this variant of O into $\mathcal{F}_{\mathsf{self}}$, a deeper form of pack is needed. The deeper form of pack is written $\mathsf{pack}\ e, \tau'[\alpha := \tau]\ \mathsf{as\ self}\ \alpha.\sigma$ and coerces an expression of type $\tau'\{\alpha := \tau\}$ into $\tau'\{\alpha := \mathsf{self}\ \alpha.\sigma\}$ when α occurs only positively in τ' and $\tau \le \sigma\{\alpha := \tau\}$. The details are messy, but the essence of the new translation is:

$$
\begin{array}{lcl}
[\![\alpha.\tau]\!]_{\mathrm{sig}}(\sigma) & = & \sigma \to ([\![\tau]\!]_{\mathrm{type}}\{\alpha := \sigma\}) \\
[\![e.m]\!]_{\mathrm{exp}} & = & \mathsf{unpack}\ \alpha, x = [\![e]\!]_{\mathrm{exp}}\ \mathsf{in} \\
 & & \quad \mathsf{pack}\ x.\mathsf{mt}.m\ x, \\
 & & \quad\quad [\![\sigma]\!]_{\mathrm{type}}[\alpha := \alpha]\ \mathsf{as}\ [\![\tau]\!]_{\mathrm{type}}
\end{array}
$$

Where e has type τ and m's signature in τ is $\alpha.\sigma$. The translation could probably be extended to a structural rule for method invocation given a structural rule for unpack, details appear elsewhere [Gle00a]. The above type translation for self types is compatible with contravariant self types, but it is unclear what to do at the term level. This is unsurprising, since formulating object languages with contravariant self types and subsumption is problematic [Coo89b][AC96, §2.8 and §3.5].

O allows no depth subtyping in fields. O could use variances, as in $\mathcal{F}_{\mathsf{self}}$, to lift this restriction. Rows would have the form $[m_i : s_i; f_j : \sigma_j^{\phi_j}]_{i\in I, j\in J}$ and subtyping would now be:

$$
\frac{\begin{array}{c} i \in I_2 : \quad \vdash_{\mathcal{O}} s_i \le s'_i \\ j \in J_2 : \quad \vdash_{\mathcal{O}} \sigma_j^{\phi_j} \le {\sigma'_j}^{\phi'_j} \end{array}}{\begin{array}{c} \vdash_{\mathcal{O}} [m_i = s_i; f_j = \sigma_j^{\phi_j}]_{i\in I_1, j\in J_1} \le \\ [m_i = s'_i; f_j = {\sigma'_j}^{\phi'_j}]_{i\in I_2, j\in J_2} \end{array}}
$$

where I_1 is a prefix of I_2 and J_1 a prefix of J_2. The field extension operation would now have a variance $e + f : \sigma^{\phi}$, and a field override operation is now possible: $e.f := \sigma^{\phi}$ where σ^{ϕ} is a subtype/variance of the current type and variance of f in e. The translation then becomes:

$$
\begin{array}{l}
[\![r = [m_i{:}s_i; f_j{:}\sigma_j^{\phi_j}]_{i\in I, j\in J}]\!]_{\mathrm{full}}(\tau) = \\
\quad \langle \mathsf{mt}{:}[\![r]\!]_{\mathrm{mt}}(\tau, \to)^+, f_j{:}[\![\sigma_j]\!]_{\mathrm{type}}^{\phi_j}\rangle_{j\in J} \\
[\![e.f := \sigma^{\phi}]\!]_{\mathrm{exp}} = \\
\quad \forall\alpha \le [\![r']\!]_{\mathrm{full}}(\alpha).[\![e]\!]_{\mathrm{exp}}[\alpha]
\end{array}
$$

where $e.f := \sigma^{\phi}$ has type $\mathsf{tempt}\ r'$.

In a similar way, O could have variances on methods and allow method override on objects. Method override on objects is incompatible with the method-table technique. However, a version of the encoding that just uses the self-application semantics and not the method-table technique could easily incorporate method override. In fact, any of Abadi and Cardelli's pure object languages [AC96] can be encoded into suitable variants of O and thus be encoded using ideas in this paper. I believe that method extension on objects could also be incorporated, but initial investigation has revealed the need for some unusual structural rules for record extension. I leave for future work a full investigation of this possibility.

It is worth remarking that the encoding can be combined with object closure conversion [Gle99a] and an encoding of functions as objects to provide a typed translation of closure-passing-style closure conversion. In particular, a function of type $\tau_1 \to \tau_2$ translates into a closure of type $\mathsf{self}\ \alpha.\langle(\alpha, [\![\tau_1]\!]_{\mathrm{type}}) \to [\![\tau_2]\!]_{\mathrm{type}}\rangle$.

