Technical Report STAR-TR-00-02
An Efficient Class and Object Encoding [*]

Neal Glew

July 2000
(Revised December 2000)

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.

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

[*] 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.

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.

This paper is a longer version of a paper that appears in OOPSLA 2000 [Gle00a]. It contains proofs, more formal language details, and elaboration of covariant self types and closure conversion. 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 [Gle99c]. The other roles, while important, are beyond the scope of this paper and are left to future work.

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 {
  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

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:

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

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:

$$\begin{aligned}
&\mathsf{let}\ w1 = \mathsf{new}\ \mathtt{Window}_{\mathtt{t}}[\mathtt{extent} = r1]\ \mathsf{in}\\
&\mathsf{let}\ w2 = \mathsf{new}\ \mathtt{ContainerWindow}_{\mathtt{t}}[\mathtt{extent} = r2, \mathtt{children} = \mathsf{array}()]\ \mathsf{in}
\end{aligned}$$

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:

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

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 $e \leftarrow\!\!+ [m_i = M_i]_{i \in I}$ 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:

[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.

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:

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

where I_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:

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

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 $i \in K$.

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, 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.

2.1 Formal Details

The operational semantics of $\mathcal{O}$ appears in Figure 2. The capture avoiding substitution of x for y in z is written $z\{y := x\}$; the substitution of an expression e for the unique hole ($\{\}$) in an evaluation context E is written $E\{e\}$. The sematnics is a left to right, call by value, context based, reduction semantics. Expressions of a template type evaluate to template values of the form $\mathsf{temp}[m_i = M_i; f_j{:}\sigma_j]_{i\in I, j\in J}$, which list the methods and their implementations and list the fields and their types. Expressions of an object type evaluate to object values of the form

Additional syntactic constructs:

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

Reduction rules:

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

$$\begin{array}{rrcl} \text{where:} & v1 & = & \mathsf{temp}[m_i = M_i; f_j{:}\sigma_j]_{i\in I, j\in J} \\ & v2 & = & \mathsf{obj}[m_i = M_i; f_j = w_j]_{i\in I, j\in J} \\ & M_i & = & x_i.e_i{:}\tau_i \end{array}$$

ι	e	Side Conditions
et	$\mathsf{temp}[;]$	
$v1 + f : \sigma$	$\mathsf{temp}[m_i = M_i; f_j{:}\sigma_j, f{:}\sigma]_{i\in I, j\in J}$	$f \notin f_{j\in J}$
$v1 \leftarrow\!\!+ [m_k = M'_k]_{k\in K}$	$\mathsf{temp}[m_l = M''_l; f_j{:}\sigma_j]_{l\in(I,K-I), j\in J}$	$M''_l = \begin{cases} M_l & l \in I - K \\ M'_l & l \in K \end{cases}$
$\mathsf{new}\ v1[f_j = w_j]_{j\in J}$	$v2$	
$v2.m_k$	$e_k\{x_k := v_2\}$	$k \in I$
$v2.f_k$	w_k	$k \in J$
$v2.f_k := v$	$\mathsf{obj}[m_i = M_i; f_j = w'_j]_{i\in I, j\in J}$	$k \in J; w'_j = \begin{cases} w_j & j \neq k \\ v & j = k \end{cases}$

Figure 2: Object Template Language Operational Semantics

$\boxed{\vdash_{\mathcal{O}} \tau_1 \leq \tau_2}$

$$\text{(subtemp)}\ \frac{}{\vdash_{\mathcal{O}} \mathsf{tempt}\ r \leq \mathsf{tempt}\ r}$$

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

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

$\boxed{\Gamma \vdash_{\mathcal{O}} e : \tau \quad \Gamma \vdash^{\mathsf{M}}_{\mathcal{O}} M : \tau \rhd s}$

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

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

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

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 $i \in K$.

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

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

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

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

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

$$\text{(method)}\ \frac{\Gamma, x : \sigma \vdash_{\mathcal{O}} e : \tau}{\Gamma \vdash^{\mathsf{M}}_{\mathcal{O}} x.e{:}\tau : \sigma \rhd \tau} \qquad \text{(meth-sub)}\ \frac{\Gamma \vdash^{\mathsf{M}}_{\mathcal{O}} M : \sigma_2 \rhd s \quad \vdash_{\mathcal{O}} \sigma_1 \leq \sigma_2}{\Gamma \vdash^{\mathsf{M}}_{\mathcal{O}} M : \sigma_1 \rhd s}$$

$$\text{(temp)}\ \frac{\Gamma \vdash^{\mathsf{M}}_{\mathcal{O}} M_i : \mathsf{objt}\ r \rhd s_i}{\Gamma \vdash_{\mathcal{O}} \mathsf{temp}[m_i = M_i; f_j{:}\sigma_j]_{i\in I, j\in J} : \mathsf{tempt}\ r}$$

$$\text{(obj)}\ \frac{\Gamma \vdash^{\mathsf{M}}_{\mathcal{O}} M_i : \mathsf{objt}\ r \rhd s_i \quad \Gamma \vdash_{\mathcal{O}} v_j : \sigma_j}{\Gamma \vdash_{\mathcal{O}} \mathsf{obj}[m_i = M_i; f_j = v_j]_{i\in I, j\in J} : \mathsf{objt}\ r}$$

where, for the last two rules, $r = [m_i{:}s_i; f_j{:}\sigma_j]_{i\in I, j\in J}$.

Figure 3: $\mathcal{O}$ Typing Rules

$\mathsf{obj}[m_i = M_i; f_j = v_j]_{i \in I, j \in J}$, which list the methods and their implementations and list the fields and their values.

The typing rules axiomatise three judgements: The judgement $\vdash_{\mathcal{O}} \tau_1 \leq \tau_2$ asserts that τ_1 is a subtype of τ_2. The judgement $\Gamma \vdash_{\mathcal{O}} e : \tau$ asserts that e has type τ in context Γ. The judgements $\Gamma \vdash^{\mathsf{M}}_{\mathcal{O}} M : \tau \rhd s$ asserts that M has signature s if `self` has type τ. A context is a list of variables and their types, $x_1{:}\tau_1, \ldots, x_n{:}\tau_n$, where the variables are distinct. The typing rules appear in Figure 3. They are sound with respect to the operational semantics, as proven in Appendix A.

3 Encoding the Template Language

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 3.3 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 $(\alpha, \texttt{Event}) \rightarrow \mathsf{bool}$.

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:

$$\begin{array}{l}\langle \texttt{handleEvent} = \texttt{ContainerWindow::handleEvent}, \\ \quad \texttt{contains} = \texttt{Window::contains}, \\ \quad \texttt{addChild} = \texttt{ContainerWindow::addChild} \rangle\end{array}$$

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

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

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 $[\![r]\!]_{\mathrm{full}}(\alpha)$ type, `Window`'s method table gets type:

$$\begin{array}{c}\langle \texttt{handleEvent} : \forall \alpha \leq [\![\texttt{Window}]\!]_{\mathrm{full}}(\alpha).(\alpha, \texttt{Event}) \rightarrow \mathsf{bool}, \\ \texttt{contains} : \forall \alpha \leq [\![\texttt{Window}]\!]_{\mathrm{full}}(\alpha).(\alpha, \texttt{Point}) \rightarrow \mathsf{bool} \rangle\end{array}$$

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:

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

This means that a method table can be installed into an object simply by instantiating it at an appropriate type. In general $[\![\mathsf{temp}\ r]\!]_{\mathrm{type}} = \forall \alpha \leq [\![r]\!]_{\mathrm{full}}(\alpha)\ .\ [\![r]\!]_{\mathrm{mt}}(\alpha)$ and $[\![r]\!]_{\mathrm{mt}}(\alpha)$ 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:

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

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

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

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

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

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

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

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

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:

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

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:

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

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

[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.

e's self type, that is, e must have type $\tau\{\alpha := \sigma\}$. The latter is achieved by requiring that $\sigma \leq \tau\{\alpha := \sigma\}$. Using $\mathtt{pack}$, the translation of $\mathsf{new}\ \mathtt{Window}_{\mathtt{t}}[\mathtt{extent} = r1]$ is:

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

Here, the object's run-time is $\mathsf{rec}\ \alpha.[\![\mathtt{Window}]\!]_{\mathrm{full}}(\alpha)$. For this type to satisfy the requirements for packing into a self type, the condition $\Delta; B \vdash_{\mathcal{F}} \sigma \leq \tau\{\alpha := \sigma\}$ above, it must be the case that $\mathsf{rec}\ \alpha.\tau \leq \tau\{\alpha := \mathsf{rec}\ \alpha.\tau\}$. 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:

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

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:

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

Note, however, that the bound $\alpha \leq \tau_1$ 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 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:

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

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:

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

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.

3.2 Formal Details

The operational semantics for $\mathcal{F}_{\mathsf{self}}$ appears in Figure 4. It is a left to right, call by value, context based, reduction semantics.

The typing rules for $\mathcal{F}_{\mathsf{self}}$ axiomatise five judgements: The judgement $\Delta \vdash_{\mathcal{F}} \tau$ asserts that τ is a well formed type in type context Δ. The judgement $\Delta \vdash_{\mathcal{F}} \tau_1 = \tau_2$ asserts that τ_1 and τ_2 are equal types in type context Δ. The judgement $\Delta; B \vdash_{\mathcal{F}} \tau_1 \leq \tau_2$ asserts that τ_1 is a subtype of τ_2 in type context Δ and with bounds B. The judgement $\Delta; B \vdash_{\mathcal{F}} \tau_1^{\phi_1} \leq \tau_2^{\phi_2}$ asserts that τ_1 and ϕ_1 are a subtype and subvariance of τ_2 and ϕ_2 in type context Δ and bounds B. The judgement $\Delta; B; \Gamma \vdash_{\mathcal{F}} e : \tau$ asserts that e has type τ in type context Δ, bounds B, and value

Additional syntactic constructs:

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

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

context Γ. A type context Δ is a sequence of distinct type variables, $\alpha_1, \ldots, \alpha_n$. A bounds set B is a sequence $\alpha_1 \leq \tau_1, \ldots, \alpha_n \leq \tau_n$ where the α_i are distinct; it is well formed, $\Delta \vdash_{\mathcal{F}} B$, when $\alpha_i \in \Delta$ and $\Delta \vdash_{\mathcal{F}} \tau_i$. A value context Γ is a sequence of variables and their types, $x_1{:}\tau_1, \ldots, x_n{:}\tau_n$; it is well formed, $\Delta \vdash_{\mathcal{F}} \Gamma$, when $\Delta \vdash_{\mathcal{F}} \tau_i$. The typing rules appear in Figures 5 and 6. One of the equality rules uses a predicate $\tau \downarrow \alpha$, read τ is contractive in α. It is defined inductively as follows:

$$\begin{array}{lll} \beta \downarrow \alpha & \Leftarrow & \alpha \neq \beta \\ \tau_1 \to \tau_2 \downarrow \alpha & & \\ \langle \ell_i{:}\tau_i^{\phi_i} \rangle_{i \in I}^{\varphi} \downarrow \alpha & & \\ \mathsf{self}\ \beta.\tau \downarrow \alpha & & \\ \mathsf{rec}\ \beta.\tau \downarrow \alpha & \Leftarrow & \alpha = \beta \vee \tau \downarrow \alpha \end{array}$$

The typing rules are sound with respect to the operational semantics, see Appendix B. 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

$$\boxed{\Delta \vdash_{\mathcal{F}} \tau \quad \Delta \vdash_{\mathcal{F}} \tau_1 = \tau_2}$$

$$\text{(wft)}\ \frac{\Delta \vdash_{\mathcal{F}} \tau = \tau}{\Delta \vdash_{\mathcal{F}} \tau} \qquad \text{(eqs)}\ \frac{\Delta \vdash_{\mathcal{F}} \tau_2 = \tau_1}{\Delta \vdash_{\mathcal{F}} \tau_1 = \tau_2} \qquad \text{(eqt)}\ \frac{\Delta \vdash_{\mathcal{F}} \tau_1 = \tau_2 \quad \Delta \vdash_{\mathcal{F}} \tau_2 = \tau_3}{\Delta \vdash_{\mathcal{F}} \tau_1 = \tau_3}$$

$$\text{(eqtv)}\ \frac{}{\Delta \vdash_{\mathcal{F}} \alpha = \alpha}\ (\alpha \in \Delta)$$

$$\text{(eqfun)}\ \frac{\Delta \vdash_{\mathcal{F}} \tau_{11} = \tau_{21} \quad \Delta \vdash_{\mathcal{F}} \tau_{12} = \tau_{22}}{\Delta \vdash_{\mathcal{F}} \tau_{11} \to \tau_{12} = \tau_{21} \to \tau_{22}} \qquad \text{(eqtup)}\ \frac{\Delta \vdash_{\mathcal{F}} \tau_i = \sigma_i}{\Delta \vdash_{\mathcal{F}} \langle \ell_i{:}\tau_i^{\phi_i} \rangle_{i \in I}^{\varphi} = \langle \ell_i{:}\sigma_i^{\phi_i} \rangle_{i \in I}^{\varphi}}$$

$$\text{(eqall)}\ \frac{\Delta, \alpha \vdash_{\mathcal{F}} \tau_{11} = \tau_{21} \quad \Delta, \alpha \vdash_{\mathcal{F}} \tau_{12} = \tau_{22}}{\Delta \vdash_{\mathcal{F}} \forall \alpha \le \tau_{11}.\tau_{12} = \forall \alpha \le \tau_{21}.\tau_{22}} \qquad \text{(eqself)}\ \frac{\Delta, \alpha \vdash_{\mathcal{F}} \tau_1 = \tau_2}{\Delta \vdash_{\mathcal{F}} \mathsf{self}\ \alpha.\tau_1 = \mathsf{self}\ \alpha.\tau_2}$$

$$\text{(eqrec)}\ \frac{\Delta, \alpha \vdash_{\mathcal{F}} \tau_1 = \tau_2}{\Delta \vdash_{\mathcal{F}} \mathsf{rec}\ \alpha.\tau_1 = \mathsf{rec}\ \alpha.\tau_2} \qquad \text{(eq1)}\ \frac{\Delta \vdash_{\mathcal{F}} \tau}{\Delta \vdash_{\mathcal{F}} \tau = \sigma\{\alpha := \tau\}}\ (\tau = \mathsf{rec}\ \alpha.\sigma)$$

$$\text{(eq2)}\ \frac{\Delta \vdash_{\mathcal{F}} \tau\{\alpha := \sigma_1\} = \sigma_1 \quad \Delta \vdash_{\mathcal{F}} \tau\{\alpha := \sigma_2\} = \sigma_2}{\Delta \vdash_{\mathcal{F}} \sigma_1 = \sigma_2}\ (\tau \downarrow \alpha)$$

$$\boxed{\Delta; B \vdash_{\mathcal{F}} \tau_1 \le \tau_2 \quad \Delta; B \vdash_{\mathcal{F}} \tau_1^{\phi_1} \le \tau_2^{\phi_2}}$$

$$\text{(subr)}\ \frac{\Delta; B \vdash_{\mathcal{F}} \tau_1 = \tau_2}{\Delta; B \vdash_{\mathcal{F}} \tau_1 \le \tau_2} \qquad \text{(subt)}\ \frac{\Delta; B \vdash_{\mathcal{F}} \tau_1 \le \tau_2 \quad \Delta; B \vdash_{\mathcal{F}} \tau_2 \le \tau_3}{\Delta; B \vdash_{\mathcal{F}} \tau_1 \le \tau_3}$$

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

$$\text{(subfun)}\ \frac{\Delta; B \vdash_{\mathcal{F}} \sigma_1 \le \tau_1 \quad \Delta; B \vdash_{\mathcal{F}} \tau_2 \le \sigma_2}{\Delta; B \vdash_{\mathcal{F}} \tau_1 \to \tau_2 \le \sigma_1 \to \sigma_2}$$

$$\text{(subtup)}\ \frac{j \in I_2 : \Delta; B \vdash_{\mathcal{F}} \tau_j^{\phi_j} \le {\tau'}_j^{\phi'_j} \quad k \in I_1 - I_2 : \Delta \vdash_{\mathcal{F}} \tau_k}{\Delta; B \vdash_{\mathcal{F}} \langle \ell_i{:}\tau_i^{\phi_i} \rangle_{i \in I_1}^{\varphi_1} \le \langle \ell_j{:}{\tau'}_j^{\phi'_j} \rangle_{j \in I_2}^{\varphi_2}}$$

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

$$\text{(suball)}\ \frac{\Delta, \alpha \vdash_{\mathcal{F}} \tau_{11} = \tau_{21} \quad \Delta, \alpha; B, \alpha \le \tau_{11} \vdash_{\mathcal{F}} \tau_{12} \le \tau_{22}}{\Delta; B \vdash_{\mathcal{F}} \forall \alpha \le \tau_{11}.\tau_{12} \le \forall \alpha \le \tau_{21}.\tau_{22}}$$

$$\text{(subself)}\ \frac{\Delta, \alpha; B \vdash_{\mathcal{F}} \tau_1 \le \tau_2}{\Delta; B \vdash_{\mathcal{F}} \mathsf{self}\ \alpha.\tau_1 \le \mathsf{self}\ \alpha.\tau_2}$$

$$\text{(subrec)}\ \frac{\Delta, \alpha_1, \alpha_2; B, \alpha_1 \le \alpha_2 \vdash_{\mathcal{F}} \tau_1 \le \tau_2}{\Delta; B \vdash_{\mathcal{F}} \mathsf{rec}\ \alpha_1.\tau_1 \le \mathsf{rec}\ \alpha_2.\tau_2} \begin{pmatrix} \alpha_1 \notin \mathrm{ftv}(\tau_2) \\ \alpha_2 \notin \mathrm{ftv}(\tau_1) \end{pmatrix}$$

$$\text{(subcov)}\ \frac{\Delta; B \vdash_{\mathcal{F}} \tau_1 \le \tau_2}{\Delta; B \vdash_{\mathcal{F}} \tau_1^{\phi} \le \tau_2^{+}}\ (\phi \in \{+, \circ\}) \qquad \text{(subinv)}\ \frac{\Delta \vdash_{\mathcal{F}} \tau_1 = \tau_2}{\Delta; B \vdash_{\mathcal{F}} \tau_1^{\circ} \le \tau_2^{\circ}}$$

Figure 5: $\mathcal{F}_{\mathsf{self}}$ Typing Rules, Typing Level

$\boxed{\Delta; B; \Gamma \vdash_{\mathcal{F}} e : \tau}$

$$\text{(subsume)}\ \frac{\Delta; B; \Gamma \vdash_{\mathcal{F}} e : \tau_1 \quad \Delta; B \vdash_{\mathcal{F}} \tau_1 \leq \tau_2}{\Delta; B; \Gamma \vdash_{\mathcal{F}} e : \tau_2} \qquad \text{(var)}\ \frac{}{\Delta; B; \Gamma \vdash_{\mathcal{F}} x : \tau}\ (\Gamma(x) = \tau)$$

$$\text{(fun)}\ \frac{\Delta \vdash_{\mathcal{F}} \tau_1 \quad \Delta; B; \Gamma, x : \tau_1 \vdash_{\mathcal{F}} e : \tau_2}{\Delta; B; \Gamma \vdash_{\mathcal{F}} \lambda x : \tau_1.e : \tau_1 \to \tau_2} \qquad \text{(app)}\ \frac{\Delta; B; \Gamma \vdash_{\mathcal{F}} e_1 : \tau_2 \to \tau_1 \quad \Delta; B; \Gamma \vdash_{\mathcal{F}} e_2 : \tau_2}{\Delta; B; \Gamma \vdash_{\mathcal{F}} e_1\ e_2 : \tau_1}$$

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

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

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

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

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

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

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

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

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

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

Figure 6: $\mathcal{F}_{\mathsf{self}}$ Typing Rules, Expression Level

typed languages including typed target languages such as TAL [MWCG99, MCG$^+$99].

3.3 The Translation

The translation appears in Figure 7. It consists of six translation functions. At the type level $[\![\cdot]\!]_{\mathrm{type}}$, $[\![\cdot]\!]_{\mathrm{mt}}(\cdot)$, and $[\![\cdot]\!]_{\mathrm{sig}}(\cdot)$ translate types, rows, and signatures. At the term level $[\![\cdot]\!]_{\mathrm{exp}}$ and $[\![\cdot]\!]_{\mathrm{mth}}(\cdot)$ translate expressions and methods. The term level translations are type directed, that is, they translate an $\mathcal{O}$ typing derivation into an $\mathcal{F}_{\mathsf{self}}$ term. The equations in Figure 7 should be interpreted as specifying the translations of the corresponding typing rules of $\mathcal{O}$. On the right-hand side are the translations of subterms, these should be interpreted as the translations of the corresponding subderivations. There are conditions on the equations that describe the typing assumptions made about various terms, these correspond exactly to typing information in the typing rules. There are no equations for the rules (subsume) and (meth-sub), their translations are the translation of their first hypothesis. However, to deal with a subtlety in the operational correctness, the rule (subsume) can also be translated as follows: if $\Gamma \vdash_{\mathcal{O}} e : \mathsf{objt}\ r_1$ and $\vdash_{\mathcal{O}} \mathsf{objt}\ r_1 \leq \mathsf{objt}\ r_2$ are the two hypotheses and the translation of the first subderivation is $\mathsf{pack}\ e', \tau\ \mathsf{as}\ [\![\mathsf{objt}\ r_1]\!]_{\mathrm{type}}$, then the translation of the derivation is $\mathsf{pack}\ e', \tau\ \mathsf{as}\ [\![\mathsf{objt}\ r_2]\!]_{\mathrm{type}}$. In this way, Figure 7 looks like a syntax-directed translation and is not cluttered by the derivations, but actually specifies a type-directed translation. As with all type-directed translations, a source term of type τ could have many derivations and thus many translations. However, as the proven in Appendices C and D, any of these translations will have type $[\![\tau]\!]_{\mathrm{type}}$ and will simulate the source term. Furthermore, inspection of the translation shows that the erasure[4] of these translations is the same untyped term.