The templates in O allow only single inheritance and concrete methods. To finish this section, I will sketch a couple of generalisations that would allow abstract methods and multiple inheritance. The first generalisation separates the assumptions a template makes from what it provides. In this version, template values consist of a row that specifies the assumptions the template makes about the eventual object and a list of methods and methods bodies, thus a form like $\mathsf{temp}(r; m_i = M_i)_{i\in I}$. Template types would mention both the assumptions and list of provided methods and signatures, as in $\mathsf{tempt}(r; m_i{:}s_i)_{i\in I}$. There would be a template operation for strengthening the assumptions made, say $e := r$, where e has type $\mathsf{tempt}(r'; m_i = M_i)_{i\in I}$ and r is a subtype of r'. The method override and addition operations would be broken into operations to add single methods, $e + m = M$, and override single methods, $e.m := M$. The instantiation operation would check that all assumptions made about methods are provided by the template being instantiated. The following rule ensures this where $r = [m_i{:}s_i; f_j{:}\sigma_j]_{i\in I, j\in J}$:

$$
\frac{\Gamma \vdash_{\mathcal{O}} e : \mathsf{tempt}(r; m_i{:}s'_i)_{i\in I} \quad \vdash_{\mathcal{O}} s'_i \le s_i \quad \Gamma \vdash_{\mathcal{O}} e_j : \sigma_j}{\Gamma \vdash_{\mathcal{O}} \mathsf{new}\ e[f_j = e_j]_{j\in J} : \mathsf{objt}\ r}
$$

The translation could probably be revised based on the fol-

lowing type translation:

$$[\![\mathsf{tempt}(r; m_i : s_i)_{i\in I}]\!]_{\mathrm{type}} = \\ \forall\alpha \leq [\![r]\!]_{\mathrm{full}}(\alpha).\langle m_i : [\![s_i]\!]_{\mathrm{sig}}(\alpha)^+\rangle^\circ_{i\in I}$$

I leave it to future work to flesh out these ideas.

The second generalisation is to introduce a template combining operation $e_1 + e_2$. There are two possible interpretations for this operation, one could require e_1 and e_2 to provide disjoint sets of methods, the other could allow e_2 to override e_1. Taking the disjoint approach, a typing rule for this operation might be, where $m_{i\in I} \cap m'_{j\in J} = \emptyset$:

$$\frac{\Gamma \vdash_{\mathcal{O}} e_1 : \mathsf{tempt}(r; m_i{:}s_i)_{i\in I} \quad \Gamma \vdash_{\mathcal{O}} e_2 : \mathsf{tempt}(r; m'_j{:}s'_j)_{j\in J}}{\Gamma \vdash_{\mathcal{O}} e_1 + e_2 : \mathsf{tempt}(r; m_i{:}s_i, m'_j{:}s'_j)_{i\in I, j\in J}}$$

Translating this operation requires a record combining operation with similar properties. Again, I leave to future work the exploration of these ideas. Note that right-extension breadth subtyping limits the usefulness of these combining operations and should be abandoned in favour of arbitrary breadth subtyping. The translation, as stated in the previous section, works for O with arbitrary breadth subtyping so long as $\mathcal{F}_{\mathsf{self}}$ also has arbitrary breadth subtyping.

5. PREVIOUS WORK

5.1 Object Encodings

An object encoding should preserve the meaning of programs, and for typed translations must preserve both typing and subtyping. For use as foundations for language implementation, an encoding should also be efficient, and for use in certifying compilers, full abstraction is a useful property. In addition to these requirements, object encodings can be compared according to the features of the source language that they can encode. Bruce *et al.* [BCP99] provide an excellent comparison of most of the known object encodings.

Cardelli [Car88] proposed the first typed object encoding. In his encoding, an object is a record that can recursively refer to itself, often called a recursive record interpretation. At the type level, an object type is the fixed point of a record type whose elements are the methods' types. The encoding preserves meaning, typing, and subtyping, but it cannot encode method update, which is used to encode inheritance. The recursive records interpretation was pursued by Reddy [Red98, KR94], Cook [Coo89a, CHC90], the Hopkins Object Group [ESTZ95], and others.

Pierce and Turner [PT94] proposed a simple object and class encoding that requires existential types but not recursive types. At the term level, an object has two parts: a private state component and a public method suite. The functions that encode methods are passed the state of `self` but not `self`'s method suite. Calls to other methods of `self` must be hardwired at the time the object is created, and the class encoding arranges this. Furthermore, if a method's return type is the self type, then the function returns only the state component, and the method invocation sites must repackage the object. This encoding is the only encoding with a nonuniform translation of method invocation. The encoding preserves meaning, typing, and subtyping, but it cannot encode method update. The lack of method update is not a concern because a separate class encoding deals with inheritance. Finally, methods can be both private and public, but mutable fields can only be private, as public fields would not be passed to the methods' functions. Thus, in a sense, the encoding is for a different object model than Cardelli considered.

Bruce *et al.* [Bru94, BSvG95] designed a functional and an imperative class-based object-oriented language, and the denotational semantics for these languages can be seen as an object and class encoding. Like Pierce and Turner's encoding, the encoding has a complementary class encoding for dealing with inheritance. The encoding is very similar to Pierce and Turner's, but methods whose result type is the self type return the whole object not just the state component. Thus, the translation of an object type is like Pierce and Turner's but wrapped with an extra fixed point. Bruce *et al.* also argue for the use of matching rather than subtyping, which has many advantages, but leads to a different object and class model than Cardelli or Pierce and Turner's.

Rémy [Rém94, RV97] uses a variant of Pierce and Turner's encoding with row variables. Row variables are used to specify polymorphism over the type of `self`, and enable a natural extension of ML that includes objects and classes without sacrificing type inference. However, this system does not include subsumption: an object must be explicitly coerced from a subtype to a supertype.

In 1996, Abadi, Cardelli, and Viswanathan discovered an adequate typed object encoding for objects with method invocation and method update [ACV96]. This encoding uses bounded existential types and recursive types to effectively encode `self`'s type. However, the technique requires, purely for typing purposes, an additional projection and an additional field.