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, the translation of et uses a function stutter$(\cdot)$. This function simplifies the proof of operation correctness, it can be any function that has the following two properties: If $\Delta; B; \Gamma \vdash_{\mathcal{F}} e : \tau$ then $\Delta; B; \Gamma \vdash_{\mathcal{F}} \mathrm{stutter}(e) : \tau$, and $\mathrm{stutter}(v) \mapsto v$. 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

[4] The erasure of a term is obtained by removing all type annotations.

$$
\begin{array}{lcl}
\llbracket \mathsf{tempt}\ r \rrbracket_{\mathrm{type}} & = & \forall \alpha \leq \llbracket r \rrbracket_{\mathrm{full}}(\alpha).\llbracket r \rrbracket_{\mathrm{mt}}(\alpha, \circ) \\
\llbracket \mathsf{objt}\ r \rrbracket_{\mathrm{type}} & = & \mathsf{self}\ \alpha.\llbracket r \rrbracket_{\mathrm{full}}(\alpha) \\
\llbracket [m_i{:}s_i; f_j{:}\sigma_j]_{i \in I, j \in J} \rrbracket_{\mathrm{mt}}(\tau, \varphi) & = & \langle m_i{:}\llbracket s_i \rrbracket_{\mathrm{sig}}(\tau)^{+} \rangle^{\varphi}_{i \in I} \\
\llbracket r \rrbracket_{\mathrm{full}}(\tau) & = & \langle \mathsf{mt}{:}\llbracket r \rrbracket_{\mathrm{mt}}(\tau, \rightarrow)^{+}, f_j{:}\llbracket \sigma_j \rrbracket^{\circ}_{\mathrm{type}} \rangle^{\rightarrow}_{j \in J} \\
 & & \text{where } r = [m_i{:}s_i; f_j{:}\sigma_j]_{i \in I, j \in J} \\
\llbracket \sigma \rrbracket_{\mathrm{sig}}(\tau) & = & \tau \rightarrow \sigma \\
 & & \\
\llbracket x \rrbracket_{\mathrm{exp}} & = & x \\
\llbracket \mathsf{et} \rrbracket_{\mathrm{exp}} & = & \mathrm{stutter}(\Lambda \alpha \leq \langle \mathsf{mt}{:}\langle\rangle^{\rightarrow +} \rangle^{\rightarrow}.\langle\rangle) \\
\llbracket e + f{:}\tau \rrbracket_{\mathrm{exp}} & = & \Lambda \alpha \leq \llbracket r' \rrbracket_{\mathrm{full}}(\alpha).\llbracket e \rrbracket_{\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} \\
 & & \qquad r' \text{ is } [m_i{:}s_i; f_j{:}\sigma_j, f{:}\tau]_{i \in I, j \in J} \\
\llbracket e \leftarrow\!\!+ [m_i = M_i]_{i \in K} \rrbracket_{\mathrm{exp}} & = & \Lambda \alpha \leq \llbracket r' \rrbracket_{\mathrm{full}}(\alpha).(\llbracket e \rrbracket_{\mathrm{exp}}[\alpha].m_i \leftarrow_{i \in I \cap K} \llbracket M_i \rrbracket_{\mathrm{mth}}(\alpha) \\
 & & \qquad\qquad\qquad\quad +_{i \in K - I}\ m_i = \llbracket M_i \rrbracket_{\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}, \\
 & & \qquad 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 \\
\llbracket \mathsf{new}\ \ e[f_j = e_j]_{j \in J} \rrbracket_{\mathrm{exp}} & = & \mathsf{let}\ x = \llbracket e \rrbracket_{\mathrm{exp}}\ \mathsf{and}\ x_j =_{j \in J} \llbracket e_j \rrbracket_{\mathrm{exp}}\ \mathsf{in} \\
 & & \mathsf{pack}\ \langle \mathsf{mt} = x[\tau], f_j = x_j \rangle_{j \in J}, \tau\ \mathsf{as}\ \llbracket \mathsf{objt}\ r \rrbracket_{\mathrm{type}} \\
 & & \text{where } \tau = \mathsf{rec}\ \alpha.\llbracket r \rrbracket_{\mathrm{full}}(\alpha) \\
 & & \qquad e \text{ has type } \mathsf{tempt}\ r \\
 & & \qquad x \text{ and } x_j \text{ are fresh} \\
\llbracket e.m \rrbracket_{\mathrm{exp}} & = & \mathsf{unpack}\ \alpha, x = \llbracket e \rrbracket_{\mathrm{exp}}\ \mathsf{in}\ x.\mathsf{mt}.m\ x \\
 & & \text{where } \alpha \text{ and } x \text{ are fresh} \\
\llbracket e.f \rrbracket_{\mathrm{exp}} & = & \mathsf{unpack}\ \alpha, x = \llbracket e \rrbracket_{\mathrm{exp}}\ \mathsf{in}\ x.f \\
 & & \text{where } \alpha \text{ and } x \text{ are fresh} \\
\llbracket e_1.f := e_1 \rrbracket_{\mathrm{exp}} & = & \mathsf{let}\ x_1 = \llbracket e \rrbracket_{\mathrm{exp}}\ \mathsf{and}\ x_2 = \llbracket e_2 \rrbracket_{\mathrm{exp}}\ \mathsf{in} \\
 & & \mathsf{unpack}\ \alpha, x = x_1\ \mathsf{in}\ \mathsf{pack}\ x.f := x_2, \alpha\ \mathsf{as}\ \llbracket \mathsf{objt}\ r \rrbracket_{\mathrm{type}} \\
 & & \text{where } e_1 \text{ has type } \mathsf{objt}\ r \text{ and } \alpha \text{ and } x \text{ are fresh} \\
\llbracket x.e{:}\sigma \rrbracket_{\mathrm{mth}}(\tau) & = & \lambda x'{:}\tau.\mathsf{let}\ x = \mathsf{pack}\ x', \tau\ \mathsf{as}\ \llbracket \tau' \rrbracket_{\mathrm{type}}\ \mathsf{in}\ \llbracket e \rrbracket_{\mathrm{exp}} \\
 & & \text{where } x.e{:}\sigma \text{ has type } \tau' \rhd \sigma \text{ and } x' \text{ is fresh} \\
\llbracket \mathsf{temp}[m_i = M_i; f_j{:}\sigma_j]_{i \in I, j \in J} \rrbracket_{\mathrm{exp}} & = & \Lambda \alpha \leq \llbracket r \rrbracket_{\mathrm{full}}(\alpha).\langle m_i = \llbracket M_i \rrbracket_{\mathrm{mth}}(\alpha) \rangle_{i \in I} \\
 & & \text{where } \mathsf{temp}[m_i = M_i; f_j{:}\sigma_j]_{i \in I, j \in J} \text{ has type } \mathsf{tempt}\ r \\
\llbracket \mathsf{obj}[m_i = M_i; f_j = v_j]_{i \in I, j \in J} \rrbracket_{\mathrm{exp}} & = & \mathsf{pack}\ \langle \mathsf{mt} = \langle m_i = \llbracket M_i \rrbracket_{\mathrm{mth}}(\tau) \rangle_{i \in I}, f_j = \llbracket v \rrbracket_{\mathrm{exp}} \rangle_{j \in J}, \tau \\
 & & \ \mathsf{as}\ \llbracket \mathsf{objt}\ r \rrbracket_{\mathrm{type}} \\
 & & \text{where } \mathsf{obj}[m_i = M_i; f_j = v_j]_{i \in I, j \in J} \text{ has type } \mathsf{objt}\ r \text{ and} \\
 & & \qquad \tau = \mathsf{rec}\ \alpha.\llbracket r \rrbracket_{\mathrm{full}}(\alpha)
\end{array}
$$

Figure 7: The Translation

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, as proven in Appendices C and D.

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:

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

To encode these method parameters $\mathcal{F}_{\mathsf{self}}$ is extended with multiargument functions.[5] Using $(\tau_1, \ldots, \tau_n) \to \tau$ for multiargument 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:

$$\begin{array}{lcl} \llbracket(\tau_1, \ldots, \tau_n) \to \tau\rrbracket_{\mathrm{sig}}(\sigma) & = & (\sigma, \llbracket\tau_1\rrbracket_{\mathrm{type}}, \ldots, \llbracket\tau_n\rrbracket_{\mathrm{type}}) \to \llbracket\tau\rrbracket_{\mathrm{type}} \\ \llbracket e.m(e_1, \ldots, e_n)\rrbracket_{\mathrm{exp}} & = & \mathsf{unpack}\ \alpha, x = \llbracket e\rrbracket_{\mathrm{exp}}\ \mathsf{in} \\ & & x.\mathsf{mt}.m(x, \llbracket e_1\rrbracket_{\mathrm{exp}}, \ldots, \llbracket e_n\rrbracket_{\mathrm{exp}}) \\ \llbracket x(x_1 : \tau_1, \ldots, x_n : \tau_n).e : \tau\rrbracket_{\mathrm{mth}}(\sigma) & = & \lambda(x' : \sigma, x_1 : \llbracket\tau_1\rrbracket_{\mathrm{type}}, \ldots, x_n : \llbracket\tau_n\rrbracket_{\mathrm{type}}). \\ & & \mathsf{let}\ x = \mathsf{pack}\ x', \sigma\ \mathsf{as}\ \llbracket\tau'\rrbracket_{\mathrm{type}}\ \mathsf{in}\ \llbracket e\rrbracket_{\mathrm{exp}} \\ & & \text{where } x(x_1 : \tau_1, \ldots, x_n : \tau_n).e : \tau \text{ has type } \tau' \rhd \tau \end{array}$$

[5] 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].

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

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

$$\begin{array}{lcl} [\![[\alpha_1 \leq \tau_1, \ldots, \alpha_n \leq \tau_n]\tau]\!]_{\mathrm{sig}}(\sigma) & = & \forall \alpha_1 \leq [\![\tau_1]\!]_{\mathrm{type}} . \ldots \forall \alpha_n \leq [\![\tau_n]\!]_{\mathrm{type}} . \sigma \to \tau \\ [\![e.m[\tau_1, \ldots, \tau_n]]\!]_{\mathrm{exp}} & = & \mathsf{unpack}\ \alpha, x = [\![e]\!]_{\mathrm{exp}}\ \mathsf{in} \\ & & x.\mathsf{mt}.m[[\![\tau_1]\!]_{\mathrm{type}}] \cdots [[\![\tau_n]\!]_{\mathrm{type}}]\ x \\ [\![x[\alpha_1 \leq \tau_1, \ldots, \alpha_n \leq \tau_n].e : \tau]\!]_{\mathrm{mth}}(\sigma) & = & \Lambda \alpha_1 \leq [\![\tau_1]\!]_{\mathrm{type}} . \ldots \Lambda \alpha_n \leq [\![\tau_n]\!]_{\mathrm{type}} . \lambda x' : \sigma . \\ & & \mathsf{let}\ x = \mathsf{pack}\ x', \sigma\ \mathsf{as}\ [\![\tau']\!]_{\mathrm{type}}\ \mathsf{in}\ [\![e]\!]_{\mathrm{exp}} \\ & & \text{where the method body has type } \tau' \rhd \tau \end{array}$$

Covariant self types can be incorporated into the source language and translation. The details are quite messy and not as clean as the basic translation or other extensions considered in this section, Appendix E contains them. The 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:

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

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:

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

where $e + f{:}\sigma^\phi$ and $e.f := \sigma^\phi$ have 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.

The proof of correctness of the translation can easily be extended to all of the extensions mentioned so far. The proof of soundness of the target language is easily extended to multiargument functions. 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. Details appear in Appendix F.

The templates in O allow only single inheritance and concrete methods. Next, 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, $e.m := M$; both operations are typed checked with self being the object type corresponding to the assumption row in the template. 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 following type translation:

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

Here are the relevant parts of the term translation. The rest of the term translation is the same.

$$\begin{array}{lcl}
[\![e := r]\!]_{\text{exp}} & = & \forall\alpha \le [\![r]\!]_{\text{full}}(\alpha).[\![e]\!]_{\text{exp}}[\alpha] \\
[\![e + m = M]\!]_{\text{exp}} & = & \forall\alpha \le [\![r]\!]_{\text{full}}(\alpha).[\![e]\!]_{\text{exp}}[\alpha] + m = [\![M]\!]_{\text{mth}}(\alpha) \\
 & & \text{where } e \text{ has type } \mathsf{tempt}(r; m_i{:}s_i)_{i\in I} \\
[\![e.m := M]\!]_{\text{exp}} & = & \forall\alpha \le [\![r]\!]_{\text{full}}(\alpha).[\![e]\!]_{\text{exp}}[\alpha].m \leftarrow [\![M]\!]_{\text{mth}}(\alpha) \\
 & & \text{where } e \text{ has type } \mathsf{tempt}(r; m_i{:}s_i)_{i\in I}
\end{array}$$

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.

Finally, I remark on some other future work. Object-oriented languages often have other modularity features like final methods, final classes, and protection modifiers (private, public, protected). Finality of methods could be ignored by the translation, but then the translation would not be fully abstract. A finality preserving translation might be possible by using singleton types for the final methods. League *et al.* [LST99] describe techniques for translating Java's class private and public modifiers using combinations of recursive and existential types. Their ideas or similar ones might be applicable to the translation described in this paper. Instance private or protected fields might be translated using junk types to hide the fields in an object's public type.

As well as correctness, another desirable property of translations is full abstractness [Aba98]. For example, if the types of the target language were used for security purposes as types are used in the Java Virtual Machine, then full abstractness would mean that a security monitor writer could reason in terms of object abstractions rather than in terms of target language abstractions. I conjecture that the basic encoding and all the extensions in this section except method update are fully abstract. Future work should attempt to prove this property.

To use this encoding in a type-directed compiler it is desirable that it not impede optimisations on the target language. An important optimisation for object-oriented languages is turning dynamic dispatch into static dispatch. Examples of such optimisations are Class Hierarchy Analysis [DGC95] and Rapid Type Analysis [Bac97]. I believe that an optimisation based on a monovariant analysis of this sort would not be impeded by the type system of $\mathcal{F}_{\mathsf{self}}$. Future work should attempt to verify this and investigate whether other optimisations are affected by $\mathcal{F}_{\mathsf{self}}$'s type system.

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[6] 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 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

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

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.

References

[Aba98] Martín Abadi. Protection in programming-language translations. In *25th International Colloquim on Automata, Languages, and Programming*, July 1998.

[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.

[Bac97] David Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, U.C. Berkeley, October 1997.

[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.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Aarhus, Denmark, August 1995. Springer-Verlag.

[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. Object closure conversion. Technical Report TR99-1763, Department of Computer Science, Cornell University, 4130 Upson Hall, Ithaca, NY 14853-7501, USA, August 1999.

[Gle99c] 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. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Minneapolis, MN, USA, October 2000. ACM Press.

[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.

A Type Soundness of Template Language

A context Γ_1 is stronger than Γ_2 if $\mathrm{dom}(\Gamma_1) \supseteq \mathrm{dom}(\Gamma_2)$ and $\forall x \in \mathrm{dom}(\Gamma_2) :\ \vdash_{\mathcal{O}} \Gamma_1(x) \leq \Gamma_2(x)$.

Lemma A.1 (Context Strengthening) *If Γ_1 is stronger than Γ_2 and $\Gamma_1 \vdash_{\mathcal{O}} e : \tau$ then $\Gamma_2 \vdash_{\mathcal{O}} e : \tau$. If Γ_1 is stronger than Γ_2, $\Gamma_1 \vdash^{\mathsf{M}}_{\mathcal{O}} M : \tau \rhd s$, and $\vdash_{\mathcal{O}} \tau' \leq \tau$ then $\Gamma_2 \vdash^{\mathsf{M}}_{\mathcal{O}} M : \tau' \rhd s$ without using the rule (meth-sub).*

Proof: By induction on the derivation and inspection of the rules. □

A corollary of this lemma is that (meth-sub) is derivable. I included this rule to make proving operational correctness easier. Throughout the rest of this appendix, all derivations will not use the rule (meth-sub).

Lemma A.2 (Value Substitution) *If $\Delta; B; \Gamma, x{:}\tau_1 \vdash_{\mathcal{O}} e_2 : \tau_2$ and $\Delta; B; \Gamma \vdash_{\mathcal{O}} e_1 : \tau_1$ then $\Delta; B; \Gamma \vdash_{\mathcal{O}} e_2\{x := e_1\} : \tau_2$.*

Proof: By induction on the derivation and inspection of the rules. □

Lemma A.3 *If $\emptyset \vdash_{\mathcal{O}} E\{e\} : \tau_1$ then there exists τ_2 such that $\emptyset \vdash_{\mathcal{O}} e : \tau_2$, where the last rule used is not (subsume), and for all e' such that $\emptyset \vdash_{\mathcal{O}} e' : \tau_2$, $\emptyset \vdash_{\mathcal{O}} E\{e'\} : \tau_1$.*

Proof: By induction on E and inspection of the typing rules. □

Lemma A.4 (Decomposition) *An expresison e is either a value or of the form $E\{\iota\}$ for:*

$$\iota ::= x \mid \mathsf{et} \mid v + f{:}\sigma \mid v \leftarrow\!\!+[m_i = M_I]_{i \in I} \mid \mathsf{new}\ \ v[v_j = v_j]_{j \in J} \mid v.m \mid v.f \mid v_1.f := v_2$$

Proof: By induction of the structure of e and inspection of the forms for e, E, and values. □

Lemma A.5 (Canonical Forms) *If $\emptyset \vdash_{\mathcal{O}} v : \tau$ then:*

v	τ
$\mathsf{temp}[m_i = M_i; f_j{:}\sigma_j^{\phi_j}]_{i \in I, j \in J}$	$\mathsf{temp}[m_i{:}s_i; f_j{:}\sigma_j^{\phi_j}]_{i \in I, j \in J}$
$\mathsf{obj}[m_i = M_i; f_j = v_j]_{i \in I_1, j \in J_1}$	$\mathsf{obj}[m_i{:}s_i; f_j{:}\sigma_j^{\phi_j}]_{i \in I_2, j \in J_2}$

where in the last case I_2 is a prefix of I_1 and J_2 is a prefix of J_1.

Proof: By induction on the derivation of $\emptyset \vdash_{\mathcal{O}} v : \tau$. There are only three rules that can deduce $\emptyset \vdash_{\mathcal{O}} v : \tau$: (temp), (obj), and (subsume). The first two rules clearly satisfy the conditions, and inspection of the subtyping rules reveals that (subsume) satisfies the conditions also. □

Theorem A.6 (Type Preservation) *If $\emptyset \vdash_{\mathcal{O}} e_1 : \tau$ and $e_1 \mapsto e_2$ then $\emptyset \vdash_{\mathcal{O}} e_2 : \tau$.*

Proof: By the operational semantics there is an E, ι, and e such that $e_1 = E\{\iota\}$ and $e_2 = E\{e\}$. By Lemma A.3 there is a σ such that $\emptyset \vdash_{\mathcal{O}} \iota : \sigma$ (where the last rule used is not (subsume)) and the result follows if $\emptyset \vdash_{\mathcal{O}} e : \sigma$. Throughout this proof, I will use the following abbreviations:

$$\begin{array}{rcl}
v1 & = & \mathsf{temp}[m_i = M_i; f_j{:}\sigma_j]_{i \in I, j \in J} \\
v2 & = & \mathsf{obj}[m_i = M_i; f_j = w_j]_{i \in I, j \in J} \\
M_i & = & x_i.b_i{:}\tau_i \\
r & = & [m_i{:}s_i; f_j{:}\sigma_j]_{i \in I, j \in J}
\end{array}$$

Consider the cases for ι and e as given in Figure 2:

$\iota = \mathsf{et}$**:** In this case $e = \mathsf{temp}[;]$. The derivation of $\emptyset \vdash_{\mathcal{O}} \iota : \sigma$ must be the rule (et) so $\sigma = \mathsf{tempt}[;]$. The result follows by (temp).

$\iota = v_1 + f = \sigma$**:** In this case $e = \mathsf{temp}[m_i = M_i; f_j{:}\sigma_j, f{:}\sigma]_{i \in I, j \in J}$. The derivation of $\emptyset \vdash_{\mathcal{O}} \iota : \sigma$ must have the form:

$$\dfrac{\dfrac{\dfrac{\emptyset \vdash^{\mathsf{M}}_{\mathcal{O}} M_i : \mathsf{objt}\ r \rhd s_i}{\emptyset \vdash_{\mathcal{O}} v_1 : \mathsf{tempt}\ r} \qquad \vdash_{\mathcal{O}} \mathsf{tempt}\ r \leq \mathsf{tempt}\ r}{\emptyset \vdash_{\mathcal{O}} v_1 : \mathsf{tempt}\ r}}{\emptyset \vdash_{\mathcal{O}} \iota : \mathsf{tempt}\ r'}$$

where $r' = [m_i{:}s_i; f_j{:}\sigma_j, f{:}\sigma]_{i \in I, j \in J}$. Clearly $\vdash_{\mathcal{O}} \mathsf{objt}\ r' \leq \mathsf{objt}\ r$ so by Context Strengthening $\emptyset \vdash^{\mathsf{M}}_{\mathcal{O}} M_i : \mathsf{objt}\ r' \rhd s_i$ and $\emptyset \vdash_{\mathcal{O}} e : \mathsf{tempt}\ r'$ by (temp) as required.

$\iota = v_1 \leftarrow\!\!+[m_k = M_k]_{k \in K}$: In this case $e = \mathsf{temp}[m_l = M_l; f_j{:}\sigma_j]_{i \in (I,K-I), j \in J}$. The derivation of $\emptyset \vdash_{\mathcal{O}} \iota : \sigma$ must have the form:

$$\dfrac{\dfrac{\dfrac{\emptyset \vdash^{\mathsf{M}}_{\mathcal{O}} M_i : \mathsf{objt}\ r \rhd s_i}{\emptyset \vdash_{\mathcal{O}} v_1 : \mathsf{tempt}\ r} \qquad \vdash_{\mathcal{O}} \mathsf{tempt}\ r \leq \mathsf{tempt}\ r}{\emptyset \vdash_{\mathcal{O}} v_1 : \mathsf{tempt}\ r} \qquad \emptyset \vdash^{\mathsf{M}}_{\mathcal{O}} M_i : \mathsf{objt}\ r' \rhd s'_i \quad \vdash_{\mathcal{O}} s'_i \leq s_i}{\emptyset \vdash_{\mathcal{O}} \iota : \mathsf{tempt}\ r'}$$

where $r' = [m_i{:}s''_i; f_j{:}\sigma_j]_{i \in (I,K-I), j \in J}$, $s''_i = s'_i$ if $i \in K$, and $s''_i = s_i$ if $i \in I - K$. Clearly $\vdash_{\mathcal{O}} \mathsf{objt}\ r' \leq \mathsf{objt}\ r$ so by Context Strengthening $\emptyset \vdash^{\mathsf{M}}_{\mathcal{O}} M_i : \mathsf{objt}\ r' \rhd s_i$ for $i \in I - K$. By (temp), $\emptyset \vdash_{\mathcal{O}} e : \mathsf{tempt}\ r'$ as required.

$\iota = \mathsf{new}\ \ v_1[f_j = w_j]_{j \in J}$: In this case $e = v2$. The derivation of $\emptyset \vdash_{\mathcal{O}} \iota : \sigma$ must have the form:

$$\dfrac{\dfrac{\dfrac{\emptyset \vdash^{\mathsf{M}}_{\mathcal{O}} M_i : \mathsf{objt}\ r \rhd s_i}{\emptyset \vdash_{\mathcal{O}} v_1 : \mathsf{tempt}\ r} \qquad \vdash_{\mathcal{O}} \mathsf{tempt}\ r \leq \mathsf{tempt}\ r}{\emptyset \vdash_{\mathcal{O}} v_1 : \mathsf{tempt}\ r} \qquad \emptyset \vdash_{\mathcal{O}} w_j : \sigma_j}{\emptyset \vdash_{\mathcal{O}} \iota : \mathsf{objt}\ r}$$

The result follows by (obj).

$\iota = v_2.m_k$: In this case $e = b_k\{x_k := v_2\}$ and $k \in I$. The derivation of $\emptyset \vdash_{\mathcal{O}} \iota : \sigma$ must have the form:

$$\dfrac{\dfrac{\dfrac{\emptyset \vdash^{\mathsf{M}}_{\mathcal{O}} M_i : \mathsf{objt}\ r \rhd s_i \quad \cdots}{\emptyset \vdash_{\mathcal{O}} v_1 : \mathsf{objt}\ r} \qquad \vdash_{\mathcal{O}} \mathsf{objt}\ r \leq \mathsf{objt}\ r'}{\emptyset \vdash_{\mathcal{O}} v_1 : \mathsf{objt}\ r'}}{\emptyset \vdash_{\mathcal{O}} \iota : s'_k}$$

where $r' = [m_i{:}s'_i; f_j{:}\sigma_j]_{i \in I', j \in J'}$, I a prefix of I', and $\vdash_{\mathcal{O}} s_i \leq s'_i$ for $i \in I'$. The derivation of $\emptyset \vdash^{\mathsf{M}}_{\mathcal{O}} M_k : \mathsf{objt}\ r \rhd s_i$ must include $x_k{:}\mathsf{objt}\ r \vdash_{\mathcal{O}} b_k : \tau_k$ and note that $\vdash_{\mathcal{O}} \tau_k \leq s'_k$. By Value Substitution $\vdash_{\mathcal{O}} b_k\{x_k := v2\} : \tau_k$ and the result follows by subsumption.

$\iota = v_2.f_k$: In this case $e = w_j$ and $k \in J$. The derivation of $\emptyset \vdash_{\mathcal{O}} \iota : \sigma$ must have the form:

$$\dfrac{\dfrac{\dfrac{\cdots \quad \emptyset \vdash_{\mathcal{O}} w_j : \sigma_j}{\emptyset \vdash_{\mathcal{O}} v_1 : \mathsf{objt}\ r} \qquad \vdash_{\mathcal{O}} \mathsf{objt}\ r \leq \mathsf{objt}\ r'}{\emptyset \vdash_{\mathcal{O}} v_1 : \mathsf{objt}\ r'}}{\emptyset \vdash_{\mathcal{O}} \iota : \sigma_k}$$

where $r' = [m_i{:}s'_i; f_j{:}\sigma_j]_{i \in I', j \in J'}$ and J a prefix of J'. The result is immediate from $\emptyset \vdash_{\mathcal{O}} w_j : \sigma_j$ and $k \in J$.

$\iota = v_2.f_k := w$: In this case $e = \mathsf{obj}[m_i = M_i; f_j = w'_j]_{i \in I, j \in J}$, $k \in J$, $w'_i = w_i$ if $i \neq k$, and $w'_k = w$. The derivation of $\emptyset \vdash_{\mathcal{O}} \iota : \sigma$ must have the form:

$$\dfrac{\dfrac{\dfrac{\emptyset \vdash^{\mathsf{M}}_{\mathcal{O}} M_i : \mathsf{objt}\ r \rhd s_i \quad \emptyset \vdash_{\mathcal{O}} w_j : \sigma_j}{\emptyset \vdash_{\mathcal{O}} v_1 : \mathsf{objt}\ r} \qquad \vdash_{\mathcal{O}} \mathsf{objt}\ r \leq \mathsf{objt}\ r'}{\emptyset \vdash_{\mathcal{O}} v_1 : \mathsf{objt}\ r'} \qquad \emptyset \vdash_{\mathcal{O}} w : \sigma_k}{\emptyset \vdash_{\mathcal{O}} \iota : \sigma_k}$$

where $r' = [m_i{:}s'_i; f_j{:}\sigma_j]_{i \in I', j \in J'}$ and J a prefix of J'. The result follows by (obj).

□

Theorem A.7 (Progress) *If $\emptyset \vdash_{\mathcal{O}} e : \tau$ then either e is a value or there exists e' such that $e \mapsto e'$.*

Proof: By Decomposition there exists E and ι of the form in that lemma such that $e = E\{\iota\}$. By Lemma A.3 there is σ such that $\emptyset \vdash_{\mathcal{O}} \iota : \sigma$. Consider the various cases for ι:

$\iota = x$**:** Variables are untypable in the empty context.

$\iota = \mathsf{et}$**:** Then $\iota \mapsto \mathsf{temp}[;]$.

$\iota = v + f = \sigma$**:** The last rule used to type ι must be (addfield), so v has a template type and f is not one of its fields. By Canonical Forms, v is a template value and f is not one of its fields. Thus ι satisfies the second case in the operational semantics.

$\iota = v \leftarrow\!\!+[m_k = M_k]_{k \in K}$**:** The last rule used to type ι must be (aometh), so v has a template type. By Canonical Forms, v is a template value. Thus ι satisfies the third case in the operational semantics.

$\iota = \mathsf{new}\ v[f_j = v_j]_{j \in J}$**:** The last rule used to type ι must be (inst), so v has a template type with fields $f_{j \in J}$. By Canonical Forms, v is a template value with fields $f_{j \in J}$. Thus ι satisfies the fourth case in the operational semantics.

$\iota = v.m_k$**:** The last rule used to type ι must be (invoke), so v has an object type with m_k one of its methods. By Canonical Forms, v is an object value with an m_k method. Thus ι satisfies the fifth case in the operational semantics.

$\iota = v.f_j$**:** The last rule used to type ι must be (select), so v has an object type with f_k one of its fields. By Canonical Forms, v is an object value with an f_k field. Thus ι satisfies the sixth case in the operational semantics.

$\iota = v.f_j := w$**:** The last rule used to type ι must be (update), so v has an object type with f_k one of its fields. By Canonical Forms, v is an object value with an f_k field. Thus ι satisfies the seventh case in the operational semantics.

□

Theorem A.8 (Type Soundness) *If $\emptyset \vdash_{\mathcal{O}} e : \tau$ and $e \mapsto^* e'$ then e' is not stuck.*

Proof: By induction on the length of $e \mapsto^* e'$, Type Preservation, and Progress. □

B Type Soundness of Target Language

Lemma B.1 (Derived Judgements)

- $\Delta \vdash_{\mathcal{F}} \tau$ *if and only if* $\mathrm{ftv}(\tau) \subseteq \Delta$.
- $\Delta \vdash_{\mathcal{F}} \tau_1 = \tau_2$ *implies* $\mathrm{ftv}(\tau_1) \cup \mathrm{ftv}(\tau_2) \subseteq \Delta$.
- $\Delta \vdash_{\mathcal{F}} \tau$ *implies* $\Delta \vdash_{\mathcal{F}} \tau = \tau$ *by only the rules (eqtv), (eqfun), (eqtup), (eqall), (eqself), and (eqrec).*
- $\Delta \vdash_{\mathcal{F}} B$ *if and only if* $\mathrm{ftv}(\mathrm{ran}(B)) \subseteq \Delta$ *and* $\mathrm{dom}(B) \subseteq \Delta$.
- $\Delta \vdash_{\mathcal{F}} \Gamma$ *if and only if* $\mathrm{ftv}(\mathrm{ran}(\Gamma)) \subseteq \Delta$.
- *If* $\Delta \vdash_{\mathcal{F}} \tau_1 = \tau_2$ *then* $\Delta \vdash_{\mathcal{F}} \tau_i$.
- *If* $\Delta \vdash_{\mathcal{F}} B$ *and* $\Delta; B \vdash_{\mathcal{F}} \tau_1 \leq \tau_2$ *then* $\Delta \vdash_{\mathcal{F}} \tau_i$.
- *If* $\Delta \vdash_{\mathcal{F}} B$ *and* $\Delta; B \vdash_{\mathcal{F}} \tau_1^{\phi_1} \leq \tau_2^{\phi_2}$ *then* $\Delta \vdash_{\mathcal{F}} \tau_i$.
- *If* $\Delta \vdash_{\mathcal{F}} B$, $\Delta \vdash_{\mathcal{F}} \Gamma$, *and* $\Delta; B; \Gamma \vdash_{\mathcal{F}} e : \tau$ *then* $\Delta \vdash_{\mathcal{F}} \tau$.

Proof: By induction on the structure of the derivation and inspection of the typing rules, or by induction on the structure of τ. □

Definition B.1 *A context* $\Delta_1; B_1; \Gamma_1$ *is stronger than a context* $\Delta_2; B_2; \Gamma_2$ *exactly when* $\Delta_1 \supseteq \Delta_2$, $\mathrm{dom}(B_1) \supseteq \mathrm{dom}(B_2)$, $\Delta_1; B_1 \vdash_{\mathcal{F}} B_1(\alpha) \leq B_2(\alpha)$ *for all* $\alpha \in \mathrm{dom}(B_2)$, $\mathrm{dom}(\Gamma_1) \supseteq \mathrm{dom}(\Gamma_2)$, *and* $\Delta_1; B_1 \vdash_{\mathcal{F}} \Gamma_1(x) \leq \Gamma_2(x)$ *for all* $x \in \mathrm{dom}(\Gamma_2)$.

Lemma B.2 (Context Strengthening) *If J is derivable then J with a stronger context is derivable.*

Proof: By induction of the structure of the derivation and inspection of the typing rules. Note that this lemma uses the fact that α-variants of binding forms can always be chosen so as not to clash with the strengthened context. □

Lemma B.3 (Type Substitution)

- *If* $\Delta, \alpha \vdash_{\mathcal{F}} \sigma$ *and* $\Delta \vdash_{\mathcal{F}} \tau$ *then* $\Delta \vdash_{\mathcal{F}} \sigma\{\alpha := \tau\}$.
- *If* $\Delta, \alpha \vdash_{\mathcal{F}} \sigma_1 = \sigma_2$ *and* $\Delta \vdash_{\mathcal{F}} \tau$ *then* $\Delta \vdash_{\mathcal{F}} \sigma_1\{\alpha := \tau\} = \sigma_2\{\alpha := \tau\}$.
- *If* $\alpha \notin \mathrm{dom}(B)$, $\Delta, \alpha; B \vdash_{\mathcal{F}} \sigma_1 \leq \sigma_2$, *and* $\Delta \vdash_{\mathcal{F}} \tau$ *then* $\Delta; B\{\alpha := \tau\} \vdash_{\mathcal{F}} \sigma_1\{\alpha := \tau\} \leq \sigma_2\{\alpha := \tau\}$.

- *If* $\alpha \notin \mathrm{dom}(B)$, $\Delta, \alpha; B; \Gamma \vdash_{\mathcal{F}} e : \sigma$, *and* $\Delta \vdash_{\mathcal{F}} \tau$ *then* $\Delta; B\{\alpha := \tau\}; \Gamma\{\alpha := \tau\} \vdash_{\mathcal{F}} e\{\alpha := \tau\} : \sigma\{\alpha := \tau\}$.
- *If* $\Delta, \alpha; B, \alpha \leq \tau_2 \vdash_{\mathcal{F}} \sigma_1 \leq \sigma_2$ *and* $\Delta; B\{\alpha := \tau_1\} \vdash_{\mathcal{F}} \tau_1 \leq \tau_2\{\alpha := \tau_1\}$ *then:*
$$\Delta; B\{\alpha := \tau_1\} \vdash_{\mathcal{F}} \sigma_1\{\alpha := \tau_1\} \leq \sigma_2\{\alpha := \tau_1\}$$
- *If* $\Delta, \alpha; B, \alpha \leq \tau_2; \Gamma \vdash_{\mathcal{F}} e : \tau$ *and* $\Delta; B\{\alpha := \tau_1\} \vdash_{\mathcal{F}} \tau_1 \leq \tau_2\{\alpha := \tau_1\}$ *then:*
$$\Delta; B\{\alpha := \tau_1\}; \Gamma\{\alpha := \tau_1\} \vdash_{\mathcal{F}} e\{\alpha := \tau_1\} : \tau\{\alpha := \tau_1\}$$

Proof: The first result follows from Derived Judgements and properties of $\mathrm{ftv}(\cdot)$. The other results are by induction of the structure of the derivations of $\Delta, \alpha \vdash_{\mathcal{F}} \tau_1 = \tau_2$, $\Delta, \alpha; B \vdash_{\mathcal{F}} \sigma_1 \leq \sigma_2$, $\Delta, \alpha; B, \alpha \leq \tau_2 \vdash_{\mathcal{F}} \sigma_1 \leq \sigma_2$, $\Delta, \alpha; B; \Gamma \vdash_{\mathcal{F}} e : \sigma$, and $\Delta, \alpha; B, \alpha \leq \tau_2; \Gamma \vdash_{\mathcal{F}} e : \tau$, inspection of the typing rules, and Context Strengthening. □

Lemma B.4 *If* $\Delta \vdash_{\mathcal{F}} \tau_1 = \tau_2$ *and* $\Delta, \alpha \vdash_{\mathcal{F}} \sigma_1 = \sigma_2$ *then* $\Delta \vdash_{\mathcal{F}} \sigma_1\{\alpha := \tau_1\} = \sigma_2\{\alpha := \tau_2\}$.

Proof: By induction of the structure of the derivation of $\Delta, \alpha \vdash_{\mathcal{F}} \sigma_1 = \sigma_2$. In the case of (eqtv) the derivation of $\Delta \vdash_{\mathcal{F}} \tau_1 = \tau_2$ is used. In the other cases the result follows from the induction hypothesis. □

Definition B.2 *The normal form of a type* τ, *written* $\mathrm{nf}(\tau)$, *is given by:*

τ	$\mathrm{nf}(\tau)$	*Side Conditions*
$\mathsf{rec}\ \alpha.\tau'$	$\mathrm{nf}(\tau'\{\alpha := \mathsf{rec}\ \alpha.\tau'\})$	$\tau' \downarrow \alpha$
$\mathsf{rec}\ \alpha.\tau'$	$\mathsf{rec}\ \alpha.\alpha$	*not* $\tau' \downarrow \alpha$
τ	*otherwise*	

The normal form outermost constructor of τ, *written* $\mathrm{nfoc}(\tau)$, *is defined by:*

$\mathrm{nf}(\tau)$	$\mathrm{nfoc}(\tau)$
α	α
$\tau_1 \rightarrow \tau_2$	$\rightarrow$
$\langle \ell{:}\tau_i^{\phi_i} \rangle_{i \in I}^{\varphi}$	$\langle\rangle$
$\forall \alpha \leq \tau_1.\tau_2$	$\forall$
$\mathsf{self}\ \alpha.\tau'$	self
$\mathsf{rec}\ \alpha.\alpha$	$\perp$

Lemma B.5 *If* $\tau \downarrow \alpha$ *then* τ *has the form* $\mathsf{rec}\ \beta_1 \ldots . \mathsf{rec}\ \beta_n.\tau'$ *for* τ' *not a recursive type nor one of the type variables* β_1, …, β_n, *or* α. *If not* $\tau \downarrow \alpha$ *then* τ *has the form* $\mathsf{rec}\ \beta_1 \ldots . \mathsf{rec}\ \beta_n.\alpha$ *for some* $\beta_i \neq \alpha$.

Proof: By induction on the structure of τ and inspection of the definition of contractiveness. □

Thus every type has a unique normal form and normal form outermost constructor.

Lemma B.6 *If* $\sigma \downarrow \alpha$ *then* $\mathrm{nf}(\sigma\{\alpha := \tau\}) = \mathrm{nf}(\sigma)\{\alpha := \tau\}$ *and* $\mathrm{nfoc}(\sigma\{\alpha := \tau\}) = \mathrm{nfoc}(\sigma)$*.*

Proof: By Lemma B.5 σ has the form $\mathsf{rec}\ \beta_1. \ldots \mathsf{rec}\ \beta_n.\sigma'$ for σ' not a recursive type nor one of the type variables α or $\beta_1, \ldots, \beta_n$. By induction on n:

$$\begin{array}{lcl} \mathrm{nf}(\sigma\{\alpha := \tau\}) & = & \sigma'\{\alpha := \tau\}\{\beta_n := \mathsf{rec}\ \beta_n.\sigma\{\alpha := \tau\}\} \cdots \{\beta_1 := \sigma\{\alpha := \tau\}\} \\ \mathrm{nf}(\sigma) & = & \sigma'\{\beta_n := \mathsf{rec}\ \beta_n.\sigma\} \cdots \{\beta_1 := \sigma\} \end{array}$$

In the right hand side of the first equation, the substitution $\alpha := \tau$ can be pulled out to the end, giving $\mathrm{nf}(\sigma)\{\alpha := \tau\}$ as required for the first conclusion. The second conclusion follows since the outermost form of σ is independent of α. □

Lemma B.7 *If* $\Delta \vdash_{\mathcal{F}} \tau$ *then* $\Delta \vdash_{\mathcal{F}} \tau = \mathrm{nf}(\tau)$*,* $\Delta \vdash_{\mathcal{F}} \mathrm{nf}(\tau) = \tau$*,* $\Delta \vdash_{\mathcal{F}} \tau \leq \mathrm{nf}(\tau)$*, and* $\Delta \vdash_{\mathcal{F}} \mathrm{nf}(\tau) \leq \tau$*.*

Proof: For the first item, if τ is not a recursive type the result is immediate. Otherwise τ has the form $\mathsf{rec}\ \beta_1. \ldots \mathsf{rec}\ \beta_n.\tau'$ for τ' not a recursive type. Case 1, $\tau' = \beta_i$: By induction on $1 \leq j \leq i$, $\mathrm{nf}(\mathsf{rec}\ \beta_j. \ldots \mathsf{rec}\ \beta_n.\tau') = \mathrm{nf}(\mathsf{rec}\ \beta_i. \ldots \mathsf{rec}\ \beta_n.\tau')$. By Lemma B.5, not $\mathsf{rec}\ \beta_{i+1}. \ldots \mathsf{rec}\ \beta_n.\tau' \downarrow \beta_i$, so $\mathrm{nf}(\mathsf{rec}\ \beta_i. \ldots \mathsf{rec}\ \beta_n.\tau') = \mathsf{rec}\ \beta_i.\beta_i$. By induction on $i < j \leq n$, Derived Judgements, and rules (eqtv), (eq1), and (eqt), $\Delta, \beta_i \vdash_{\mathcal{F}} \mathsf{rec}\ \beta_j. \ldots \mathsf{rec}\ \beta_n.\tau' = \beta_i$. By (eqrec), $\Delta \vdash_{\mathcal{F}} \mathsf{rec}\ \beta_i. \ldots \mathsf{rec}\ \beta_n.\tau' = \mathsf{rec}\ \beta_i.\beta_i$. By induction on n, Derived Judgements, and rules (eq1) and (eqt), $\Delta \vdash_{\mathcal{F}} \tau = \mathsf{rec}\ \beta_i.\beta_i$ as required. Case 2, $\tau' \notin \{\beta_1, \ldots, \beta_n\}$: By induction on n, Derived Judgements, and rules (eq1) and (eqt), $\Delta \vdash_{\mathcal{F}} \tau = \tau'\{\beta_n := \mathsf{rec}\ \beta_n.\tau'\} \cdots \{\beta_1 := \tau\}$. By induction on n, $\mathrm{nf}(\tau) = \tau'\{\beta_n := \mathsf{rec}\ \beta_n.\tau'\} \cdots \{\beta_1 := \tau\}$, and the result is immediate. The other items follow from the first by rules (eqs) and (subr). □