Abadi *et al.*'s encoding is also not fully abstract; in particular, the translation of method update allows the target language to distinguish objects that were indistinguishable in the source language. Viswanathan [Vis98] fixed this problem, but by introducing considerably more computation.

Concurrently with the work of this paper, Hickey, Crary, and League *et al.* have proposed typed encodings of the self-application semantics. Hickey [Hic] shows how to type the self-application semantics in the Nuprl type theory, using an intersection type to make methods polymorphic over the type of `self`. However, the Nuprl type theory is undecidable, so it is not clear how to use this encoding in a type-directed compiler. Crary [Cra99] shows how to use unbounded existential and binary intersection types to type the self application semantics. League *et al.* [LST99] show how to type the self application semantics using existentially quantified row variables and recursive types. They also show how to deal with classes, as described below. Both Crary and League *et al.*'s ideas can be seen as encodings of the self quantifier introduced in this paper.

5.2 Class Encodings

Abadi and Cardelli [AC96] show informally how to encode classes into their pure object calculi. In the encoding, a class becomes an object with premethods[5] for each of the instance's methods and a new method for instantiating the class. The new method copies the premethods into a newly created object. Subclasses copy the premethods of the superclass that they inherit and provide their own premethods for overridden or addition methods. F-bounded polymorphism is used to type the premethods. This encoding shows

[5]A premethod for a method is a function that takes `self` as an argument and computes the methods.

that classes add no expressiveness to a pure object calculus, but does not faithfully encode the efficient method-table technique.

Fisher and Mitchell with others [Fis96, Mit90, FHM94, FM95a, FM95b, FM96, BF98, FM98] have pursued a line of research into encoding classes as extensible objects. The object calculi they consider have a method extension operation for adding a new method to an already existing object. This construct does not appear in the object calculi usually considered for object encodings. Method extension interacts poorly with breadth subtyping, and so extensible object calculi need to have complicated type systems for tracking the absence of methods. Often a distinction is made between prototype objects, which are extensible but do not have breadth subtyping, and proper objects, which are not extensible but do have breadth subtyping. Like Abadi and Cardelli's class encoding, these encodings show that classes do not add expressiveness, and also provide a good basis for the design and definition of languages. However, also like Abadi and Cardelli's encoding, they do not lead directly to efficient implementations. In particular, class instantiation involves creating an empty object, and then adding all its methods to the object.

Pierce and Turner's class encoding [PT94], unlike the previous two approaches, encodes classes directly into records and functions and not into objects. Recall that in their encoding, an object consists of a public method suite and a private state component. A class is encoded as a function f that returns the public method suite for objects in that class. The public methods may want to invoke other public methods of `self`, so f takes another method suite s as an argument, and uses s to do method invocations on `self`. To instantiate a class, the fixed point of f is used as the public method suite of the new object along with an appropriate initial state component. However, subclasses may have more fields than superclasses, so f is parameterised by functions that convert between the final representation and the one the current class defines. Unfortunately, these conversion functions persist beyond class instantiation time and in general are evaluated every time a method is invoked, making this encoding particularly inefficient. Later work [HP95] shows how to avoid some of these problems.

Bruce *et al.*'s class encoding [Bru94, BSvG95] essentially encodes a class as a pair containing both the initial values of the private fields of the class and a function for the public methods. Additionally the pair is polymorphic in the final object type and the type of the private fields. The function for the public methods takes the final object and returns a record of the results of each method. Similarly to Pierce and Turner, class instantiation requires taking the fixed point of the function for the public methods to produce a function from the private state to the method suite, and then packaging this function with the initial private state. This encoding also results in inefficiencies.

Concurrently with the work of this paper, League *et al.* show how to encode a subset of the Java class model into a variant of F_ω [LST99]. A class is encoded as a method table (they call it a dictionary), a function to initialise the class's private fields, and a function to instantiate the class. Using a combination of row polymorphism and existential types, they are able to encode class private fields and their work can probably be extended to handle most of Java's protection modifiers.

Reppy and Reicke [RR96a] show how to encode classes as modules in the SML module system extended with objects in the core language [RR96b]. Their encoding is essentially the same as Abadi and Cardelli's, but with some twists for handling protection. Vouillon [Vou98] shows how to combine the classes and modules of Objective ML [RV97] into a single construct.

6. CONCLUSION

This paper presented a small language with the key features of a class-based object-oriented language and a typed encoding of this language into a language with records and functions. The encoding uses the self-application semantics and method-table techniques, providing a typed formalisation for both. Section 4 showed how to incorporate a number of extensions and variants into the template language and the encoding. This provides evidence that the ideas of this paper provide a nice framework for the description of class-based languages and their implementation and that the use of self quantifiers is the right way to type self application.

7. REFERENCES

[AC93] Roberto Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Progamming Languages and Systems*, 15(4):575–631, September 1993.

[AC96] Martín Abadi and Luca Cardelli. *A Theory Of Objects*. Springer-Verlag, 1996.

[ACV96] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 396–409, January 1996.

[BCP99] Kim Bruce, Luca Cardelli, and Benjamin Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, 1999.

[BF98] Viviana Bono and Kathleen Fisher. An imperative, first-order calculus with object extension. In *5th International Workshop on Foundations of Object Oriented Programming Languages*, pages 8–1 to 8–13, San Diego, California, USA, January 1998.