Lemma B.8 *If* $\Delta \vdash_{\mathcal{F}} \tau_1 = \tau_2$ *then* $\mathrm{ftv}(\tau_1) = \mathrm{ftv}(\tau_2)$*.*

Proof: Define the depth of a type variable α in a type τ to be the least number of $\rightarrow$, $\langle\rangle$, $\forall$, and selfs between an occurance of α and the root of τ. I will prove the stronger result that τ_1 and τ_2 have the same free variables at the same depth by induction on the derivation of $\Delta \vdash_{\mathcal{F}} \tau_1 = \tau_2$. The only interesting case is rule (eq4). In this case there is a σ and α such that $\sigma \downarrow \alpha$, $\Delta \vdash_{\mathcal{F}} \sigma\{\alpha := \tau_1\} = \tau_1$, and $\Delta \vdash_{\mathcal{F}} \sigma\{\alpha := \tau_2\} = \tau_2$. By the induction hypothesis $\sigma\{\alpha := \tau_1\}$ and τ_1 have the same free variables at the same depths, and similarly for τ_2. If $\beta \in \mathrm{ftv}(\tau_1) - (\mathrm{ftv}(\sigma) - \{\alpha\})$ then β must be at a depth in $\sigma\{\alpha := \tau_1\}$ equal to its depth in τ_1 plus the depth of α is σ. By Lemma B.5, it must be that the depth of α in σ is at least 1. Therefore, the depth of β in $\sigma\{\alpha := \tau_1\}$ exceeds the depth of β in τ_1. But this is impossible so $\mathrm{ftv}(\tau_1) = \mathrm{ftv}(\sigma) - \{\alpha\}$ and at the same depths. Similarly for τ_2. By transitivity τ_1 and τ_2 have the same free variables at the same depths as required. □

Lemma B.9 (Equality) *If* $\Delta \vdash_{\mathcal{F}} \tau_1 = \tau_2$ *then* $\mathrm{nfoc}(\tau_1) = \mathrm{nfoc}(\tau_2)$ *and* $\Delta \vdash_{\mathcal{F}} \mathrm{nf}(\tau_1) = \mathrm{nf}(\tau_2)$ *by the (eqtv), (eqfun), (eqtup), (eqall), and (eqself) rule if* $\mathrm{nfoc}(\tau_1)$ *is* α*,* $\rightarrow$*,* $\langle\rangle$*,* $\forall$*, and* self *respectively. If* $\mathrm{nfoc}(\tau_1) = \bot = \mathrm{nfoc}(\tau_2)$ *then* $\Delta \vdash_{\mathcal{F}} \mathrm{nf}(\tau_1) = \mathrm{nf}(\tau_2)$ *by the rules (eqrec) and (eqtv).*

Proof: The last part follows by inspection. The rest is proven by induction on the derivation of $\Delta \vdash_{\mathcal{F}} \tau_1 = \tau_2$. If the last rule used was (eqs), (eqt), (eqtv), (eqfun), (eqtup), (eqall), or (eqself) the result follows easily. Consider the other cases:

case (eqrec): In this case $\tau_1 = \mathsf{rec}\ \alpha.\tau_1'$, $\tau_2 = \mathsf{rec}\ \alpha.\tau_2'$, and $\Delta, \alpha \vdash_{\mathcal{F}} \tau_1' = \tau_2'$. Case 1: $\tau_1' \downarrow \alpha$. By the induction hypothesis $\Delta, \alpha \vdash_{\mathcal{F}} \mathrm{nf}(\tau_1') = \mathrm{nf}(\tau_2')$. By Lemma B.5 and inspection of the definition of normal forms, $\mathrm{nf}(\tau_1') \neq \alpha$ thus $\mathrm{nf}(\tau_2') \neq \alpha$. Again by Lemma B.5 and inspection of the definition of normal forms, $\tau_2' \downarrow \alpha$. By Lemma B.4 $\Delta \vdash_{\mathcal{F}} \mathrm{nf}(\tau_1')\{\alpha := \tau_1\} = \mathrm{nf}(\tau_2')\{\alpha := \tau_2\}$. By definition $\mathrm{nf}(\tau_1) = \mathrm{nf}(\tau_1'\{\alpha := \tau_1\})$. By Lemma B.6 $\mathrm{nf}(\tau_1'\{\alpha := \tau_1\}) = \mathrm{nf}(\tau)\{\alpha := \tau_1\}$. Similarly for τ_2. Case 2: not $\tau_1' \downarrow \alpha$. By Lemma B.5 and inspection of the definition of normal forms, $\mathrm{nf}(\tau_1') = \alpha$. By the induction hypothesis $\mathrm{nf}(\tau_2') = \alpha$, and again by Lemma B.5 and inspection of the definition of normal forms, not $\tau_2' \downarrow \alpha$. Thus $\mathrm{nfoc}(\tau_1) = \bot = \mathrm{nfoc}(\tau_2)$ as required.

case (eq1): In this case $\tau_1 = \mathsf{rec}\ \alpha.\sigma$ and $\tau_2 = \sigma\{\alpha := \tau_1\}$. Clearly $\mathrm{nf}(\tau_1) = \mathrm{nf}(\tau_2)$ and the result is immediate.

case (eq2): In this case there is σ and α such that $\sigma \downarrow \alpha$, $\Delta \vdash_{\mathcal{F}} \sigma\{\alpha := \tau_1\} = \tau_1$, and $\Delta \vdash_{\mathcal{F}} \sigma\{\alpha := \tau_2\} = \tau_2$. By the induction hypothesis $\mathrm{nfoc}(\sigma\{\alpha := \tau_i\}) = \mathrm{nfoc}(\tau_i)$. By Lemma B.6, $\mathrm{nf}(\sigma\{\alpha := \tau\}) = \mathrm{nf}(\sigma)\{\alpha := \tau\}$ and $\mathrm{nfoc}(\sigma\{\alpha := \tau\}) = \mathrm{nfoc}(\sigma)$. Thus $\mathrm{nfoc}(\tau_1) = \mathrm{nfoc}(\tau_2)$. Case 1: $\mathrm{nfoc}(\tau_1) = \beta$ then $\mathrm{nf}(\tau_1) = \beta = \mathrm{nf}(\tau_2)$ and the result follows by (eqtv).

Case 2: $\mathrm{nfoc}(\tau_1) = \rightarrow$, then $\mathrm{nf}(\tau_1) = \tau_{11} \rightarrow \tau_{12}$ and $\mathrm{nf}(\tau_2) = \tau_{21} \rightarrow \tau_{22}$. By the induction hypothesis $\Delta \vdash_{\mathcal{F}} \mathrm{nf}(\sigma\{\alpha := \tau_i\}) = \tau_{i1} \rightarrow \tau_{i2}$ by (eqfun). By Lemma B.6 and inspection of (eqfun), there must be σ_1 and σ_2 such that $\mathrm{nf}(\sigma\{\alpha := \tau_i\}) = \sigma_1\{\alpha := \tau_i\} \rightarrow \sigma_2\{\alpha := \tau_i\}$ and $\Delta \vdash_{\mathcal{F}} \sigma_j\{\alpha := \tau_i\} = \tau_{ij}$. By Lemma B.7, Lemma B.4, and (eqt) $\Delta \vdash_{\mathcal{F}} \sigma_j\{\alpha := \tau_{i1} \rightarrow \tau_{i2}\} = \tau_{ij}$. By using rules (eq1), (eqs), and (eq2) $\Delta \vdash_{\mathcal{F}} \tau_{i1} = \mathsf{rec}\ \alpha_1.\sigma_1\{\alpha := \alpha_1 \rightarrow \tau_{i2}\}$ and $\Delta \vdash_{\mathcal{F}} \tau_{i2} = \mathsf{rec}\ \alpha_2.\sigma_2\{\alpha := \tau_{i1} \rightarrow \alpha_2\}$ for fresh α_1 and α_2. By Lemma B.4 and (eqt) $\Delta \vdash_{\mathcal{F}} \tau_{i1} = \mathsf{rec}\ \alpha_1.\sigma_1\{\alpha := \alpha_1 \rightarrow \mathsf{rec}\ \alpha_2.\sigma_2\{\alpha := \tau_{i1} \rightarrow \alpha_2\}\}$ and $\Delta \vdash_{\mathcal{F}} \tau_{i2} = \mathsf{rec}\ \alpha_2.\sigma_2\{\alpha := \mathsf{rec}\ \alpha_1.\sigma_1\{\alpha := \alpha_1 \rightarrow \tau_{i2}\} \rightarrow \alpha_2\}$. By rules (eq1), (eqs), and (eq2) $\Delta \vdash_{\mathcal{F}} \tau_{i1} = \mathsf{rec}\ \beta.\mathsf{rec}\ \alpha_1.\sigma_1\{\alpha := \alpha_1 \rightarrow \mathsf{rec}\ \alpha_2.\sigma_2\{\alpha := \beta \rightarrow \alpha_2\}\}$ and $\Delta \vdash_{\mathcal{F}} \tau_{i2} = \mathsf{rec}\ \beta.\mathsf{rec}\ \alpha_2.\sigma_2\{\alpha := \mathsf{rec}\ \alpha_1.\sigma_1\{\alpha := \alpha_1 \rightarrow \beta\} \rightarrow \alpha_2\}$ for fresh β. By rules (eqs) and (eqt) $\Delta \vdash_{\mathcal{F}} \tau_{1j} = \tau_{2j}$. So $\Delta \vdash_{\mathcal{F}} \tau_{11} \rightarrow \tau_{12} = \tau_{21} \rightarrow \tau_{22}$ by (eqfun) as required. The cases for $\mathrm{nfoc}(\tau_1) \in \{\langle\rangle, \forall, \mathsf{self}\}$ are similar.

□

Define reachability in B as the least relation satisfying: If $\alpha \in \mathrm{ftv}(\tau)$ then α is reachable from τ in B. If α is reachable from $B(\beta)$ in B then α is reachable from β in B.

Lemma B.10 *If α is not reachable from τ_1 in B and $\Delta; B \vdash_{\mathcal{F}} \tau_1 \leq \tau_2$ then $\alpha \notin \mathrm{ftv}(\tau_2)$.*

Proof: By induction on the derivation of $\Delta; B \vdash_{\mathcal{F}} \tau_1 \leq \tau_2$. □

Lemma B.11 *If $\Delta; B \vdash_{\mathcal{F}} \tau_1 \leq \tau_2$ then:*

- $\mathrm{nfoc}(\tau_1) = \alpha$ *or* $\mathrm{nfoc}(\tau_1) = \mathrm{nfoc}(\tau_2)$.
- *If* $\mathrm{nf}(\tau_1) = \alpha$ *then either* $\Delta \vdash_{\mathcal{F}} \alpha = \tau_2$ *or* $\Delta; B \vdash_{\mathcal{F}} B(\alpha) \leq \tau_2$ *by a derivation whose height of subtyping rules is at most that of* $\Delta; B \vdash_{\mathcal{F}} \tau_1 \leq \tau_2$.
- *If* $\mathrm{nfoc}(\tau_1) = \mathrm{nfoc}(\tau_2) \in \{\rightarrow, \langle\rangle, \forall, \mathsf{self}\}$ *then* $\Delta; B \vdash_{\mathcal{F}} \mathrm{nf}(\tau_1) \leq \mathrm{nf}(\tau_2)$ *by the (subfun), (subtup), (suball), and (subself) rule respectively.*

Proof: By induction on the height (of subtyping rules) of the derivation of $\Delta; B \vdash_{\mathcal{F}} \tau_1 \leq \tau_2$. If the last rule used was (subtv), (subfun), (subtup), (suball), or (subself), the result follows easily. Consider the other cases:

case (subr): By Equality $\mathrm{nfoc}(\tau_1) = \mathrm{nfoc}(\tau_1)$. If $\mathrm{nf}(\tau_1) = \alpha$ then $\mathrm{nf}(\tau_2) = \alpha$. The result follows by Lemma B.7. If $\mathrm{nfoc}(\tau_1) = \mathrm{nfoc}(\tau_2) = \rightarrow$ then by Equality $\Delta \vdash_{\mathcal{F}} \mathrm{nf}(\tau_1) = \mathrm{nf}(\tau_2)$ by rule (eqfun). By inspection of the definition, $\mathrm{nf}(\tau_1) = \tau_{11} \rightarrow \tau_{12}$ and $\mathrm{nf}(\tau_2) = \tau_{21} \rightarrow \tau_{22}$. By inspection of rule (eqfun), $\Delta \vdash_{\mathcal{F}} \tau_{11} = \tau_{21}$ and $\Delta \vdash_{\mathcal{F}} \tau_{12} = \tau_{22}$. By rule (eqs) $\Delta \vdash_{\mathcal{F}} \tau_{21} = \tau_{11}$. By rule (subr) $\Delta; B \vdash_{\mathcal{F}} \tau_{21} \leq \tau_{11}$ and $\Delta; B \vdash_{\mathcal{F}} \tau_{12} \leq \tau_{22}$. Thus $\Delta; B \vdash_{\mathcal{F}} \mathrm{nf}(\tau_1) \leq \mathrm{nf}(\tau_2)$ by rule (subfun) as required. The cases for $\mathrm{nfoc}(\tau_1)$ equal to $\langle\rangle$, $\forall$, and self are similar.

case (subt): In this case $\Delta; B \vdash_{\mathcal{F}} \tau_1 \leq \sigma$ and $\Delta; B \vdash_{\mathcal{F}} \sigma \leq \tau_2$. If $\mathrm{nfoc}(\tau_1) = \alpha$ then by the induction hypothesis on the first hypothesis either $\Delta \vdash_{\mathcal{F}} \alpha = \sigma$ or $\Delta; B \vdash_{\mathcal{F}} B(\alpha) \leq \sigma$. If the former then by Equality $\mathrm{nfoc}(\sigma) = \alpha$ and the result follows by the induction hypothesis on the second hypothesis. If the latter then by (eqt) $\Delta; B \vdash_{\mathcal{F}} B(\alpha) \leq \tau_2$ as required.

If $\mathrm{nfoc}(\tau_1) \neq \alpha$ then by the induction hypothesis $\mathrm{nfoc}(\tau_1) = \mathrm{nfoc}(\sigma) = \mathrm{nfoc}(\tau_2)$. If $\mathrm{nfoc}(\tau_1) = \bot$ then nothing further needs to be proven. If $\mathrm{nfoc}(\tau_1) = \rightarrow$ then by the induction hypothesis $\Delta; B \vdash_{\mathcal{F}} \mathrm{nf}(\tau_1) \leq \mathrm{nf}(\sigma)$ and $\Delta; B \vdash_{\mathcal{F}} \mathrm{nf}(\sigma) \leq \mathrm{nf}(\tau_2)$, both by the (subfun) rule. Thus $\mathrm{nf}(\tau_1) = \tau_{11} \rightarrow \tau_{12}$, $\mathrm{nf}(\sigma) = \sigma_1 \rightarrow \sigma_2$, $\mathrm{nf}(\tau_2) = \tau_{21} \rightarrow \tau_{22}$, $\Delta; B \vdash_{\mathcal{F}} \sigma_1 \leq \tau_{11}$, $\Delta; B \vdash_{\mathcal{F}} \tau_{12} \leq \sigma_2$, $\Delta; B \vdash_{\mathcal{F}} \tau_{21} \leq \sigma_1$, and $\Delta; B \vdash_{\mathcal{F}} \sigma_2 \leq \tau_{22}$. By rule (subt) $\Delta; B \vdash_{\mathcal{F}} \tau_{21} \leq \tau_{11}$ and $\Delta; B \vdash_{\mathcal{F}} \tau_{12} \leq \tau_{22}$. Thus $\Delta; B \vdash_{\mathcal{F}} \mathrm{nf}(\tau_1) \leq \mathrm{nf}(\tau_2)$ by rule (subfun) as required. The cases for $\mathrm{nfoc}(\tau_1)$ equal to $\langle\rangle$, $\forall$, and self are similar.

case (subrec): In this case $\tau_i = \mathsf{rec}\ \alpha_i.\tau_i'$ and $\Delta, \alpha_1, \alpha_2; B, \alpha_1 \leq \alpha_2 \vdash_{\mathcal{F}} \tau_1' \leq \tau_2'$. By the inductive hypothesis there are two cases.

Case 1, $\mathrm{nfoc}(\tau_1')$ is a type variable. If $\mathrm{nfoc}(\tau_1') \neq \alpha_1$ then let $\mathrm{nfoc}(\tau_1') = \beta$. By definition $\tau_1' \downarrow \alpha_1$ and $\mathrm{nf}(\tau_1) = \mathrm{nf}(\tau_1')$, so $\mathrm{nfoc}(\tau_1)$ is a type variable. By the induction hypothesis either $\Delta, \alpha_1, \alpha_2 \vdash_{\mathcal{F}} \beta = \tau_2'$ or $\Delta, \alpha_1, \alpha_2; B, \alpha_1 \leq \alpha_2 \vdash_{\mathcal{F}} B(\beta) \leq \tau_2'$. If the former then by Equality $\mathrm{nf}(\tau_2') = \beta$, so $\mathrm{nf}(\tau_2) = \beta$. The result follows by Lemma B.7. If the latter then α_2 is not reachable from β in $B, \alpha_1 \leq \alpha_2$. By Lemma B.10 $\alpha_2 \notin \mathrm{ftv}(\tau_2')$. By (eq1) and (eqs) $\Delta, \alpha_1, \alpha_2 \vdash_{\mathcal{F}} \tau_2' = \tau_2$. By (subr) and (subt) $\Delta, \alpha_1, \alpha_2; B, \alpha_1 \leq \alpha_2 \vdash_{\mathcal{F}} B(\beta) \leq \tau_2$. By induction on this derivation, it is easy to see that $\Delta; B \vdash_{\mathcal{F}} B(\beta) \leq \tau_2$ by a derivation of exactly the same height of subtyping rules as required.

If $\mathrm{nfoc}(\tau_1') = \alpha_1$ then $\mathrm{nfoc}(\tau_1) = \bot$. By the induction hypothesis either $\Delta, \alpha_1, \alpha_2 \vdash_{\mathcal{F}} \alpha_1 = \tau_2'$ or $\Delta, \alpha_1, \alpha_2; B, \alpha_1 \leq \alpha_2 \vdash_{\mathcal{F}} \alpha_2 \leq \tau_2'$. The former is impossible since $\alpha_1 \notin \mathrm{ftv}(\tau_2')$. Since the height of $\Delta, \alpha_1, \alpha_2; B, \alpha_1 \leq \alpha_2 \vdash_{\mathcal{F}} \alpha_2 \leq \tau_2'$ is at most the height of $\Delta, \alpha_1, \alpha_2; B, \alpha_1 \leq \alpha_2 \vdash_{\mathcal{F}} \tau_1' \leq \tau_2'$, which is shorter than $\Delta; B \vdash_{\mathcal{F}} \tau_1 \leq \tau_2$, by the induction hypothesis either $\Delta, \alpha_1, \alpha_2 \vdash_{\mathcal{F}} \alpha_2 = \tau_2'$, note that α_2 has no bound in $B, \alpha_1 \leq \alpha_2$. By Equality $\mathrm{nf}(\tau_2') = \alpha_2$ so $\mathrm{nfoc}(\tau_2') = \bot$ as required.

Case 2, $\mathrm{nfoc}(\tau'_1) = \mathrm{nfoc}(\tau'_2)$ is not a type variable: By Lemma B.5, $\tau'_i \downarrow \alpha$. By Lemma B.6 and the definition of normal forms, $\mathrm{nf}(\tau_i) = \mathrm{nf}(\tau'_i)\{\alpha_i := \tau_i\}$ and $\mathrm{nfoc}(\tau_i) = \mathrm{nfoc}(\tau'_i)$. Thus $\mathrm{nfoc}(\tau_1) = \mathrm{nfoc}(\tau_2)$. If $\mathrm{nfoc}(\tau_1) = \bot$ nothing further needs to be proven. If $\mathrm{nfoc}(\tau_1) = \rightarrow$ then by the induction hypothesis there must be τ_{11}, τ_{12}, τ_{21}, and τ_{22} such that $\Delta, \alpha_1, \alpha_2; B, \alpha_1 \leq \alpha_2 \vdash_{\mathcal{F}} \tau_{21} \leq \tau_{12}$, $\Delta, \alpha_1, \alpha_2; B, \alpha_1 \leq \alpha_2 \vdash_{\mathcal{F}} \tau_{12} \leq \tau_{22}$, and $\mathrm{nf}(\tau_i) = \tau_{i1}\{\alpha_i := \tau_i\} \rightarrow \tau_{i2}\{\alpha_i := \tau_i\}$. By type substitution twice $\Delta; B \vdash_{\mathcal{F}} \tau_{21}\{\alpha_2 := \tau_2\} \leq \tau_{11}\{\alpha_1 := \tau_1\}$ and $\Delta; B \vdash_{\mathcal{F}} \tau_{12}\{\alpha_1 := \tau_1\} \leq \tau_{22}\{\alpha_2 := \tau_2\}$. So by (subfun) $\Delta; B \vdash_{\mathcal{F}} \mathrm{nf}(\tau_1) \leq \mathrm{nf}(\tau_2)$ as required. The cases for $\mathrm{nfoc}(\tau_1) \in \{\langle\rangle, \forall, \mathsf{self}\}$ are similar.

□

Note that Lemma B.11 implies that there is an algorithm that given τ and B will compute the least τ' that is a supertype of τ and has an arrow, tuple, forall, or self quantified form. The algorithm normalises τ and if the result is a type variable repeats with its bound in B until either it finds either a normal form that is not a type variable or detects a cycle in the type variables. If a normal form of the right form is found the algorithm succeeds with that type, otherwise it fails.

Lemma B.12 (Subtyping)

- *If* $\Delta; B \vdash_{\mathcal{F}} \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2$ *then* $\Delta; B \vdash_{\mathcal{F}} \sigma_1 \leq \tau_1$ *and* $\Delta; B \vdash_{\mathcal{F}} \tau_2 \leq \sigma_2$*.*
- *If* $\Delta; B \vdash_{\mathcal{F}} \langle \ell_i{:}\tau_i^{\phi_i} \rangle^{\varphi}_{i \in I} \leq \langle \ell_i{:}\tau_i'^{\phi'_i} \rangle^{\varphi'}_{i \in I'}$ *then* $I \leq I'$*, for* $i \in I'$*:* $\Delta; B \vdash_{\mathcal{F}} \tau_i^{\phi_i} \leq \tau_i'^{\phi'_i}$*, and* $\varphi' = \circ$ *implies* $I = I'$*.*
- *If* $\Delta; B \vdash_{\mathcal{F}} \tau_1^{\phi_1} \leq \tau_2^{\phi_2}$ *and* $\phi_2 \in \{+, \circ\}$ *then* $\Delta; B \vdash_{\mathcal{F}} \tau_1 \leq \tau_2$*.*
- *If* $\Delta; B \vdash_{\mathcal{F}} \tau_1^{\phi_1} \leq \tau_2^{\phi_2}$ *and* $\phi_2 \in \{-, \circ\}$ *then* $\Delta; B \vdash_{\mathcal{F}} \tau_2 \leq \tau_1$*.*
- *If* $\Delta; B \vdash_{\mathcal{F}} \forall \alpha \leq \tau_1.\tau_2 \leq \forall \alpha \leq \sigma_1.\sigma_2$ *then* $\Delta, \alpha \vdash_{\mathcal{F}} \tau_1 = \sigma_1$ *and* $\Delta, \alpha; B, \alpha_1 \leq \tau_1 \vdash_{\mathcal{F}} \tau_2 \leq \sigma_2$*.*
- *If* $\Delta; B \vdash_{\mathcal{F}} \mathsf{self}\ \alpha.\tau_1 \leq \mathsf{self}\ \alpha.\tau_2$ *then* $\Delta, \alpha; B \vdash_{\mathcal{F}} \tau_1 \leq \tau_2$*.*

Proof: The first, second, fifth, and sixth items follow immediately from Lemma B.11. The third and fourth items are by inspection of the rules for variance subtyping. □

Lemma B.13 (Value Substitution) *If* $\Delta; B; \Gamma, x{:}\tau_1 \vdash_{\mathcal{F}} e_2 : \tau_2$ *and* $\Delta; B; \Gamma \vdash_{\mathcal{F}} e_1 : \tau_1$ *then* $\Delta; B; \Gamma \vdash_{\mathcal{F}} e_2\{x := e_1\} : \tau_2$*.*

Proof: By induction of the structure of the derivation of $\Delta; B; \Gamma, x{:}\tau_1 \vdash_{\mathcal{F}} e_2 : \tau_2$, inspection of the typing rules, and Context Strengthening. □

Lemma B.14 *If* $\emptyset; \emptyset; \emptyset \vdash_{\mathcal{F}} E\{e\} : \tau_1$ *then there exists* τ_2 *such that* $\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} e : \tau_2$ *where the last rule used is not (subsume) and for all* e' *such that* $\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} e' : \tau_2$*,* $\emptyset; \emptyset; \emptyset \vdash_{\mathcal{F}} E\{e'\} : \tau_1$*.*

Proof: By induction on the structure of E and inspection of the typing rules. □

Lemma B.15 (Decomposition) *An expresison e is either a value or of the form $E\{\iota\}$ for:*

$$\begin{array}{rcl} \iota & ::= & x \mid v_1\ v_2 \mid v.\ell \mid v_1.\ell \leftarrow v_2 \mid v_1.\ell := v_2 \mid v_1 + \ell = v_2 \mid v[\tau] \mid \\ & & \mathsf{unpack}\ \alpha, x = v\ \mathsf{in}\ b \mid \mathsf{let}\ \overrightarrow{x = v}\ \mathsf{in}\ b \end{array}$$

Proof: By induction of the structure of E and inspection of the various cases. □

Lemma B.16 (Canonical Forms) *If $\Delta; B; \emptyset \vdash_{\mathcal{F}} v : \tau$ and τ has one of the forms listed in the following table then v has the corresponding form and the side conditions are true.*

v	τ	*Side Conditions*
$\lambda x{:}\tau_1'.b$	$\tau_1 \to \tau_2$	
$\langle \ell_i = v_i \rangle_{i \in I'}$	$\langle \ell_i{:}\tau_i^{\phi_i} \rangle_{i \in I}^{\varphi}$	*I prefix of I'; $\varphi = \circ$ implies $I = I'$*
$\Lambda\alpha \le \tau_1'.v$	$\forall\alpha \le \tau_1.\tau_2$	
$\mathsf{pack}\ v, \tau\ \mathsf{as\ self}\ \alpha.\sigma$	$\mathsf{self}\ \alpha.\sigma'$	

Proof: The only applicable rules for $\Delta; B; \emptyset \vdash_{\mathcal{F}} v : \tau$ are the rule for λ-abstractions, the rule for tuples, the rule for Λ-abstractions, the rule for pack, and the subsumption rule. Inspection of the first three shows them to satisfy the forms and side conditions in the table. Next note that these rules give a type only of the form shown in the table. By Subtyping, if one of these types is a subtype of another then they must have the same form, as each form has a different normal form outermost constructor. The result follows by Subtyping and inspection. □

Theorem B.17 (Type Preservation) *If $\emptyset; \emptyset; \emptyset \vdash_{\mathcal{F}} e_1 : \tau$ and $e_1 \mapsto e_2$ then $\emptyset; \emptyset; \emptyset \vdash_{\mathcal{F}} e_2 : \tau$.*

Proof: Let $e_1 = E\{\iota\}$ and $e_2 = E\{e\}$ where ι and e have one of the forms shown in Figure 4. By Lemma B.14 $\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \iota : \sigma$, and $\emptyset; \emptyset; \emptyset \vdash_{\mathcal{F}} e_2 : \tau$ follows from $\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} e : \sigma$. Let TD be the derivation of $\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \iota : \sigma$, and consider the cases for ι:

case $\iota = (\lambda x{:}\sigma_1.b)\ v$: In this case $e = b\{x := v\}$. TD must have the form:

$$\dfrac{\dfrac{\dfrac{\Delta_E; B_E; x{:}\sigma_1 \vdash_{\mathcal{F}} b : \sigma_2}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \lambda x{:}\sigma_1.b : \sigma_1 \to \sigma_2} \quad \Delta_E; B_E \vdash_{\mathcal{F}} \sigma_1 \to \sigma_2 \le \sigma' \to \sigma}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \lambda x{:}\sigma_1.b : \sigma' \to \sigma} \qquad \Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} v : \sigma'}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \iota : \sigma}$$

By Subtyping $\Delta_E; B_E \vdash_{\mathcal{F}} \sigma' \le \sigma_1$ (1) and $\Delta_E; B_E \vdash_{\mathcal{F}} \sigma_2 \le \sigma$. By (1), the judgement for v, and subsumption $\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} v : \sigma_1$. By Value Substitution $\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} v : \sigma_2$. The result follows by (2) and subsumption.

case $\iota = \langle \ell_i = v_i \rangle_{i \in I}.\ell_k$**:** In this case $e = v_k$ and $k \in I$. TD must have the form:

$$\dfrac{\dfrac{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} v_i : \tau_i}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \langle \ell_i = v_i \rangle_{i \in I}.\ell_k : \langle \ell_i{:}\tau_i^\circ \rangle^\circ_{i \in I}} \quad \Delta_E; B_E \vdash_{\mathcal{F}} \langle \ell_i{:}\tau_i^\circ \rangle^\circ_{i \in I} \le \langle \ell_i{:}\sigma_i^{\phi_i} \rangle^\varphi_{i \in I'}}{\dfrac{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \langle \ell_i = v_i \rangle_{i \in I}.\ell_k : \langle \ell_i{:}\sigma_i^{\phi_i} \rangle^\varphi_{i \in I'}}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \iota : \sigma}}$$

Where $\sigma = \sigma_k$, $k \in I'$, and $\phi_k \in \{+, \circ\}$. By Subtyping $\Delta_E; B_E \vdash_{\mathcal{F}} \tau_k \le \sigma_k$, and the result follows by the judgement on v_k and subsumption.

case $\iota = \langle \ell_i = v_i \rangle_{i \in I}.\ell_k \leftarrow v$**:** In this case $k \in I$ and $e = \langle \ell_i = v_i' \rangle_{i \in I}$ where $v_i' = v_i$ when $i \ne k$ and $v_k' = v$. TD must have the form:

$$\dfrac{A \quad \Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} v : \sigma'}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \iota : \sigma}$$

Where $k \in I'$, $\sigma = \langle \ell_i{:}\sigma_i'^{\phi_i'} \rangle^\varphi_{i \in I'}$, for $i \ne k$: $\sigma_i' = \sigma_i$ and $\phi_i' = \phi_i$, $\sigma_k' = \sigma'$, $\phi_k' = \circ$, and A is:

$$\dfrac{\dfrac{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} v_i : \tau_i}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \langle \ell_i = v_i \rangle_{i \in I}.\ell_k : \langle \ell_i{:}\tau_i^\circ \rangle^\circ_{i \in I}} \quad \Delta_E; B_E \vdash_{\mathcal{F}} \langle \ell_i{:}\tau_i^\circ \rangle^\circ_{i \in I} \le \langle \ell_i{:}\sigma_i^{\phi_i} \rangle^\varphi_{i \in I'}}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \langle \ell_i = v_i \rangle_{i \in I} : \langle \ell_i{:}\sigma_i^{\phi_i} \rangle^\varphi_{i \in I'}}$$

By Subtyping for $i \in I'$: $\Delta_E; B_E \vdash_{\mathcal{F}} \tau_i^\circ \le \sigma_i^{\phi_i}$, so $\Delta_E; B_E \vdash_{\mathcal{F}} \langle \ell_i{:}\tau_i'^\circ \rangle^\circ_{i \in I} \le \langle \ell_i{:}\sigma_i'^{\phi_i'} \rangle^\varphi_{I'}$, where for $i \ne k$: $\tau_i' = \tau_i$ and $\tau_k' = \sigma'$. Also $\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} v_i' : \tau_i'$, so $\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} e : \langle \ell_i{:}\tau_i'^\circ \rangle^\circ_{i \in I}$. The result follows by subsumption.

case $\iota = \langle \ell_i = v_i \rangle_{i \in I}.\ell_k := v$**:** In this case $k \in I$ and $e = \langle \ell_i = v_i' \rangle_{i \in I}$ where $v_i' = v_i$ when $i \ne k$ and $v_k' = v$. TD must have the form:

$$\dfrac{\dfrac{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} v_i : \tau_i}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \langle \ell_i = v_i \rangle_{i \in I} : \sigma} \quad \Delta_E; B_E \vdash_{\mathcal{F}} \sigma \le \langle \ell_i{:}\sigma_i^{\phi_i} \rangle^\varphi_{i \in I'} \quad \Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} v : \sigma_k}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \iota : \sigma}$$

Where $\sigma = \langle \ell_i{:}\tau_i^\circ \rangle^\circ_{i \in I}$, $k \in I'$, $\phi_k \in \{-, \circ\}$, and note that subsumption on the left hypothesis can be moved into the middle hypothesis and the typing of E. By Subtyping $\Delta_E; B_E \vdash_{\mathcal{F}} \sigma_k \le \tau_k$. By the judgement on v and subsumption $\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} v : \tau_k$, so $\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} v_i' : \tau_i$. By the tuple rule $\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} e : \sigma$ as required.

case $\iota = \langle \ell_i = v_i \rangle_{i \in I} + \ell = v$**:** In this case $e = \langle \ell_i = v_i, \ell = v \rangle_{i \in I}$ and $\ell \notin \ell_{i \in I}$. TD must have the form:

$$\dfrac{A \quad \Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} v : \sigma'}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \iota : \sigma}$$

Where $\sigma = \langle \ell_i{:}\sigma_i^{\phi_i}, \ell{:}\sigma'^\circ \rangle^\circ_{i \in I'}$ and A is:

$$\dfrac{\dfrac{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} v_i : \tau_i}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \langle \ell_i = v_i \rangle_{i \in I} : \langle \ell_i{:}\tau_i^\circ \rangle^\circ_{i \in I}} \quad \Delta_E; B_E \vdash_{\mathcal{F}} \langle \ell_i{:}\tau_i^\circ \rangle^\circ_{i \in I} \le \langle \ell_i{:}\sigma_i^{\phi_i} \rangle^\circ_{i \in I'}}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \langle \ell_i = v_i \rangle_{i \in I} : \langle \ell_i{:}\sigma_i^{\phi_i} \rangle^\varphi_{i \in I'}}$$

By Subtyping $I = I'$ and $\Delta_E; B_E \vdash_{\mathcal{F}} \tau_i^\circ \le \sigma_i^{\phi_i}$. So $\Delta_E; B_E \vdash_{\mathcal{F}} \langle \ell_i{:}\tau_i^\circ, \ell{:}\sigma'^\circ \rangle^\circ_{i \in It} \le \langle \ell_i{:}\sigma_i^{\phi_i}, \ell{:}\sigma'^\circ \rangle^\circ_{i \in I'}$. Clearly $\Delta_E; B_E \vdash_{\mathcal{F}} e : \langle \ell_i{:}\tau_i^\circ, \ell{:}\sigma'^\circ \rangle^\circ_{i \in I}$, so the result follows by subsumption.

case $\iota = (\Lambda\alpha \le \tau_1.v)[\sigma']$**:** In this case $e = v\{\alpha := \sigma'\}$. TD must have the form:

$$\dfrac{A \quad \Delta_E; B_E \vdash_{\mathcal{F}} \sigma' \le \tau_1'\{\alpha := \sigma'\}}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \iota : \sigma}$$

Where $\sigma = \tau_2'\{\alpha := \sigma'\}$ and A is:

$$\dfrac{\dfrac{\Delta_E, \alpha; B_E, \alpha \le \tau_1; \emptyset \vdash_{\mathcal{F}} v : \tau_2}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \Lambda\alpha \le \tau_1.v : \forall\alpha \le \tau_1.\tau_2} \quad \Delta_E; B_E \vdash_{\mathcal{F}} \forall\alpha \le \tau_1.\tau_2 \le \forall\alpha \le \tau_1'.\tau_2'}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \Lambda\alpha \le \tau_1.v : \forall\alpha \le \tau_1'.\tau_2'}$$

By Subtyping $\Delta_E, \alpha \vdash_{\mathcal{F}} \tau_1 = \tau_1'$ and $\Delta_E, \alpha; B_E, \alpha_1 \le \tau_1 \vdash_{\mathcal{F}} \tau_2 \le \tau_2'$. By Type Substitution $\Delta_E \vdash_{\mathcal{F}} \tau_1\{\alpha := \sigma'\} = \tau_1'\{\alpha := \sigma'\}$. By rules (eqs), (subr), and (subt), $\Delta_E; B_E \vdash_{\mathcal{F}} \sigma' \le \tau_1\{\alpha := \sigma'\}$. By Type Subsitution, $\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} v\{\alpha := \sigma'\} : \tau_2\{\alpha := \sigma'\}$ and $\Delta_E; B_E \vdash_{\mathcal{F}} \tau_2\{\alpha := \sigma'\} \le \tau_2'\{\alpha := \sigma'\}$. The result follows by subsumption.

case $\iota = \mathsf{unpack}\ \alpha, x = \mathsf{pack}\ v, \tau'\ \mathsf{as\ self}\ \alpha.\tau_1\ \mathsf{in}\ b$**:** In this case $e = b\{\alpha, x := \tau', v\}$. TD must have the form, where $v' = \mathsf{pack}\ v, \tau\ \mathsf{as\ self}\ \alpha.\sigma$:

$$\dfrac{A \quad \Delta_E, \alpha; B_E, \alpha \le \tau_2; x{:}\alpha \vdash_{\mathcal{F}} b : \sigma \quad \Delta_E \vdash_{\mathcal{F}} \sigma}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \iota : \sigma}$$

Where A is:

$$\dfrac{\dfrac{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} v : \tau' \quad \Delta_E; B_E \vdash_{\mathcal{F}} \tau' \le \tau_1\{\alpha := \tau'\}}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} v' : \mathsf{self}\ \alpha.\tau_1} \quad \Delta_E; B_E \vdash_{\mathcal{F}} \mathsf{self}\ \alpha.\tau_1 \le \mathsf{self}\ \alpha.\tau_2}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} v' : \mathsf{self}\ \alpha.\tau_2}$$

By Subtyping $\Delta_E, \alpha; B_E \vdash_{\mathcal{F}} \tau_1 \le \tau_2$. By Type Substitution $\Delta_E; B_E \vdash_{\mathcal{F}} \tau_1\{\alpha := \tau'\} \le \tau_2\{\alpha := \tau'\}$. By (subt) $\Delta_E; B_E \vdash_{\mathcal{F}} \tau' \le \tau_2\{\alpha := \tau'\}$. By Type Substitution $\Delta_E; B_E; x{:}\tau' \vdash_{\mathcal{F}} b\{\alpha := \tau'\} : \sigma$. Note that $B_E\{\alpha := \tau\} = B_E$ because $\alpha \notin \Delta_E \supseteq \mathrm{ftv}(\mathrm{ran}(B_E))$ and $\sigma\{\alpha := \tau'\} = \sigma$ because $\alpha \notin \Delta_E \supseteq \mathrm{ftv}(\sigma)$ by Derived Judgements. By Value Substitution $\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} b\{\alpha := \tau'\}\{x := v\} : \sigma$ as required.

case $\iota = \mathsf{let}\ x_1 = v_2\ \mathsf{and}\ \cdots\ \mathsf{and}\ x_n = v_n\ \mathsf{in}\ b$**:** In this case $e = b\{\vec{x} := \vec{v}\}$. TD must have the form:

$$\dfrac{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} v_i : \tau_i \quad \Delta_E; B_E; x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash_{\mathcal{F}} b : v}{\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \iota : \sigma}$$

The result follows by Value Substitution n times.

□

Theorem B.18 (Progress) *If $\emptyset; \emptyset; \emptyset \vdash_{\mathcal{F}} e : \tau$ then either e is a value or there exists e' such that $e \mapsto e'$.*

Proof: By Decomposition either e is a value, as required, or e has the form $E\{\iota\}$ for ι as in that lemma. By Lemma B.14 $\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \iota : \sigma$ for some type σ. It remains to show that ι has one of the forms in Figure 4 and that the side conditions are satisfied. For $\iota = x$, x is not typeable. For ι one of $v_1\ v_2$, $v.\ell$, $v_1.\ell \leftarrow v_2$, $v_1.\ell := v_2$, $v_1 + \ell = v_2$, $v[\tau]$, and $\mathsf{unpack}\ \alpha, x = v\ \mathsf{in}\ b$ inspection of the last type rule used in $\Delta_E; B_E; \emptyset \vdash_{\mathcal{F}} \iota : \sigma$ reveals a certain form for the type of v_1 or v. The result then follows by Canonical Forms. For $\iota = \mathsf{let}\ \overrightarrow{x = v}\ \mathsf{in}\ b$, ι has the last form and there are no sideconditions. □

Theorem B.19 *If $\emptyset; \emptyset; \emptyset \vdash_{\mathcal{F}} e : \tau$ and $e \mapsto^* e'$ then e' is not stuck.*

Proof: By induction of the length of $\mapsto^*$, Type Preservation, and Progress. □

C Type Preservation

Theorem C.1

- *For any source type τ, $\emptyset \vdash_{\mathcal{F}} [\![\tau]\!]_{\mathrm{type}}$.*
- *For any source row r, $\alpha \vdash_{\mathcal{F}} [\![r]\!]_{\mathrm{mt}}(\alpha, \varphi)$ and $\alpha \vdash_{\mathcal{F}} [\![r]\!]_{\mathrm{full}}(\alpha)$.*
- *For any source row r, and if $\Delta \vdash_{\mathcal{F}} \tau$ then $\Delta \vdash_{\mathcal{F}} [\![r]\!]_{\mathrm{mt}}(\tau, \varphi)$ and $\Delta \vdash_{\mathcal{F}} [\![r]\!]_{\mathrm{full}}(\tau)$.*
- *For any source signature s, $\alpha \vdash_{\mathcal{F}} [\![s]\!]_{\mathrm{sig}}(\alpha)$.*
- *For any source signature s, and if $\Delta \vdash_{\mathcal{F}} \tau$ then $\Delta \vdash_{\mathcal{F}} [\![s]\!]_{\mathrm{sig}}(\tau)$.*

Proof: I will prove that:

$$\begin{array}{lcl} \mathrm{ftv}([\![\tau]\!]_{\mathrm{type}}) & = & \emptyset \\ \mathrm{ftv}([\![r]\!]_{\mathrm{mt}}(\tau, \varphi)) & = & \mathrm{ftv}(\tau) \\ \mathrm{ftv}([\![r]\!]_{\mathrm{full}}(\tau)) & = & \mathrm{ftv}(\tau) \\ \mathrm{ftv}([\![s]\!]_{\mathrm{sig}}(\tau)) & = & \mathrm{ftv}(\tau) \end{array}$$