[Bru94] Kim Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, April 1994.

[BSvG95] Kim Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *1995 European Conference on Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 27–51. Springer-Verlag, 1995.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, 1988. Also published in volume 173 of Lecture Notes in Computer Science, pages 51–67, 1984.

[CCH+89] Peter Canning, William Cook, Walter Hill, John Mitchell, and Walter Olthoff. F-bounded quantification for object-oriented

programming. In *4th ACM Conference on Functional Programming and Computer Architecture*, pages 273–280, London, UK, September 1989. ACM Press.

[CG99] Dario Colazzo and Giorgio Ghelli. Subtyping recursive types in kernel fun. In *1999 Symposium on Logic in Computer Science*, pages 137–146, Trento, Italy, July 1999.

[CHC90] William Cook, Walter Hill, and Peter Canning. Inheritance is not subtyping. In *17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 125–135. ACM Press, January 1990.

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

[Coo89b] William Cook. A proposal for making Eiffel type-safe. In *3rd European Conference on Object-Oriented Programming*, pages 57–72, Nottingham, UK, July 1989. Cambridge University Press.

[Cra99] Karl Crary. Simple, efficient object encoding using intersection types. Technical Report CMU-CS-99-100, Carnegie Mellon University, Pittsburgh, PA 15213, USA, 1999.

[ESTZ95] Jonathan Eifrig, Scott Smith, Valery Trifonov, and Amy Zwarico. An interpretation of typed oop in a language with state. *Lisp and Symbolic Computation*, 8(4):357–397, 1995.

[FHM94] Kathleen Fisher, F. Honsell, and John Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1:3–37, 1994. Preliminary version appeared in IEEE Symposium on Logic in Computer Science, 1993, 26–38.

[Fis96] Kathleen Fisher. *Type Systems for object-oriented programming languages*. PhD thesis, Stanford University, California 94305, USA, 1996.

[FM95a] Kathleen Fisher and John Mitchell. A delegation-based object calculus with subtyping. In *10th International Conference on Fundamentals of Computation theory*, volume 965 of *Lecture Notes in Computer Science*, pages 42–61. Springer-Verlag, 1995.

[FM95b] Kathleen Fisher and John Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1(3):189–220, 1995.

[FM96] Kathleen Fisher and John Mitchell. Classes = objects + data abstraction. Technical Report STAN-CS-TN-96-31, Stanford University, California 94305, USA, 1996.

[FM98] Kathleen Fisher and John Mitchell. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems*, 1998. To appear.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[Gle99a] Neal Glew. Object closure conversion. In Andrew Gordon and Andrew Pitts, editors, *3rd International Workshop on Higher Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*, Paris, France, September 1999. Elsevier. `http://www.elsevier.nl/locate/entcs/volume26.html`.

[Gle99b] Neal Glew. Type dispatch for named hierarchical types. In *4th ACM SIGPLAN International Conference on Functional Programming*, Paris, France, September 1999.

[Gle00a] Neal Glew. An efficient class and object encoding. Technical Report STAR-TR-00.07-02, STAR Lab, InterTrust Technologies Corporation, 4750 Patrick Henry Drive, Santa Clara, CA 95054-1851, USA, July 2000.

[Gle00b] Neal Glew. *Low-Level Type Systems for Modularity and Object-Oriented Constructs*. PhD thesis, Cornell University, 4130 Upson Hall, Ithaca, NY 14853-7501, USA, January 2000.

[Hic] Jason Hickey. Predicative type-theoretic interpretation of objects. Unpublished, author's contact: `jyh@cs.cornell.edu`.

[HP95] Martin Hoffman and Benjamin Pierce. Positive subtyping. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 186–197, San Francisco, CA, USA, January 1995. ACM Press.

[HP98] Martin Hofmann and Benjamin Pierce. Type destructors. In *5th International Workshop on Foundations of Object Oriented Programming Languages*, pages 3–1 to 3–11, San Diego, CA, USA, January 1998.

[Kam88] Samuel Kamin. Inheritance in SMALLTALK-80: A denotational definition. In *15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 80–87, San Diego, CA, USA, January 1988.

[KPS95] Dexter Kozen, Jens Palsberg, and Michael Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science*, 5(1):113–125, March 1995.

[KR94] Samuel Kamin and Uday Reddy. Two semantic models of object-oriented languages. In Carl Gunter and John Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. MIT Press, Cambridge, MA 02142, 1994.

[LST99] Christopher League, Zhong Shao, and Valerey Trifonov. Representing java classes in a typed intermediate language. In *1999 ACM SIGPLAN International Conference on Functional Programming*, Paris, France, September 1999. ACM Press.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *ACM SIGPLAN Workshop on Compiler Support for System*

Software, volume 0228 of *INRIA Research Reports*, Atlanta, GA, USA, May 1999.

[Mit90] John Mitchell. Toward a typed foundation for method specialization and inheritance. In *17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 109–124. ACM Press, January 1990.

[MTC+96] Greg Morrisett, David Tarditi, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. The TIL/ML compiler: Performance and safety through types. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, Tucson, AZ, USA, February 1996.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Progamming Languages and Systems*, 21(3):528–569, May 1999.

[NL98] George Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–344, Montreal, Canada, June 1998.