Then the result follows by Derived Judgements. The proof proceeds by mutual induction on the structure of τ, r, and s. The rest is calculation:

$$\begin{array}{lcl}
\mathrm{ftv}([\![\mathsf{temp}\ r]\!]_{\mathrm{type}}) & = & \mathrm{ftv}(\forall\alpha \leq [\![r]\!]_{\mathrm{full}}(\alpha).[\![r]\!]_{\mathrm{mt}}(\alpha, \circ)) \\
& = & (\mathrm{ftv}([\![r]\!]_{\mathrm{full}}(\alpha)) \cup \mathrm{ftv}([\![r]\!]_{\mathrm{mt}}(\alpha, \circ))) - \alpha \\
& = & (\mathrm{ftv}(\alpha) \cup \mathrm{ftv}(\alpha)) - \alpha \\
& = & \emptyset \\
\mathrm{ftv}([\![\mathsf{obj}\ r]\!]_{\mathrm{type}}) & = & \mathrm{ftv}(\mathsf{self}\ \alpha.[\![r]\!]_{\mathrm{full}}(\alpha)) \\
& = & \mathrm{ftv}([\![r]\!]_{\mathrm{full}}(\alpha)) - \alpha \\
& = & \mathrm{ftv}(\alpha) - \alpha \\
& = & \emptyset \\
\mathrm{ftv}([\![r]\!]_{\mathrm{mt}}(\tau, \varphi)) & = & \mathrm{ftv}(\langle m_i{:}[\![s_i]\!]_{\mathrm{sig}}(\tau)^+\rangle^{\varphi}_{i\in I}) \\
& = & \cup_{i\in I}\mathrm{ftv}([\![s_i]\!]_{\mathrm{sig}}(\tau)) \\
& = & \cup_{i\in I}\mathrm{ftv}(\tau) \\
& = & \mathrm{ftv}(\tau) \\
\mathrm{ftv}([\![r]\!]_{\mathrm{full}}(\tau)) & = & \mathrm{ftv}(\langle \mathsf{mt}{:}[\![r]\!]_{\mathrm{mt}}(\tau, \rightarrow)^+, f_j{:}[\![\sigma]\!]^{\phi_j}_{\mathrm{type}}\rangle_{j\in J}) \\
& = & \mathrm{ftv}([\![r]\!]_{\mathrm{mt}}(\tau, \rightarrow)) \cup_{j\in J} \mathrm{ftv}([\![\sigma_j]\!]_{\mathrm{type}}) \\
& = & \mathrm{ftv}(\tau) \cup_{j\in J} \emptyset \\
& = & \mathrm{ftv}(\tau) \\
\mathrm{ftv}([\![\sigma]\!]_{\mathrm{sig}}(\tau)) & = & \mathrm{ftv}(\tau \rightarrow [\![\sigma]\!]_{\mathrm{type}}) \\
& = & \mathrm{ftv}(\tau) \cup \mathrm{ftv}([\![\sigma]\!]_{\mathrm{type}}) \\
& = & \mathrm{ftv}(\tau) \cup \emptyset \\
& = & \mathrm{ftv}(\tau)
\end{array}$$

□

Theorem C.2 (Subtyping Preservation)

- *If* $\vdash_{\mathcal{O}} \tau_1 \leq \tau_2$ *then* $\emptyset; \emptyset \vdash_{\mathcal{F}} [\![\tau_1]\!]_{\mathrm{type}} \leq [\![\tau_2]\!]_{\mathrm{type}}$.
- *If* $\vdash_{\mathcal{O}} \mathsf{objt}\ r_1 \leq \mathsf{objt}\ r_2$ *then* $\alpha; \emptyset \vdash_{\mathcal{F}} [\![r_1]\!]_{\mathrm{mt}}(\alpha, \rightarrow) \leq [\![r_2]\!]_{\mathrm{mt}}(\alpha, \rightarrow)$ *and* $\alpha; \emptyset \vdash_{\mathcal{F}} [\![r_1]\!]_{\mathrm{full}}(\alpha) \leq [\![r_2]\!]_{\mathrm{full}}(\alpha)$.
- *If* $\vdash_{\mathcal{O}} s_1 \leq s_2$ *then* $\alpha; \emptyset \vdash_{\mathcal{F}} [\![s_1]\!]_{\mathrm{sig}}(\alpha) \leq [\![s_2]\!]_{\mathrm{sig}}(\alpha)$.

Proof: By induction on the source derivation. The third item follows from the first item and Context Strengthening by the following derivation:

$$\text{(subfun)}\ \dfrac{\text{(subr)}\ \dfrac{\alpha \vdash_{\mathcal{F}} \alpha = \alpha}{\alpha; \emptyset \vdash_{\mathcal{F}} \alpha \leq \alpha} \quad \alpha; \emptyset \vdash_{\mathcal{F}} [\![s_1]\!]_{\mathrm{type}} \leq [\![s_2]\!]_{\mathrm{type}}}{\alpha; \emptyset \vdash_{\mathcal{F}} [\![s_1]\!]_{\mathrm{sig}}(\alpha) \leq [\![s_2]\!]_{\mathrm{sig}}(\alpha)}$$

To prove the other two items, consider the last rule used in the derivation:

(subtemp): In this case $\tau_1 = \mathsf{temp}\ r = \tau_2$. By Lemma C.1 $\emptyset \vdash_{\mathcal{F}} [\![\tau]\!]_{\mathrm{type}}$. The result follows by Derived Judgements and (subr).

(subobj): In this case $\tau_k = \mathsf{obj}\ r_k$, $r_k = [m_i{:}s_{ki}; f_k{:}\sigma_j]_{i\in I_k, j\in J_k}$, I_1 is a prefix of I_2, J_1 is a prefix of J_2, and $\vdash_{\mathcal{O}} s_{1i} \le s_{2i}$ for $i \in I_2$.

First I show that $\alpha;\emptyset \vdash_{\mathcal{F}} [\![r_1]\!]_{\text{mt}}(\alpha,\rightarrow) \le [\![r_2]\!]_{\text{mt}}(\alpha,\rightarrow)$. By the induction hypothesis $\alpha;\emptyset \vdash_{\mathcal{F}} [\![s_{1i}]\!]_{\text{sig}}(\alpha) \le [\![s_{2i}]\!]_{\text{sig}}(\alpha)$. By Lemma C.1, $\alpha \vdash_{\mathcal{F}} [\![s_{1i}]\!]_{\text{sig}}(\alpha)$. So:

$$\text{(subtup)}\ \dfrac{i \in I_2 : \dfrac{\alpha;\emptyset \vdash_{\mathcal{F}} [\![s_{1i}]\!]_{\text{sig}}(\alpha) \le [\![s_{2i}]\!]_{\text{sig}}(\alpha)}{\alpha;\emptyset \vdash_{\mathcal{F}} [\![s_{1i}]\!]_{\text{sig}}(\alpha)^+ \le [\![s_{2i}]\!]_{\text{sig}}(\alpha)^+} \quad i \in I_1 - I_2 : \alpha \vdash_{\mathcal{F}} [\![s_{1i}]\!]_{\text{sig}}(\alpha)}{\alpha;\emptyset \vdash_{\mathcal{F}} [\![r_1]\!]_{\text{mt}}(\alpha,\rightarrow) \le [\![r_2]\!]_{\text{mt}}(\alpha,\rightarrow)}$$

Second I show that $\alpha;\emptyset \vdash_{\mathcal{F}} [\![r_1]\!]_{\text{full}}(\alpha) \le [\![r_2]\!]_{\text{full}}(\alpha)$, thus establishing the second item. By the previous result and (subcov), $\alpha;\emptyset \vdash_{\mathcal{F}} [\![r_1]\!]_{\text{mt}}(\alpha,\rightarrow)^+ \le [\![r_2]\!]_{\text{mt}}(\alpha,\rightarrow)^+$, call this A. By Lemma C.1, Context Strengthening, and Derived Judgements, $\alpha \vdash_{\mathcal{F}} [\![\sigma_j]\!]_{\text{type}} = [\![\sigma_j]\!]_{\text{type}}$ for $j \in J_2$. By Lemma C.1 and Context Strengthening, $\alpha \vdash_{\mathcal{F}} [\![\sigma_j]\!]_{\text{type}} =$ for $j \in J_1 - J_2$. The derivation is:

$$\text{(subtup)}\ \dfrac{A \quad j \in J_2 : \dfrac{\alpha \vdash_{\mathcal{F}} [\![\sigma_j]\!]_{\text{type}} = [\![\sigma_j]\!]_{\text{type}}}{\alpha;\emptyset \vdash_{\mathcal{F}} [\![\sigma_j]\!]^{\circ}_{\text{type}} \le [\![\sigma_j]\!]^{\circ}_{\text{type}}} \quad j \in J_1 - J_2 : \alpha \vdash_{\mathcal{F}} \sigma_j}{\alpha;\emptyset \vdash_{\mathcal{F}} [\![r_1]\!]_{\text{full}}(\alpha) \le [\![r_2]\!]_{\text{full}}(\alpha)}$$

Finally the first item follows by (subself).

□

Lemma C.3 *If* $\vdash_{\mathcal{O}} r_1 \le r_2$ *then* $\alpha; \alpha \le [\![r_1]\!]_{\text{full}}(\alpha) \vdash_{\mathcal{F}} \alpha \le [\![r_2]\!]_{\text{full}}(\alpha)$.

Proof: By Subtyping Preservation $\alpha;\emptyset \vdash_{\mathcal{F}} [\![r_1]\!]_{\text{full}}(\alpha) \le [\![r_2]\!]_{\text{full}}(\alpha)$. By Context Strengthening $\alpha; \alpha \le [\![r_1]\!]_{\text{full}}(\alpha) \vdash_{\mathcal{F}} [\![r_1]\!]_{\text{full}}(\alpha) \le [\![r_2]\!]_{\text{full}}(\alpha)$. The result follows by (subtv) and (subt). □

Theorem C.4 (Typing Preservation) *If* $\Gamma \vdash_{\mathcal{O}} e : \tau$ *then* $\emptyset;\emptyset;[\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{F}} [\![e]\!]_{\text{exp}} : [\![\tau]\!]_{\text{type}}$. *If* $\Gamma \vdash^{\mathsf{M}}_{\mathcal{O}} M : \mathsf{obj}\ r \rhd s$ *then* $\alpha; \alpha \le [\![r]\!]_{\text{full}}(\alpha); [\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{F}} [\![M]\!]_{\text{mth}}(\alpha) : [\![s]\!]_{\text{sig}}(\alpha)$.

Proof: By mutual induction on the derivations. Throughout this proof, I will use the abbreviations for r, r_1, r_2, and s''_i at the bottom of Figure 7. It is easy to establish that $\vdash_{\mathcal{O}} r_i \le r$ for $i \in \{1,2\}$. Consider the last rule used:

(subsume): In this case $\Gamma \vdash_{\mathcal{O}} e : \tau'$ and $\vdash_{\mathcal{O}} \tau' \le \tau$. There are two subcases according to which rule is used to translate the derivation. Case 1, the translation of $\Gamma \vdash_{\mathcal{O}} e : \tau$ is the translation of $\Gamma \vdash_{\mathcal{O}} e : \tau'$. By the induction hypothesis $\emptyset;\emptyset;[\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{F}} [\![e]\!]_{\text{exp}} : [\![\tau']\!]_{\text{type}}$. By Subtyping Preservation, $\emptyset;\emptyset \vdash_{\mathcal{F}} [\![\tau']\!]_{\text{type}} \le [\![\tau]\!]_{\text{type}}$. The result follows by subsumption. Case 2, the translation of $\Gamma \vdash_{\mathcal{O}} e : \tau'$ is $\mathsf{pack}\ e', \sigma\ \mathsf{as}\ [\![\tau']\!]_{\text{type}}$ and the translation of $\Gamma \vdash_{\mathcal{O}} e : \tau$ is $\mathsf{pack}\ e', \sigma\ \mathsf{as}\ [\![\tau]\!]_{\text{type}}$: By the induction hypothesis $\emptyset;\emptyset;[\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{F}} \mathsf{pack}\ e', \tau\ \mathsf{as}\ [\![\tau']\!]_{\text{type}} : [\![\tau']\!]_{\text{type}}$. By inspection of the rules, it follows that $\emptyset;\emptyset;[\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{F}} e' : \sigma$ and $\emptyset;\emptyset \vdash_{\mathcal{F}} \sigma \le [\![r']\!]_{\text{full}}(\alpha)\{\alpha := \sigma\}$ where $\tau' = \mathsf{objt}\ r'$. By Subtyping Preservation, $\alpha;\emptyset \vdash_{\mathcal{F}} [\![r']\!]_{\text{full}}(\alpha) \le [\![r]\!]_{\text{full}}(\alpha)$ where $\tau = \mathsf{objt}\ r$. By Type Substitution, $\emptyset;\emptyset \vdash_{\mathcal{F}} [\![r']\!]_{\text{full}}(\alpha)\{\alpha := \sigma\} \le [\![r]\!]_{\text{full}}(\alpha)\{\alpha := \sigma\}$. By (subt), $\emptyset;\emptyset \vdash_{\mathcal{F}} \sigma \le [\![r]\!]_{\text{full}}(\alpha)\{\alpha := \sigma\}$. By (pack), $\emptyset;\emptyset;[\![\Gamma]\!]_{\text{ctxt}} \vdash_{\mathcal{F}} \mathsf{pack}\ e', \sigma\ \mathsf{as}\ [\![\tau']\!]_{\text{type}} : [\![\tau]\!]_{\text{type}}$ as required.

(var): In this case $e = x$ and $\tau = \Gamma(x)$. The result follows by (var).

(et): In this case $e = \mathsf{et}$ and $\tau = \mathsf{temp}[;]$. So:

$$\dfrac{\dfrac{\dfrac{\alpha \vdash_{\mathcal{F}} \langle \mathsf{mt}{:}\langle\rangle^{\rightarrow+}\rangle^{\rightarrow} \quad \alpha; \alpha \leq \langle \mathsf{mt}{:}\langle\rangle^{\rightarrow+}\rangle^{\rightarrow}; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \langle\rangle : \langle\rangle^{\rightarrow}}{\emptyset; \emptyset; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \forall\alpha \leq \langle \mathsf{mt}{:}\langle\rangle^{\rightarrow+}\rangle^{\rightarrow}.\langle\rangle : \forall\alpha \leq \langle \mathsf{mt}{:}\langle\rangle^{\rightarrow+}\rangle^{\rightarrow}.\langle\rangle^{\rightarrow}}}{\emptyset; \emptyset; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \forall\alpha \leq \langle \mathsf{mt}{:}\langle\rangle^{\rightarrow+}\rangle^{\rightarrow}.\langle\rangle : \forall\alpha \leq \langle \mathsf{mt}{:}\langle\rangle^{\rightarrow+}\rangle^{\rightarrow}.\langle\rangle^{\circ}}}{\emptyset; \emptyset; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \llbracket e\rrbracket_{\mathrm{exp}} : \llbracket\tau\rrbracket_{\mathrm{type}}}$$

(addfield): In this case $e = e' + f{:}\sigma$, $\tau = \mathsf{temp}\ r_1$, and $\Delta; B; \Gamma \vdash_{\mathcal{O}} e' : \mathsf{temp}\ r$. By the induction hypothesis, $\emptyset; \emptyset; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \llbracket e'\rrbracket_{\mathrm{exp}} : \llbracket\mathsf{temp}\ r\rrbracket_{\mathrm{type}}$. By Lemma C.1, $\alpha \vdash_{\mathcal{F}} \llbracket r_1\rrbracket_{\mathrm{full}}(\alpha)$. Let $B' = \alpha \leq \llbracket r_1\rrbracket_{\mathrm{full}}(\alpha)$. By Lemma C.3 $\alpha; B' \vdash_{\mathcal{F}} \alpha \leq \llbracket r\rrbracket_{\mathrm{full}}(\alpha)$. Note that $\llbracket r_1\rrbracket_{\mathrm{mt}}(\alpha, \circ) = \llbracket r\rrbracket_{\mathrm{mt}}(\alpha, \circ)$ because r_1 adds no methods nor changes their signatures. Then:

$$\dfrac{\alpha \vdash_{\mathcal{F}} \llbracket r_1\rrbracket_{\mathrm{full}}(\alpha) \qquad \dfrac{\alpha; B'; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \llbracket e'\rrbracket_{\mathrm{exp}} : \llbracket\mathsf{temp}\ r\rrbracket_{\mathrm{type}} \quad \alpha; B' \vdash_{\mathcal{F}} \alpha \leq \llbracket r\rrbracket_{\mathrm{full}}(\alpha)}{\alpha; B'; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \llbracket e'\rrbracket_{\mathrm{exp}}[\alpha] : \llbracket r_1\rrbracket_{\mathrm{mt}}(\alpha, \circ)}}{\emptyset; \emptyset; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \llbracket e\rrbracket_{\mathrm{exp}} : \llbracket\tau\rrbracket_{\mathrm{type}}}$$

(aometh): In this case $e = e' \leftarrow\!\!+[m_k = M_k]_{k \in K}$, $\tau = \mathsf{temp}\ r_2$, $\Gamma \vdash_{\mathcal{O}} e' : \mathsf{temp}\ r$, and $\Gamma \vdash^{\mathsf{M}}_{\mathcal{O}} M_k : \mathsf{obj}\ r_3 \rhd s'_k$ for $k \in K$. Let:

$$\begin{array}{rcl}
B' & = & \alpha \leq \llbracket r_2\rrbracket_{\mathrm{full}}(\alpha) \\
e_1 & = & \llbracket e\rrbracket_{\mathrm{exp}}[\alpha] +_{k \in K-I} m_k = \llbracket M_k\rrbracket_{\mathrm{mth}}(\alpha) \\
e_2 & = & e_1.m_k \leftarrow_{k \in K \cap I} \llbracket M_k\rrbracket_{\mathrm{mth}}(\alpha) \\
\tau_1 & = & \langle m_i{:}\llbracket s_i\rrbracket_{\mathrm{sig}}(\alpha)^{+}, m_k{:}\llbracket s'_k\rrbracket_{\mathrm{sig}}(\alpha)^{\circ}\rangle^{\circ}_{i \in I, k \in K-I} \\
\tau_2 & = & \langle m_i{:}\llbracket s''_i\rrbracket_{\mathrm{sig}}(\alpha)^{\phi'_i}, m_k{:}\llbracket s'_k\rrbracket_{\mathrm{sig}}(\alpha)^{\circ}\rangle^{\circ}_{i \in I, k \in K-I} \\
\phi'_i & = & \begin{cases} + & i \notin K \\ \circ & i \in K \end{cases}
\end{array}$$

Similar to the previous case, $\alpha \vdash_{\mathcal{F}} \llbracket r_3\rrbracket_{\mathrm{full}}(\alpha)$ and $\alpha; B'; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \llbracket e'\rrbracket_{\mathrm{exp}}[\alpha] : \llbracket r\rrbracket_{\mathrm{mt}}(\alpha, \circ)$. By the induction hypothesis $\alpha; B'; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \llbracket M_k\rrbracket_{\mathrm{mth}}(\alpha) : \llbracket s'_i\rrbracket_{\mathrm{sig}}(\alpha)$. It is easy to establish $\alpha; B' \vdash_{\mathcal{F}} \tau_2 \leq \llbracket r_2\rrbracket_{\mathrm{mt}}(\alpha, \circ)$. Then:

$$\dfrac{\alpha \vdash_{\mathcal{F}} \llbracket r_3\rrbracket_{\mathrm{full}}(\alpha) \quad \dfrac{A \quad \alpha; B' \vdash_{\mathcal{F}} \tau_2 \leq \llbracket r_3\rrbracket_{\mathrm{mt}}(\alpha, \circ)}{\alpha; B'; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} e_2 : \llbracket r_3\rrbracket_{\mathrm{mt}}(\alpha, \circ)}}{\emptyset; \emptyset; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \llbracket e\rrbracket_{\mathrm{exp}} : \llbracket\tau\rrbracket_{\mathrm{type}}}$$

Where A is:

$$\dfrac{B \quad k \in K \cap I : \alpha; B'; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \llbracket M_k\rrbracket_{\mathrm{mth}}(\alpha) : \llbracket s'_k\rrbracket_{\mathrm{sig}}(\alpha)}{\alpha; B'; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} e_2 : \tau_2}$$

Where B is:

$$\dfrac{\begin{array}{c} \alpha; B'; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \llbracket e'\rrbracket_{\mathrm{exp}}[\alpha] : \llbracket r\rrbracket_{\mathrm{mt}}(\alpha, \circ) \\ k \in K - I : \alpha; B'; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \llbracket M_k\rrbracket_{\mathrm{mth}}(\alpha) : \llbracket s'_k\rrbracket_{\mathrm{sig}}(\alpha) \end{array}}{\alpha; B'; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} e_1 : \tau_1}$$

(inst): In this case $e = \mathsf{new}\ e'[f_j = e_j]_{j \in J}$, $\tau = \mathsf{obj}\ r$, $\Gamma \vdash_{\mathcal{O}} e' : \mathsf{temp}\ r$, and $\Gamma \vdash_{\mathcal{O}} e_j : \sigma_j$. By the induction hypothesis $\emptyset; \emptyset; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \llbracket e'\rrbracket_{\mathrm{exp}} : \llbracket\mathsf{temp}\ r\rrbracket_{\mathrm{type}}$ and $\emptyset; \emptyset; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \llbracket e_j\rrbracket_{\mathrm{exp}} : \llbracket\sigma_j\rrbracket_{\mathrm{type}}$. Let $\tau' = \mathsf{rec}\ \alpha.\llbracket r\rrbracket_{\mathrm{full}}(\alpha)$. By Lemma C.1 and Derived Judgements in the target language twice, $\emptyset \vdash_{\mathcal{F}} \tau'$. By Lemma C.1, Derived Judgements in the target language, and (subr), $\emptyset;\emptyset \vdash_{\mathcal{F}} [\![s_i]\!]_{\mathrm{sig}}(\tau') \leq [\![s_i]\!]_{\mathrm{sig}}(\tau')$ and $\emptyset;\emptyset \vdash_{\mathcal{F}} [\![\sigma_j]\!]_{\mathrm{type}} \leq [\![\sigma_j]\!]_{\mathrm{type}}$. Let $\tau'' = \langle \mathsf{mt} = [\![r]\!]_{\mathrm{mt}}(\tau',\circ)^{\circ}, f_j{:}[\![\sigma_j]\!]^{\circ}_{\mathrm{type}}\rangle^{\circ}_{j\in J}$. Let D be the following derivation:

$$\dfrac{D' \quad \dfrac{\text{(eq1)}\ \dfrac{\dfrac{\emptyset \vdash_{\mathcal{F}} \tau'}{\emptyset \vdash_{\mathcal{F}} \tau' = [\![r]\!]_{\mathrm{full}}(\tau')}}{\emptyset \vdash_{\mathcal{F}} [\![r]\!]_{\mathrm{full}}(\tau') = \tau'}}{\emptyset;\emptyset \vdash_{\mathcal{F}} [\![r]\!]_{\mathrm{full}}(\tau') \leq \tau'}}{\emptyset;\emptyset \vdash_{\mathcal{F}} \tau'' \leq \tau'}$$

where D' is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\emptyset;\emptyset \vdash_{\mathcal{F}} [\![s_i]\!]_{\mathrm{sig}}(\tau') \leq [\![s_i]\!]_{\mathrm{sig}}(\tau')}{\emptyset;\emptyset \vdash_{\mathcal{F}} [\![s_i]\!]_{\mathrm{sig}}(\tau')^{+} \leq [\![s_i]\!]_{\mathrm{sig}}(\tau')^{+}}}{\emptyset;\emptyset \vdash_{\mathcal{F}} [\![r]\!]_{\mathrm{mt}}(\tau',\circ) \leq [\![r]\!]_{\mathrm{mt}}(\tau',+)}}{\emptyset;\emptyset \vdash_{\mathcal{F}} [\![r]\!]_{\mathrm{mt}}(\tau',\circ)^{\circ} \leq [\![r]\!]_{\mathrm{mt}}(\tau',+)^{+}} \quad \dfrac{\emptyset;\emptyset \vdash_{\mathcal{F}} [\![\sigma_j]\!]_{\mathrm{type}} \leq [\![\sigma_j]\!]_{\mathrm{type}}}{\emptyset;\emptyset \vdash_{\mathcal{F}} [\![\sigma_j]\!]^{\circ}_{\mathrm{type}} \leq [\![\sigma_j]\!]^{+}_{\mathrm{type}}}}{\emptyset;\emptyset \vdash_{\mathcal{F}} \tau'' \leq [\![r]\!]_{\mathrm{full}}(\tau')}$$

Let $\Gamma' = [\![\Gamma]\!]_{\mathrm{ctxt}}, x{:}[\![\mathsf{temp}\ r]\!]_{\mathrm{type}}, x_j{:}[\![\sigma_j]\!]_{\mathrm{type}}$. Then:

$$\dfrac{\emptyset;\emptyset;[\![\Gamma]\!]_{\mathrm{ctxt}} \vdash_{\mathcal{F}} [\![e']\!]_{\mathrm{exp}} : [\![\mathsf{temp}\ r]\!]_{\mathrm{type}} \quad \emptyset;\emptyset;[\![\Gamma]\!]_{\mathrm{ctxt}} \vdash_{\mathcal{F}} [\![e_j]\!]_{\mathrm{exp}} : [\![\sigma_j]\!]_{\mathrm{type}} \quad A}{\emptyset;\emptyset;[\![\Gamma]\!]_{\mathrm{ctxt}} \vdash_{\mathcal{F}} [\![e]\!]_{\mathrm{exp}} : [\![\tau]\!]_{\mathrm{type}}}$$

where A is:

$$\dfrac{\dfrac{\dfrac{B \quad \emptyset;\emptyset;\Gamma' \vdash_{\mathcal{F}} x_j : [\![\sigma_j]\!]_{\mathrm{type}}}{\emptyset;\emptyset;\Gamma' \vdash_{\mathcal{F}} \langle \mathsf{mt} = x[\tau'], f_j = x_j\rangle_{j\in J} : \tau''} \quad D}{\emptyset;\emptyset;\Gamma' \vdash_{\mathcal{F}} \langle \mathsf{mt} = x[\tau'], f_j = x_j\rangle_{j\in J} : \tau'} \quad \dfrac{\text{(eq1)}\ \dfrac{\ }{\emptyset \vdash_{\mathcal{F}} \tau' = [\![r]\!]_{\mathrm{full}}(\tau')}}{\emptyset;\emptyset \vdash_{\mathcal{F}} \tau' \leq [\![r]\!]_{\mathrm{full}}(\tau')}}{\emptyset;\emptyset;\Gamma' \vdash_{\mathcal{F}} \mathsf{pack}\ \langle \mathsf{mt} = x[\tau'], f_j = x_j\rangle_{j\in J}, \tau'\ \mathsf{as}\ [\![\tau]\!]_{\mathrm{type}} : [\![\tau]\!]_{\mathrm{type}}}$$

and B is:

$$\dfrac{\emptyset;\emptyset;[\![\Gamma]\!]_{\mathrm{ctxt}} \vdash_{\mathcal{F}} x : [\![\mathsf{temp}\ r]\!]_{\mathrm{type}} \quad \dfrac{\text{(eq1)}\ \dfrac{\emptyset \vdash_{\mathcal{F}} \tau'}{\emptyset \vdash_{\mathcal{F}} \tau' = [\![r]\!]_{\mathrm{full}}(\tau')}}{\emptyset;\emptyset \vdash_{\mathcal{F}} \tau' \leq [\![r]\!]_{\mathrm{full}}(\tau')}}{\emptyset;\emptyset;[\![\Gamma]\!]_{\mathrm{ctxt}} \vdash_{\mathcal{F}} x[\tau'] : [\![r]\!]_{\mathrm{mt}}(\tau',\circ)}$$

(invoke): In this case $e = e'.m_k$, $\tau = s_k$, $\Gamma \vdash_{\mathcal{O}} e' : \mathsf{obj}\ r$. Let $B' = \alpha \leq [\![r]\!]_{\mathrm{full}}(\alpha)$ and $\Gamma' = [\![\Gamma]\!]_{\mathrm{ctxt}}, x{:}\alpha$. By the induction hypothesis, $\emptyset;\emptyset;[\![\Gamma]\!]_{\mathrm{ctxt}} \vdash_{\mathcal{F}} [\![e']\!]_{\mathrm{exp}} : [\![\mathsf{obj}\ r]\!]_{\mathrm{type}}$. By Lemma C.1, $\emptyset \vdash_{\mathcal{F}} [\![\tau]\!]_{\mathrm{type}}$. Then:

$$\dfrac{\emptyset;\emptyset;[\![\Gamma]\!]_{\mathrm{ctxt}} \vdash_{\mathcal{F}} [\![e']\!]_{\mathrm{exp}} : [\![\mathsf{obj}\ r]\!]_{\mathrm{type}} \quad A \quad \emptyset \vdash_{\mathcal{F}} [\![\tau]\!]_{\mathrm{type}}}{\emptyset;\emptyset;[\![\Gamma]\!]_{\mathrm{ctxt}} \vdash_{\mathcal{F}} [\![e]\!]_{\mathrm{exp}} : [\![\tau]\!]_{\mathrm{type}}}$$

A is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\alpha;B';\Gamma' \vdash_{\mathcal{F}} x : \alpha \\ \alpha;B' \vdash_{\mathcal{F}} \alpha \leq [\![r]\!]_{\mathrm{full}}(\alpha)\end{array}}{\alpha;B';\Gamma' \vdash_{\mathcal{F}} x : [\![r]\!]_{\mathrm{full}}(\alpha)}}{\alpha;B';\Gamma' \vdash_{\mathcal{F}} x.\mathsf{mt} : [\![r]\!]_{\mathrm{mt}}(\alpha,\rightarrow)}}{\alpha;B';\Gamma' \vdash_{\mathcal{F}} x.\mathsf{mt}.m_k : \alpha \rightarrow [\![\tau]\!]_{\mathrm{type}}} \quad \alpha;B';\Gamma' \vdash_{\mathcal{F}} x : \alpha}{\alpha;B';\Gamma' \vdash_{\mathcal{F}} x.\mathsf{mt}.m_k\ x : [\![\tau]\!]_{\mathrm{type}}}$$

(select): In this case $e = e'.f_k$, $\tau = \sigma_k$, and $\Gamma \vdash_{\mathcal{O}} e' : \mathsf{obj}\ r$. By the induction hypothesis $\emptyset;\emptyset;\llbracket\Gamma\rrbracket_{\text{ctxt}} \vdash_{\mathcal{F}} \llbracket e'\rrbracket_{\text{exp}} : \llbracket\mathsf{obj}\ r\rrbracket_{\text{type}}$. Lemma C.1, $\Delta \vdash_{\mathcal{F}} \llbracket\tau\rrbracket_{\text{type}}$. Let $B' = \llbracket B\rrbracket_{\text{ctxt}}, \alpha \leq \llbracket r\rrbracket_{\text{full}}(\alpha)$. Then:

$$\frac{\begin{array}{c}\emptyset;\emptyset;\llbracket\Gamma\rrbracket_{\text{ctxt}} \vdash_{\mathcal{F}} \llbracket e'\rrbracket_{\text{exp}} : \llbracket\mathsf{obj}\ r\rrbracket_{\text{type}}\\ \emptyset \vdash_{\mathcal{F}} \llbracket\tau\rrbracket_{\text{type}}\end{array} \quad \dfrac{\dfrac{\begin{array}{c}\alpha;B';\llbracket\Gamma\rrbracket_{\text{ctxt}}, x{:}\alpha \vdash_{\mathcal{F}} x : \alpha\\ \alpha;B' \vdash_{\mathcal{F}} \alpha \leq \llbracket r\rrbracket_{\text{full}}(\alpha)\end{array}}{\alpha;B';\llbracket\Gamma\rrbracket_{\text{ctxt}}, x{:}\alpha \vdash_{\mathcal{F}} x : \llbracket r\rrbracket_{\text{full}}(\alpha)}}{\alpha;B';\llbracket\Gamma\rrbracket_{\text{ctxt}}, x{:}\alpha \vdash_{\mathcal{F}} x.f_k : \llbracket\sigma_k\rrbracket_{\text{type}}}}{\emptyset;\emptyset;\llbracket\Gamma\rrbracket_{\text{ctxt}} \vdash_{\mathcal{F}} \llbracket e\rrbracket_{\text{exp}} : \llbracket\tau\rrbracket_{\text{type}}}$$

(update): In this case $e = e_1.f_k := e_2$, $\tau = \mathsf{obj}\ r$, $\Gamma \vdash_{\mathcal{O}} e_1 : \tau$, and $\Gamma \vdash_{\mathcal{O}} e_2 : \sigma_k$. By the induction hypothesis, $\emptyset;\emptyset;\llbracket\Gamma\rrbracket_{\text{ctxt}} \vdash_{\mathcal{F}} \llbracket e_1\rrbracket_{\text{exp}} : \llbracket\mathsf{obj}\ r\rrbracket_{\text{type}}$ and $\emptyset;\emptyset;\llbracket\Gamma\rrbracket_{\text{ctxt}} \vdash_{\mathcal{F}} \llbracket e_2\rrbracket_{\text{exp}} : \llbracket\sigma_k\rrbracket_{\text{type}}$. Then:

$$\frac{\emptyset;\emptyset;\llbracket\Gamma\rrbracket_{\text{ctxt}} \vdash_{\mathcal{F}} \llbracket e_1\rrbracket_{\text{exp}} : \llbracket\tau\rrbracket_{\text{type}} \quad \emptyset;\emptyset;\llbracket\Gamma\rrbracket_{\text{ctxt}} \vdash_{\mathcal{F}} \llbracket e_2\rrbracket_{\text{exp}} : \llbracket\sigma_k\rrbracket_{\text{type}} \quad A}{\emptyset;\emptyset;\llbracket\Gamma\rrbracket_{\text{ctxt}} \vdash_{\mathcal{F}} \llbracket e\rrbracket_{\text{exp}} : \llbracket\tau\rrbracket_{\text{type}}}$$

Let $\Gamma' = \llbracket\Gamma\rrbracket_{\text{ctxt}}, x_1{:}\llbracket\tau\rrbracket_{\text{type}}, x_2{:}\llbracket\sigma_k\rrbracket_{\text{type}}$ and $B' = \alpha \leq \llbracket r\rrbracket_{\text{full}}(\alpha)$. Lemma C.1, $\Delta \vdash_{\mathcal{F}} \llbracket\tau\rrbracket_{\text{type}}$. Then A is:

$$\frac{\begin{array}{c}\emptyset;\emptyset;\Gamma' \vdash_{\mathcal{F}} x_1 : \llbracket\tau\rrbracket_{\text{type}}\\ \emptyset \vdash_{\mathcal{F}} \llbracket\tau\rrbracket_{\text{type}}\end{array} \quad \dfrac{B \quad \alpha;B' \vdash_{\mathcal{F}} \alpha \leq \llbracket r\rrbracket_{\text{full}}(\alpha)}{\alpha;B';\Gamma', x{:}\alpha \vdash_{\mathcal{F}} \mathsf{pack}\ x.f := x_2, \alpha\ \mathsf{as}\ \llbracket\tau\rrbracket_{\text{type}} : \llbracket\tau\rrbracket_{\text{type}}}}{\emptyset;\emptyset;\Gamma' \vdash_{\mathcal{F}} \mathsf{unpack}\ \alpha.\llbracket r\rrbracket_{\text{full}}(\alpha), x = x_1\ \mathsf{in}\ x.f_k := x_2 : \llbracket\tau\rrbracket_{\text{type}}}$$

and B is:

$$\frac{\alpha;B';\Gamma', x{:}\alpha \vdash_{\mathcal{F}} x : \alpha \quad \alpha;B' \vdash_{\mathcal{F}} \alpha \leq \llbracket r\rrbracket_{\text{full}}(\alpha) \quad \alpha;B';\Gamma', x{:}\alpha \vdash_{\mathcal{F}} x_2 : \llbracket\sigma_k\rrbracket_{\text{type}}}{\alpha;B';\Gamma', x{:}\alpha \vdash_{\mathcal{F}} x.f := x_2 : \alpha}$$

(method): In this case $M = x.b{:}\sigma$, $s = \sigma$, and $x{:}\alpha \vdash_{\mathcal{O}} b : \sigma$. Let $B' = \alpha \leq \llbracket r\rrbracket_{\text{full}}(\alpha)$ and $\Gamma' = \llbracket\Gamma\rrbracket_{\text{ctxt}}, x'{:}\alpha, x{:}\llbracket\mathsf{objt}\ r\rrbracket_{\text{type}}$. By the induction hypothesis $\emptyset;\emptyset;\llbracket\Gamma\rrbracket_{\text{ctxt}}, x{:}\llbracket\mathsf{objt}\ r\rrbracket_{\text{type}} \vdash_{\mathcal{F}} \llbracket b\rrbracket_{\text{exp}} : \llbracket\sigma\rrbracket_{\text{type}}$. By Context Strengthening, $\alpha;B';\Gamma' \vdash_{\mathcal{F}} \llbracket b\rrbracket_{\text{exp}} : \llbracket\sigma\rrbracket_{\text{type}}$. Then:

$$\frac{\alpha \vdash_{\mathcal{F}} \alpha \quad \dfrac{A \quad \alpha;B';\Gamma' \vdash_{\mathcal{F}} \llbracket b\rrbracket_{\text{exp}} : \llbracket\sigma\rrbracket_{\text{type}}}{\alpha;B';\llbracket\Gamma\rrbracket_{\text{ctxt}}, x'{:}\alpha \vdash_{\mathcal{F}} \mathsf{let}\ x = \mathsf{pack}\ x', \alpha\ \mathsf{as}\ \llbracket\mathsf{objt}\ r\rrbracket_{\text{type}}\ \mathsf{in}\ \llbracket b\rrbracket_{\text{exp}} : \llbracket\sigma\rrbracket_{\text{type}}}}{\alpha;B';\llbracket\Gamma\rrbracket_{\text{ctxt}} \vdash_{\mathcal{F}} \llbracket M\rrbracket_{\text{mth}}(\alpha) : \llbracket s\rrbracket_{\text{sig}}(\alpha)}$$

where A is:

$$\frac{\alpha;B';\llbracket\Gamma\rrbracket_{\text{ctxt}}, x'{:}\alpha \vdash_{\mathcal{F}} x' : \alpha \quad \alpha;B' \vdash_{\mathcal{F}} \alpha \leq \llbracket r\rrbracket_{\text{full}}(\alpha)}{\alpha;B';\llbracket\Gamma\rrbracket_{\text{ctxt}}, x'{:}\alpha \vdash_{\mathcal{F}} \mathsf{pack}\ x', \alpha\ \mathsf{as}\ \llbracket\mathsf{objt}\ r\rrbracket_{\text{type}} : \llbracket\mathsf{objt}\ r\rrbracket_{\text{type}}}$$

(meth-sub): In this case $\Gamma \vdash^{\mathsf{M}}_{\mathcal{O}} M : \sigma' \rhd s$ and $\vdash_{\mathcal{O}} \sigma' \leq \mathsf{objt}\ r$. Note that σ' must be of the form $\mathsf{objt}\ r'$. By the induction hypothesis $\alpha;\alpha \leq \llbracket r'\rrbracket_{\text{full}}(\alpha);\llbracket\Gamma\rrbracket_{\text{ctxt}} \vdash_{\mathcal{F}} \llbracket M\rrbracket_{\text{mth}}(\alpha) : \llbracket s\rrbracket_{\text{sig}}(\alpha)$. By Subtyping Preservation and Context Strengthening in $\mathcal{F}_{\mathsf{self}}$, $\alpha;\alpha \leq \llbracket r'\rrbracket_{\text{full}}(\alpha) \vdash_{\mathcal{F}} \llbracket r'\rrbracket_{\text{full}}(\alpha) \leq \llbracket r\rrbracket_{\text{full}}(\alpha)$. By Context Strengthening in $\mathcal{F}_{\mathsf{self}}$, $\alpha;\alpha \leq \llbracket r'\rrbracket_{\text{full}}(\alpha);\llbracket\Gamma\rrbracket_{\text{ctxt}} \vdash_{\mathcal{F}} \llbracket M\rrbracket_{\text{mth}}(\alpha) \leq \llbracket s\rrbracket_{\text{sig}}(\alpha)$ as required.