[PT94] Benjamin Pierce and David Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. An early version appeared in POPL'93, pages 299–312.

[Red98] Uday Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *ACM Symposium on LISP and Functional Programming*, pages 289–297. ACM, July 1998.

[Rém94] Didier Rémy. Programming objects with ML-ART, an extension to ML with abstract and record types. In Masami Hagiya and John Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Science*, volume 789 of *Lecture Notes in Computer Science*, pages 321–346, Sendai, Japan, April 1994. Springer-Verlag.

[RR96a] John Reppy and Jon Riecke. Classes in Object ML via modules. In *3rd International Workshop on Foundations of Object Oriented Programming Languages*, New Brunswick, NJ, USA, July 1996.

[RR96b] John Reppy and Jon Riecke. Simple objects for Standard ML. In *1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 171–180. ACM Press, 1996.

[RV97] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object-oriented extension of ML. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 40–53, Paris, France, January 1997.

[TMC+96] David Tarditi, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, USA, May 1996. ACM Press.

[Vis98] Ramesh Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *13th Symposium on Logic in Computer Science*, 1998.

[Vou98] Jérôme Vouillon. Using modules as classes. In *5th International Workshop on Foundations of Object Oriented Programming Languages*, pages 4–1 to 4–10, San Diego, CA, USA, January 1998.

APPENDIX

A. O SEMANTICS

The operational semantics for the template language appears in Figure 3. Capture avoiding substitution of x for y in z is written $z\{y := x\}$.

A.1 Subtyping rules

$$\frac{}{\vdash_{\mathcal{O}} \mathsf{tempt}\ r \leq \mathsf{tempt}\ r}$$

$$\frac{i \in I_2: \quad \vdash_{\mathcal{O}} s_i \leq s'_i}{\vdash_{\mathcal{O}} \mathsf{objt}[m_i{:}s_i; f_j{:}\sigma_j]_{i\in I_1, j\in J_1} \leq \mathsf{objt}[m_i{:}s'_i; f_j{:}\sigma_j]_{i\in I_2, j\in J_2}}$$

where I_1 is a prefix of I_2 and J_1 is a prefix of J_2.

A.2 Expression Typing

A typing context Γ is a list of variables and their types, $x_1{:}\tau_1, \ldots, x_n{:}\tau_n$, where the variables are distinct.

$$\frac{\Gamma \vdash_{\mathcal{O}} e : \tau_1 \quad \vdash_{\mathcal{O}} \tau_1 \leq \tau_2}{\Gamma \vdash_{\mathcal{O}} e : \tau_2} \qquad \frac{}{\Gamma \vdash_{\mathcal{O}} x : \tau}\ (\Gamma(x) = \tau)$$

$$\frac{}{\Gamma \vdash_{\mathcal{O}} \mathsf{et} : \mathsf{tempt}[;]}$$

$$\frac{\Gamma \vdash_{\mathcal{O}} e : \mathsf{tempt}[m_i{:}s_i; f_j{:}\sigma_j]_{i\in I, j\in J}}{\Gamma \vdash_{\mathcal{O}} e + f : \sigma : \mathsf{tempt}[m_i{:}s_i; f_j{:}\sigma_j, f{:}\sigma]_{i\in I, j\in J}}\ (f \notin f_{j\in J})$$

$$\frac{\begin{array}{ll} & \Gamma \vdash_{\mathcal{O}} e : \mathsf{tempt}[m_i{:}s_i; f_j{:}\sigma_j]_{i\in I, j\in J} \\ i \in K: & \Gamma \vdash^{\mathsf{M}}_{\mathcal{O}} M_i : \mathsf{objt}\ r' \rhd s'_i \\ i \in I \cap K: & \vdash_{\mathcal{O}} s'_i \leq s_i \end{array}}{\Gamma \vdash_{\mathcal{O}} e \leftarrow\!\!+ [m_i = M_i]_{i\in K} : \mathsf{tempt}\ r'}$$

where $r' = [m_i{:}s''_i; f_j{:}\sigma_j]_{i\in(I,K-I), j\in J}$, $s''_i = s_i$ if $i \in I - K$, and $s''_i = s'_i$ if $i \in K$.

$$\frac{\Gamma \vdash_{\mathcal{O}} e : \mathsf{tempt}\ r \quad \Gamma \vdash_{\mathcal{O}} e_j : \sigma_j}{\Gamma \vdash_{\mathcal{O}} \mathsf{new}\ e[f_j = e_j]_{j\in J} : \mathsf{objt}\ r}\ (r = [m_i{:}s_i; f_j{:}\sigma_j]_{i\in I, j\in J})$$

$$\frac{\Gamma \vdash_{\mathcal{O}} e : \mathsf{objt}[m_i{:}s_i; f_j{:}\sigma_j]_{i\in I, j\in J}}{\Gamma \vdash_{\mathcal{O}} e.m_k : s_k}\ (k \in I)$$

$$\frac{\Gamma \vdash_{\mathcal{O}} e : \mathsf{objt}[m_i{:}s_i; f_j{:}\sigma_j]_{i\in I, j\in J}}{\Gamma \vdash_{\mathcal{O}} e.f_k : \sigma_k}\ (k \in J)$$

$$\frac{\Gamma \vdash_{\mathcal{O}} e_1 : \tau_1 \quad \Gamma \vdash_{\mathcal{O}} e_2 : \sigma_k}{\Gamma \vdash_{\mathcal{O}} e_1.f_k := e_2 : \tau_1}\ (k \in J)$$

Additional syntactic constructs:

$$\begin{array}{llll} \text{Values} & v, w & ::= & \mathsf{temp}[m_i = M_i; f_j{:}\sigma_j]_{i\in I, j\in J} \mid \mathsf{obj}[m_i = M_i; f_j = v_j]_{i\in I, j\in J} \\ \text{Contexts} & E & ::= & \{\} \mid E + f : \sigma \mid E \leftarrow\!\!+ [m_i = M_i]_{i\in I} \mid \mathsf{new}\ E[f_j = e_j]_{j\in J} \mid \mathsf{new}\ v[\overrightarrow{f = v}, f = E, \overrightarrow{f' = e}] \mid \\ & & & E.m \mid E.f \mid E.f := e \mid v.f := E \end{array}$$

Reduction rules:

$$E\{\iota\} \mapsto E\{e\}$$
where:

ι	e	Side Conditions
et	$\mathsf{temp}[;]$	
$\mathsf{temp}[m_i = M_i; f_j{:}\sigma_j]_{i\in I, j\in J} + f : \sigma$	$\mathsf{temp}[m_i = M_i; f_j{:}\sigma_j, f{:}\sigma]_{i\in I, j\in J}$	$f \notin f_{j\in J}$
$\mathsf{temp}[m_i = M_i; f_j{:}\sigma_j]_{i\in I, j\in J} \leftarrow\!\!+ [m_k = M'_k]_{k\in K}$	$\mathsf{temp}[m_l = M''_l; f_j{:}\sigma_j]_{l\in(I,K-I), j\in J}$	$M''_l = \begin{cases} M_l & l \in I - K \\ M'_l & l \in K \end{cases}$
$\mathsf{new}\ \mathsf{temp}[m_i = M_i; f_j{:}\sigma_j]_{i\in I, j\in J}[f_j = w_j]_{j\in J}$	$\mathsf{obj}[m_i = M_i; f_j = w_j]_{i\in I, j\in J}$	
$\mathsf{obj}[m_i = M_i; f_j = w_j]_{i\in I, j\in J}.m_k$	$e_k\{x_k := v_2\}$	$k \in I$
$\mathsf{obj}[m_i = M_i; f_j = w_j]_{i\in I, j\in J}.f_k$	w_k	$k \in J, M_k = x_k.e_k{:}\tau_k$
$\mathsf{obj}[m_i = M_i; f_j = w_j]_{i\in I, j\in J}.f_k := v$	$\mathsf{obj}[m_i = M_i; f_j = w'_j]_{i\in I, j\in J}$	$k \in J; w'_j = \begin{cases} w_j & j \neq k \\ v & j = k \end{cases}$

Figure 3: Object Template Language Operational Semantics

where $\tau_1 = \mathsf{objt}[m_i{:}s_i; f_j{:}\sigma_j]_{i\in I, j\in J}$.

$$\frac{\Gamma, x : \sigma \vdash_{\mathcal{O}} e : \tau}{\Gamma \vdash^{\mathsf{M}}_{\mathcal{O}} x.e{:}\tau : \sigma \rhd \tau}$$

B. $\mathcal{F}_{\mathsf{self}}$ SEMANTICS

The operational semantics for the target language appears in Figure 4.

B.1 Type well formedness

$$\frac{}{\Delta \vdash_{\mathcal{F}} \tau}\ (\mathrm{ftv}(\tau) \subseteq \Delta)$$

where $\mathrm{ftv}(\tau)$ are the free type variables of τ.

B.2 Equality Rules

Along with the usual reflexivity, transitivity, and congruence rules, $\mathcal{F}_{\mathsf{self}}$ has the following equality rules.

$$\frac{\Delta \vdash_{\mathcal{F}} \tau}{\Delta \vdash_{\mathcal{F}} \tau = \sigma\{\alpha := \tau\}}\ (\tau = \mathsf{rec}\ \alpha.\sigma)$$

$$\frac{\Delta \vdash_{\mathcal{F}} \tau\{\alpha := \sigma_1\} = \sigma_1 \quad \Delta \vdash_{\mathcal{F}} \tau\{\alpha := \sigma_2\} = \sigma_2}{\Delta \vdash_{\mathcal{F}} \sigma_1 = \sigma_2}\ (\tau \downarrow \alpha)$$

Where $\tau \downarrow \alpha$ means that τ is syntactically contractive in α, that is, there is a function, record, polymorphic type constructor, or self quantifier before any occurrences of α.

B.3 Subtyping Rules

$$\frac{\Delta; B \vdash_{\mathcal{F}} \tau_1 = \tau_2}{\Delta; B \vdash_{\mathcal{F}} \tau_1 \leq \tau_2} \qquad \frac{\Delta; B \vdash_{\mathcal{F}} \tau_1 \leq \tau_2 \quad \Delta; B \vdash_{\mathcal{F}} \tau_2 \leq \tau_3}{\Delta; B \vdash_{\mathcal{F}} \tau_1 \leq \tau_3}$$

$$\frac{}{\Delta; B \vdash_{\mathcal{F}} \alpha \leq \tau}\ (\alpha \in \Delta; \alpha \leq \tau \in B)$$

$$\frac{\Delta; B \vdash_{\mathcal{F}} \sigma_1 \leq \tau_1 \quad \Delta; B \vdash_{\mathcal{F}} \tau_2 \leq \sigma_2}{\Delta; B \vdash_{\mathcal{F}} \tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2}$$

$$\frac{j \in I_2 : \Delta; B \vdash_{\mathcal{F}} \tau_j^{\phi_j} \leq \tau_j'^{\phi'_j} \quad k \in I_1 - I_2 : \Delta \vdash_{\mathcal{F}} \tau_k}{\Delta; B \vdash_{\mathcal{F}} \langle \ell_i{:}\tau_i^{\phi_i}\rangle^{\varphi_1}_{i\in I_1} \leq \langle \ell_j{:}\tau_j'^{\phi'_j}\rangle^{\varphi_2}_{j\in I_2}}$$

where I_2 is prefix of I_1 if $\varphi_2 = \!\!\rightarrow$, otherwise $I_1 = I_2$ and $\varphi_1 = \circ$.

$$\frac{\Delta, \alpha \vdash_{\mathcal{F}} \tau_{11} = \tau_{21} \quad \Delta, \alpha; B, \alpha \leq \tau_{11} \vdash_{\mathcal{F}} \tau_{12} \leq \tau_{22}}{\Delta; B \vdash_{\mathcal{F}} \forall\alpha \leq \tau_{11}.\tau_{12} \leq \forall\alpha \leq \tau_{21}.\tau_{22}}$$

$$\frac{\Delta, \alpha; B \vdash_{\mathcal{F}} \tau_1 \leq \tau_2}{\Delta; B \vdash_{\mathcal{F}} \mathsf{self}\ \alpha.\tau_1 \leq \mathsf{self}\ \alpha.\tau_2}$$

$$\frac{\Delta, \alpha_1, \alpha_2; B, \alpha_1 \leq \alpha_2 \vdash_{\mathcal{F}} \tau_1 \leq \tau_2}{\Delta; B \vdash_{\mathcal{F}} \mathsf{rec}\ \alpha_1.\tau_1 \leq \mathsf{rec}\ \alpha_2.\tau_2}\ (\alpha_1 \neq \alpha_2)$$

$$\frac{\Delta; B \vdash_{\mathcal{F}} \tau_1 \leq \tau_2}{\Delta; B \vdash_{\mathcal{F}} \tau_1^{\phi} \leq \tau_2^{+}}\ (\phi \in \{+, \circ\})$$

$$\frac{\Delta \vdash_{\mathcal{F}} \tau_1 = \tau_2}{\Delta; B \vdash_{\mathcal{F}} \tau_1^{\circ} \leq \tau_2^{\circ}}$$

B.4 Expression Typing

$$\frac{\Delta; B; \Gamma \vdash_{\mathcal{F}} e : \tau_1 \quad \Delta; B \vdash_{\mathcal{F}} \tau_1 \leq \tau_2}{\Delta; B; \Gamma \vdash_{\mathcal{F}} e : \tau_2}$$

$$\frac{}{\Delta; B; \Gamma \vdash_{\mathcal{F}} x : \tau}\ (\Gamma(x) = \tau)$$

$$\frac{\Delta \vdash_{\mathcal{F}} \tau_1 \quad \Delta; B; \Gamma, x : \tau_1 \vdash_{\mathcal{F}} e : \tau_2}{\Delta; B; \Gamma \vdash_{\mathcal{F}} \lambda x : \tau_1.e : \tau_1 \to \tau_2}$$

$$\frac{\Delta; B; \Gamma \vdash_{\mathcal{F}} e_1 : \tau_2 \to \tau_1 \quad \Delta; B; \Gamma \vdash_{\mathcal{F}} e_2 : \tau_2}{\Delta; B; \Gamma \vdash_{\mathcal{F}} e_1\ e_2 : \tau_1}$$

Additional syntactic constructs:

$$\begin{array}{llll} \text{Values} & v, w & ::= & \lambda x{:}\tau.e \mid \langle \ell_i = v_i \rangle_{i \in I} \mid \Lambda\alpha \leq \tau.v \mid \mathsf{pack}\ v, \tau\ \mathsf{as\ self}\ \alpha.\sigma \\ \text{Contexts} & E & ::= & \{\} \mid E\ e \mid v\ E \mid \langle \overrightarrow{\ell = v}, \ell = E, \overrightarrow{\ell' = e} \rangle \mid E.\ell \mid E.\ell \leftarrow e \mid v.\ell \leftarrow E \mid E.\ell := e \mid v.\ell := E \mid \\ & & & E + \ell = e \mid v + \ell = E \mid \Lambda\alpha \leq \tau.E \mid E[\tau] \mid \mathsf{pack}\ E, \tau\ \mathsf{as\ self}\ \alpha.\sigma \mid \mathsf{unpack}\ \alpha, x = E\ \mathsf{in}\ e \end{array}$$

Reduction rules:

$$E\{\iota\} \mapsto E\{e\}$$

Where:

ι	e	Side Conditions
$(\lambda x{:}\tau.e)\ v$	$e\{x := v\}$	
$\langle \ell_i = v_i \rangle_{i \in I}.\ell_k$	v_k	$k \in I$
$\left\{ \begin{array}{l} \langle \ell_i = v_i \rangle_{i \in I}.\ell_k \leftarrow v \\ \langle \ell_i = v_i \rangle_{i \in I}.\ell_k := v \end{array} \right\}$	$\langle \ell_i = v'_i \rangle_{i \in I}$	$k \in I; v'_i = \left\{ \begin{array}{ll} v_i & i \neq k \\ v & i = k \end{array} \right.$
$\langle \ell_i = v_i \rangle_{i \in I} + \ell = v$	$\langle \ell_i = v_i, \ell = v \rangle_{i \in I}$	$\ell \notin \ell_{i \in I}$
$(\Lambda\alpha \leq \tau.v)[\sigma]$	$v\{\alpha := \sigma\}$	
$\mathsf{unpack}\ \alpha, x = \mathsf{pack}\ v, \tau\ \mathsf{as\ self}\ \alpha.\sigma\ \mathsf{in}\ e$	$e\{\alpha, x := \tau, v\}$	

Figure 4: Target Language Operational Semantics

$$\frac{\Delta; B; \Gamma \vdash_{\mathcal{F}} e_i : \tau_i}{\Delta; B; \Gamma \vdash_{\mathcal{F}} \langle \ell_i = e_i \rangle_{i \in I} : \langle \ell_i{:}\tau_i^{\circ} \rangle_{i \in I}^{\circ}}$$

$$\frac{\Delta; B; \Gamma \vdash_{\mathcal{F}} e : \langle \ell_i{:}\tau_i^{\phi_k} \rangle_{i \in I}^{\varphi}}{\Delta; B; \Gamma \vdash_{\mathcal{F}} e.\ell_k : \tau_k}\ (k \in I; \phi_k \in \{+, \circ\})$$

$$\frac{\Delta; B; \Gamma \vdash_{\mathcal{F}} e_1 : \langle \ell_i{:}\tau_i^{\phi_i} \rangle_{i \in I}^{\varphi} \quad \Delta; B; \Gamma \vdash_{\mathcal{F}} e_2 : \sigma}{\Delta; B; \Gamma \vdash_{\mathcal{F}} e_1.\ell_k \leftarrow e_2 : \langle \ell_i{:}\tau_i'^{\phi'_i} \rangle_{i \in I}^{\varphi}}\ (k \in I)$$

where $\tau_i'^{\phi'_i} = \tau_i^{\phi_i}$ if $i \neq k$ and $\tau_k'^{\phi'_k} = \sigma^{\circ}$.

$$\frac{\begin{array}{c} \Delta; B; \Gamma \vdash_{\mathcal{F}} e_1 : \sigma_1 \\ \Delta; B \vdash_{\mathcal{F}} \sigma_1 \leq \langle \ell_i{:}\tau_i^{\phi_i} \rangle_{i \in I}^{\varphi} \\ \Delta; B; \Gamma \vdash_{\mathcal{F}} e_2 : \tau_k \end{array}}{\Delta; B; \Gamma \vdash_{\mathcal{F}} e_1.\ell_k := e_2 : \sigma_1}\ (k \in I; \phi_k = \circ)$$

$$\frac{\Delta; B; \Gamma \vdash_{\mathcal{F}} e_1 : \langle \ell_i{:}\tau_i^{\phi_i} \rangle_{i \in I}^{\circ} \quad \Delta; B; \Gamma \vdash_{\mathcal{F}} e_2 : \sigma}{\Delta; B; \Gamma \vdash_{\mathcal{F}} e_1 + \ell = e_2 : \langle \ell_i{:}\tau_i^{\phi_i}, \ell{:}\sigma^{\circ} \rangle_{i \in I}^{\circ}}\ (\ell \notin \ell_{i \in I})$$

$$\frac{\Delta, \alpha \vdash_{\mathcal{F}} \tau_1 \quad \Delta, \alpha; B, \alpha \leq \tau_1 \vdash_{\mathcal{F}} e : \tau_2}{\Delta; B; \Gamma \vdash_{\mathcal{F}} \Lambda\alpha \leq \tau_1.e : \forall \alpha \leq \tau_1.\tau_2}$$

$$\frac{\Delta; B; \Gamma \vdash_{\mathcal{F}} e : \forall \alpha \leq \tau_1.\tau_2 \quad \Delta; B \vdash_{\mathcal{F}} \sigma \leq \tau_1\{\alpha := \sigma\}}{\Delta; B; \Gamma \vdash_{\mathcal{F}} e[\sigma] : \tau_2\{\alpha := \sigma\}}$$

$$\frac{\Delta; B; \Gamma \vdash_{\mathcal{F}} e : \tau \quad \Delta; B \vdash_{\mathcal{F}} \tau \leq \sigma\{\alpha := \tau\}}{\Delta; B; \Gamma \vdash_{\mathcal{F}} \mathsf{pack}\ e, \tau\ \mathsf{as\ self}\ \alpha.\sigma : \mathsf{self}\ \alpha.\sigma}$$

$$\frac{\begin{array}{c} \Delta; B; \Gamma \vdash_{\mathcal{F}} e_1 : \mathsf{self}\ \alpha.\tau_1 \\ \Delta, \alpha; B, \alpha \leq \tau_1; \Gamma, x : \alpha \vdash_{\mathcal{F}} e_2 : \tau_2 \\ \Delta \vdash_{\mathcal{F}} \tau_2 \end{array}}{\Delta; B; \Gamma \vdash_{\mathcal{F}} \mathsf{unpack}\ \alpha, x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$