(temp): In this case $e = \mathsf{temp}[m_i = M_i; f_j{:}\sigma_j]_{i\in I, j\in J}$, $\tau = \mathsf{tempt}\ r$, and $\Gamma \vdash^{\mathsf{M}}_{\mathcal{O}} M_i : \mathsf{objt}\ r \rhd s_i$. Let $B' = \alpha \le \llbracket r\rrbracket_{\mathrm{full}}(\alpha)$, $\tau_1 = \langle m_i{:}\llbracket s_i\rrbracket_{\mathrm{sig}}(\alpha, r)^{\circ}\rangle^{\circ}$, and $\tau_2 = \langle m_i{:}\llbracket s_i\rrbracket_{\mathrm{sig}}(\alpha, r)^{+}\rangle^{\circ}$. It is easy to establish that $\alpha; B' \vdash_{\mathcal{F}} \tau_1 \le \tau_2$. By Lemma C.1, $\alpha \vdash_{\mathcal{F}} \llbracket r\rrbracket_{\mathrm{full}}(\alpha)$. By the induction hypothesis, $\alpha; B'; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \llbracket M_i\rrbracket_{\mathrm{mth}}(\alpha) : \llbracket s_i\rrbracket_{\mathrm{sig}}(\alpha)$. Then:

$$\frac{\alpha \vdash_{\mathcal{F}} \llbracket r\rrbracket_{\mathrm{full}}(\alpha) \qquad \dfrac{\dfrac{\dfrac{\alpha; B'; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \llbracket M_i\rrbracket_{\mathrm{mth}}(\alpha) : \llbracket s_i\rrbracket_{\mathrm{sig}}(\alpha)}{\alpha; B'; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \langle m_i = \llbracket M_i\rrbracket_{\mathrm{mth}}(\alpha)\rangle_{i\in I} : \tau_1} \qquad \alpha; B' \vdash_{\mathcal{F}} \tau_1 \le \tau_2}{\alpha; B'; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \langle m_i = \llbracket M_i\rrbracket_{\mathrm{mth}}(\alpha)\rangle_{i\in I} : \tau_2}}{}}{\emptyset; \emptyset; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \llbracket e\rrbracket_{\mathrm{exp}} : \llbracket\tau\rrbracket_{\mathrm{type}}}$$

(obj): In this case $e = \mathsf{obj}[m_i = M_i; f_j = v_j]_{i\in I, j\in J}$, $\tau = \mathsf{objt}\ r$, $\Gamma \vdash^{\mathsf{M}}_{\mathcal{O}} M_i : \mathsf{objt}\ r \rhd s_i$, and $\Gamma \vdash_{\mathcal{O}} v_j : \sigma_j$. Let:

$$\begin{array}{rcl}
\tau' & = & \mathsf{rec}\ \alpha.\llbracket r\rrbracket_{\mathrm{full}}(\alpha) \\
\tau'' & = & \langle \mathsf{mt}{:}\llbracket r\rrbracket_{\mathrm{mt}}(\tau', \rightarrow)^{\circ}, f_j{:}\llbracket \sigma_j\rrbracket^{\circ}_{\mathrm{type}}\rangle_{j\in J} \\
w_1 & = & \langle m_i = \llbracket M_i\rrbracket_{\mathrm{mth}}(\tau')\rangle_{i\in I} \\
w_2 & = & \langle \mathsf{mt} = w_1, f_j = v_j\rangle_{j\in J}
\end{array}$$

By reasoning similar to the case for (inst), $\emptyset; \emptyset \vdash_{\mathcal{F}} \tau'' \le \tau'$ and $\emptyset; \emptyset \vdash_{\mathcal{F}} \tau' \le \llbracket r\rrbracket_{\mathrm{full}}(\tau')$. By reasoning similar to the case for (temp), $\alpha; \alpha \le \llbracket r\rrbracket_{\mathrm{full}}(\alpha) \vdash_{\mathcal{F}} \langle m_i = \llbracket M_i\rrbracket_{\mathrm{mth}}(\alpha)\rangle_{i\in I} : \llbracket r\rrbracket_{\mathrm{mt}}(\alpha, \rightarrow)$. By Type Substitution, $\emptyset; \emptyset \vdash_{\mathcal{F}} w_1 : \llbracket r\rrbracket_{\mathrm{mt}}(\tau', \rightarrow)$. By the induction hypothesis, $\emptyset; \emptyset; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \llbracket v_j\rrbracket_{\mathrm{exp}} : \llbracket\sigma_j\rrbracket_{\mathrm{type}}$. Then:

$$\frac{\dfrac{\dfrac{\begin{array}{c}\emptyset; \emptyset; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} w_1 : \llbracket r\rrbracket_{\mathrm{mt}}(\tau', \rightarrow) \\ \emptyset; \emptyset; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} v_j : \llbracket\sigma_j\rrbracket_{\mathrm{type}}\end{array}}{\emptyset; \emptyset; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} w_2 : \tau''} \qquad \emptyset; \emptyset \vdash_{\mathcal{F}} \tau'' \le \tau'}{\emptyset; \emptyset; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} w_2 : \tau'} \qquad \emptyset; \emptyset \vdash_{\mathcal{F}} \tau' \le \llbracket r\rrbracket_{\mathrm{full}}(\tau')}{\emptyset; \emptyset; \llbracket\Gamma\rrbracket_{\mathrm{ctxt}} \vdash_{\mathcal{F}} \llbracket e\rrbracket_{\mathrm{exp}} : \llbracket\tau\rrbracket_{\mathrm{type}}}$$

□

D Operational Correctness

Lemma D.1 $\llbracket e_1\{x := e_2\}\rrbracket_{\mathrm{exp}} = \llbracket e_1\rrbracket_{\mathrm{exp}}\{x := \llbracket e_2\rrbracket_{\mathrm{exp}}\}$

Proof: Technically this means that for typing derivations for $e_1\{x := e_2\}$ and $\llbracket e_1\rrbracket_{\mathrm{exp}}$ and $\llbracket e_2\rrbracket_{\mathrm{exp}}$ that correspond, the two translations are equal. In fact the derivations of $e_1\{x := e_2\}$ might contain different derivations for the different occurances of x, this means that $\llbracket e_1\rrbracket_{\mathrm{exp}}\{x := \llbracket e_2\rrbracket_{\mathrm{exp}}\}$ should be interpreted as substituted different translations for different occurances of x. Other than these technicalities, the result follows by induction on the derivation of e_1. □

Define $\llbracket E\rrbracket_{\mathrm{exp}}$ as $\llbracket E\rrbracket_{\mathrm{exp}} = \llbracket E\{x\}\rrbracket_{\mathrm{exp}}\{x := \{\}\}$. Technically this is a function from derivations of the form $x : \tau \vdash_{\mathcal{O}} E\{x\} : \sigma$ (which can be thought of as typing judgements for contexts) to target language terms with holes in them.

Lemma D.2 *If E is a template language context then $[\![E]\!]_{\mathrm{exp}}$ is a target language context.*

Proof: By induction on the structure of E and inspection of the translation. □

Lemma D.3 $[\![E\{e\}]\!]_{\mathrm{exp}} = [\![E]\!]_{\mathrm{exp}}\{[\![e]\!]_{\mathrm{exp}}\}$

Proof: Follows immediately by definition and Lemma D.1. Note that because there is exactly one occurance of the hole in E, the substitution in Lemma D.1 is of exactly one occurance. Thus the derivations of $E\{e\}$ correspond exactly to the derivations of $[\![E]\!]_{\mathrm{exp}}$ and $[\![e]\!]_{\mathrm{exp}}$. □

Lemma D.4 *For template language value v, $[\![v]\!]_{\mathrm{exp}}$ is a target language value.*

Proof: By induction on the structure of v and inspection of the translation. □

Theorem D.5 *If $\emptyset;\emptyset;\emptyset \vdash_{\mathcal{O}} e_1 : \tau$ and $e_1 \mapsto e_2$ then $[\![e_1]\!]_{\mathrm{exp}} \mapsto^{+} [\![e_2]\!]_{\mathrm{exp}}$.*

Proof: Technically, this result means that for any derivation of e_1 there exists some derivation of e_2 such that the translations of those derivations satisfy the given condition. Let $e_1 = E\{\iota\}$ and $e_2 = E\{e\}$ for some ι and e in Figure 2. By Lemma D.3 $[\![e_1]\!]_{\mathrm{exp}} = [\![E]\!]_{\mathrm{exp}}\{[\![\iota]\!]_{\mathrm{exp}}\}$ and $[\![e_2]\!]_{\mathrm{exp}} = [\![E]\!]_{\mathrm{exp}}\{[\![e]\!]_{\mathrm{exp}}\}$. It suffices to show that $[\![\iota]\!]_{\mathrm{exp}} \mapsto^{+} [\![e]\!]_{\mathrm{exp}}$. Throughout this proof, I will use the abbreviations at the bottom of Figure 7. Consider the cases for ι:

case $\iota = \mathsf{et}$: The result follows from $[\![\iota]\!]_{\mathrm{exp}} = \mathrm{stutter}([\![e]\!]_{\mathrm{exp}})$.

case $\iota = v1 + f{:}\sigma^{\phi}$: In this case $e = \mathsf{temp}[m_i = M_i; f_j{:}\sigma_j^{\phi_j}, f{:}\sigma^{\phi}]_{i\in I, j\in J}$. So:

$$\begin{array}{rcl} [\![\iota]\!]_{\mathrm{exp}} & = & \Lambda\alpha \leq [\![r_1]\!]_{\mathrm{full}}(\alpha).((\Lambda\beta \leq [\![r]\!]_{\mathrm{full}}(\beta).\langle m_i = [\![M_i]\!]_{\mathrm{mth}}(\beta)\rangle_{i\in I})[\alpha]) \\ & \mapsto & \Lambda\alpha \leq [\![r_1]\!]_{\mathrm{full}}(\alpha).\langle m_i = [\![M_i]\!]_{\mathrm{mth}}(\alpha)\rangle_{i\in I} \\ & = & [\![e]\!]_{\mathrm{exp}} \end{array}$$

Note that the (meth-sub) rule is used to get a derivation for e that uses exactly the same translation of M_i as was used in ι. Without the rule (meth-sub), the translation of e would use the type $\mathsf{objt}\ r_1$ to translate M_i, whereas the translation of ι would use the type $\mathsf{objt}\ r$, resulting in a different translation and the simulation result would not hold.

case $\iota = v1 \leftarrow\!\!\!+ [m_k = M'_k]_{k \in K}$**:** In this case $e = \mathsf{temp}[m_l = M''_l; f_j{:}\sigma_j^{\phi_j}]_{l \in (I, K-I), j \in J}$. So:

$$\begin{array}{rl}
\llbracket \iota \rrbracket_{\text{exp}} \;=\; & \Lambda\alpha \leq \llbracket r_2 \rrbracket_{\text{full}}(\alpha). \\
& \quad ((\Lambda\beta \leq \llbracket r \rrbracket_{\text{full}}(\beta).\langle m_i = \llbracket M_i \rrbracket_{\text{mth}}(\beta)\rangle_{i \in I})[\alpha] \\
& \qquad .m_k \leftarrow_{k \in I \cap K} \llbracket M'_k \rrbracket_{\text{mth}}(\alpha) \\
& \qquad +_{k \in K-I} m_k = \llbracket M'_k \rrbracket_{\text{mth}}(\alpha)) \\
\mapsto \; & \Lambda\alpha \leq \llbracket r_2 \rrbracket_{\text{full}}(\alpha). \\
& \quad (\langle m_i = \llbracket M_i \rrbracket_{\text{mth}}(\alpha)\rangle_{i \in I} \\
& \qquad .m_k \leftarrow_{k \in I \cap K} \llbracket M'_k \rrbracket_{\text{mth}}(\alpha) \\
& \qquad +_{k \in K-I} m_k = \llbracket M'_k \rrbracket_{\text{mth}}(\alpha)) \\
\mapsto^{|I \cap K|} \; & \Lambda\alpha \leq \llbracket r_2 \rrbracket_{\text{full}}(\alpha). \\
& \quad (\langle m_i = \llbracket M''_i \rrbracket_{\text{mth}}(\alpha)\rangle_{i \in I} +_{k \in K-I} m_k = \llbracket M'_k \rrbracket_{\text{mth}}(\alpha)) \\
\mapsto^{|K-I|} \; & \Lambda\alpha \leq \llbracket r_2 \rrbracket_{\text{full}}(\alpha).\langle m_i = \llbracket M''_i \rrbracket_{\text{mth}}(\alpha)\rangle_{i \in (I, K-I)} \\
= \; & \llbracket e \rrbracket_{\text{exp}}
\end{array}$$

Where $M''_i = M_i$ if $i \notin K$ and $M''_i = M'_i$ if $i \in K$. Note that, as in the previous case, the rule (meth-sub) is crucial.

case $\iota = \mathsf{new}\ v1[f_j = w_j]_{j \in J}$**:** In this case $e = v2$. Let $\tau = \mathsf{rec}\ \alpha..\llbracket r \rrbracket_{\text{full}}(\alpha)$, then:

$$\begin{array}{rl}
\llbracket \iota \rrbracket_{\text{exp}} \;=\; & \mathsf{let}\ x = \Lambda\alpha \leq \llbracket r \rrbracket_{\text{full}}(\alpha).\langle m_i = \llbracket M_i \rrbracket_{\text{mth}}(\alpha)\rangle_{i \in I}\ \mathsf{and}\ x_j =_{j \in J} \llbracket w_j \rrbracket_{\text{exp}}\ \mathsf{in} \\
& \mathsf{pack}\ \langle \mathsf{mt} = x[\tau], f_j = x_j\rangle_{j \in J}, \tau\ \mathsf{as}\ \llbracket \mathsf{objt}\ r \rrbracket_{\text{type}} \\
\mapsto \; & \mathsf{pack}\ \langle \mathsf{mt} = (\Lambda\alpha \leq \llbracket r \rrbracket_{\text{full}}(\alpha).\langle m_i = \llbracket M_i \rrbracket_{\text{mth}}(\alpha)\rangle_{i \in I})[\tau], \\
& \qquad f_j = \llbracket w_j \rrbracket_{\text{exp}}\rangle_{j \in J}, \tau\ \mathsf{as}\ \llbracket \mathsf{objt}\ r \rrbracket_{\text{type}} \\
\mapsto \; & \mathsf{pack}\ \langle \mathsf{mt} = \langle m_i = \llbracket M_i \rrbracket_{\text{mth}}(\tau)\rangle_{i \in I}, f_j = \llbracket w_j \rrbracket_{\text{exp}}\rangle_{j \in J}, \tau\ \mathsf{as}\ \llbracket \mathsf{objt}\ r \rrbracket_{\text{type}} \\
= \; & \llbracket v2 \rrbracket_{\text{exp}}
\end{array}$$

case $\iota = v2.m_k$**:** In this case $e = b_k\{x_k := v2\}$. The derivation of $\emptyset \vdash_{\mathcal{O}} e_1 : \tau$ must include a derivation of $\emptyset \vdash_{\mathcal{O}} v2 : \mathsf{objt}\ r$, which in turn must include a derivation of $\emptyset \vdash_{\mathcal{O}}^{\mathsf{M}} M_k : \sigma \rhd \sigma_k$ for some σ. Furthermore, $\vdash_{\mathcal{O}} \mathsf{objt}\ r \leq \sigma$, so σ must have the form $\mathsf{objt}\ r'$ for some r'. Let $\tau = \mathsf{rec}\ \alpha.\llbracket r \rrbracket_{\text{full}}(\alpha)$ and $v2' = \langle \mathsf{mt} = \langle m_i = \llbracket M_i \rrbracket_{\text{mth}}(\tau)\rangle_{i \in I}, f_j = \llbracket w_j \rrbracket_{\text{exp}}\rangle_{j \in J}$.

$$\begin{array}{rl}
\llbracket \iota \rrbracket_{\text{exp}} \;=\; & \mathsf{unpack}\ \alpha, x = \llbracket v2 \rrbracket_{\text{exp}}\ \mathsf{in}\ x.\mathsf{mt}.m_k\ x \\
\mapsto \; & v2'.\mathsf{mt}.m_k\ v2' \\
\mapsto \; & \langle m_i = \llbracket M_i \rrbracket_{\text{mth}}(\tau)\rangle_{i \in I}.m_k\ v2' \\
\mapsto \; & \llbracket M_k \rrbracket_{\text{mth}}(\tau)\ v2' \\
\mapsto \; & \mathsf{let}\ x_k = \mathsf{pack}\ v2', \tau\ \mathsf{as}\ \llbracket \mathsf{objt}\ r' \rrbracket_{\text{type}}\ \mathsf{in}\ \llbracket b_k \rrbracket_{\text{exp}} \\
\mapsto \; & \llbracket b_k \rrbracket_{\text{exp}}\{x_k := \mathsf{pack}\ v2', \tau\ \mathsf{as}\ \llbracket \mathsf{objt}\ r' \rrbracket_{\text{type}}\} \\
= \; & \llbracket b_k\{x_k := \llbracket v2 \rrbracket_{\text{exp}}\} \rrbracket_{\text{exp}} \\
= \; & \llbracket e \rrbracket_{\text{exp}}
\end{array}$$

The second to last step uses Lemma D.1 and it uses a translation for $v2$ based on the derivation $\emptyset \vdash_{\mathcal{O}} v2 : \mathsf{objt}\ r'$, which is obtained from $\emptyset \vdash_{\mathcal{O}} v2 : \mathsf{objt}\ r$ and $\vdash_{\mathcal{O}} \mathsf{objt}\ r \leq \mathsf{objt}\ r'$ by (subsume).

case $\iota = v2.f_k$**:** In this case $e = w_k$. Let $\tau = \mathsf{rec}\ \alpha.\llbracket r \rrbracket_{\text{full}}(\alpha)$. Then:

$$\begin{array}{rl}
\llbracket \iota \rrbracket_{\text{exp}} \;=\; & \mathsf{unpack}\ \alpha, x = \llbracket v2 \rrbracket_{\text{exp}}\ \mathsf{in}\ x.f_k \\
\mapsto \; & \langle \mathsf{mt} = \langle m_i = \llbracket M_i \rrbracket_{\text{mth}}(\tau)\rangle_{i \in I}, f_j = \llbracket w_j \rrbracket_{\text{exp}}\rangle_{j \in J}.f_k \\
\mapsto \; & \llbracket w_k \rrbracket_{\text{exp}} \\
= \; & \llbracket e \rrbracket_{\text{exp}}
\end{array}$$

case $\iota = v2.f_k := v$**:** In this case $e = \mathsf{obj}[m_i = M_i; f_j = w'_j]_{i\in I, j\in J}$ where $w'_j = w_j$ if $j \neq k$ and $w'_k = v$. By inspection of the typing rules, the type for $v2.f_k := v$ must be $\mathsf{objt}\ r'$ for some r' such that $\vdash_{\mathcal{O}} \mathsf{objt}\ r \leq \mathsf{objt}\ r'$. Let $\tau = \mathsf{rec}\ \alpha.[\![r]\!]_{\mathrm{full}}(\alpha)$ and $meths = \langle m_i = [\![M_i]\!]_{\mathrm{mth}}(\alpha, r)\rangle_{i\in I}$. Then:

$$\begin{array}{rcl}
[\![\iota]\!]_{\mathrm{exp}} & = & \mathsf{let}\ x_1 = [\![v2]\!]_{\mathrm{exp}}\ \mathsf{and}\ x_2 = [\![v]\!]_{\mathrm{exp}}\ \mathsf{in} \\
 & & \mathsf{unpack}\ \alpha, x_1 = x_1\ \mathsf{in} \\
 & & \mathsf{pack}\ x_1.f_k := x_2, \alpha\ \mathsf{as}\ [\![\mathsf{objt}\ r']\!]_{\mathrm{type}} \\
 & \mapsto & \mathsf{unpack}\ \alpha, x_1 = [\![v2]\!]_{\mathrm{exp}}\ \mathsf{in} \\
 & & \mathsf{pack}\ x_1.f_k := [\![v]\!]_{\mathrm{exp}}, \alpha\ \mathsf{as}\ [\![\mathsf{objt}\ r']\!]_{\mathrm{type}} \\
 & \mapsto & \mathsf{pack}\ \langle \mathsf{mt} = meths, f_j = [\![w_j]\!]_{\mathrm{exp}}\rangle.f_k := [\![v]\!]_{\mathrm{exp}}, \tau\ \mathsf{as}\ [\![\mathsf{objt}\ r']\!]_{\mathrm{type}} \\
 & \mapsto & \mathsf{pack}\ \langle \mathsf{mt} = meths, f_j = [\![w'_j]\!]_{\mathrm{exp}}\rangle, \tau\ \mathsf{as}\ [\![\mathsf{objt}\ r']\!]_{\mathrm{type}} \\
 & = & [\![e]\!]_{\mathrm{exp}}
\end{array}$$

Note that the translation of e is using the typing $\Gamma \vdash_{\mathcal{O}} e : \mathsf{objt}\ r'$.

□

Theorem D.6 *If $\emptyset \vdash_{\mathcal{O}} e : \tau$ then $e \not\mapsto$ if and only if $[\![e]\!]_{\mathrm{exp}} \not\mapsto$.*

Proof: Follows from the soundness of $\mathcal{O}$, Lemma D.4, and Lemma D.5. □

Theorem D.7 *If $\emptyset \vdash_{\mathcal{O}} e : \tau$ then: $e \mapsto^* v$ if and only if $[\![e]\!]_{\mathrm{exp}} \mapsto^* [\![v]\!]_{\mathrm{exp}}$, and $e \mapsto \cdots$ if and only if $[\![e]\!]_{\mathrm{exp}} \mapsto \cdots$.*

Proof: By induction, Theorem D.5, and Theorem D.6. □

E Covariant Self Types

This appendix describes how to extend the target language and the encoding to handle covariant self types in the object language.

E.1 O with Covariant Self Types

The extended syntax for $\mathcal{O}$ is the following with the original defintions for r, e, and E.

$$\begin{array}{lcl}
\tau & ::= & \alpha \mid \mathsf{tempt}\ r \mid \mathsf{objt}\ r \\
s & ::= & \alpha.\tau \\
M & ::= & \alpha, x.e : \tau \\
v & ::= & \mathsf{temp}[m_i = M_i; f_j{:}\sigma_j]_{i\in I, j\in J} \mid \mathsf{obj}[\tau; m_i = M_i; f_j = v_j]_{i\in I, j\in J}
\end{array}$$

Subject to the restriction that α must occur syntactically positive in τ for $\alpha.\tau$ to be a signature. A type variable α appears syntactically positive in τ if τ is a type variable, τ is $\mathsf{tempt}\ r$ and $\alpha \notin \mathrm{ftv}(\tau)$, or $\tau = \mathsf{objt}[m_i{:}\alpha_i.\tau_i; f_j{:}\sigma_j]_{i\in I,j\in J}$, $\alpha \notin \mathrm{ftv}(\sigma_j)$, and $\alpha = \alpha_i$ or α appears syntactically positive in τ_i.

The operational semantics changes in two places. The rule for instantiation becomes:

$$\mathsf{new}\ \mathsf{temp}[m_i = M_i; f_j{:}\sigma_j]_{i\in I,j\in J}[f_j = w_j]_{j\in J} \mapsto \mathsf{obj}[\tau; m_i = M_i; f_j = w_j]_{i\in I,j\in J}$$

where $M_i = \alpha_i, x_i.e_i{:}\tau_i$ and $\tau = \mathsf{objt}[m_i{:}\alpha_i.\tau_i; f_j{:}\sigma_j]_{i\in I,j\in J}$. The rule for method invocation becomes:

$$v.m_k \mapsto e\{\alpha, x := \tau, v\}$$

where $v = \mathsf{obj}[\tau; m_i = M_i; f_j = v_j]_{i\in I,j\in J}$ and $M_k = \alpha, x.e{:}\sigma$.

To type the extended language, the judgements must be extended to include typing contexts and subtyping bounds. A typing context Δ is a sequence of type variables, and a subtying bounds B is a list of type variables and their bounds $\alpha_1 \le \tau_1$. A type if well formed $\Delta \vdash_{\mathcal{O}} \tau$ exactly when $\mathrm{ftv}(\tau) \subseteq \Delta$. Subtyping is giving by these rules:

$$\frac{}{\Delta; B \vdash_{\mathcal{O}} \alpha \le \alpha}\ (\alpha \in \Delta) \qquad \frac{\Delta; B \vdash_{\mathcal{O}} \tau_1 \le \tau_2}{\Delta; B \vdash_{\mathcal{O}} \alpha \le \tau_2}\ (\alpha \in \Delta; \alpha \le \tau_1 \in B)$$

$$\frac{\Delta \vdash_{\mathcal{O}} \mathsf{tempt}\ r}{\Delta; B \vdash_{\mathcal{O}} \mathsf{tempt}\ r \le \mathsf{tempt}\ r}$$

$$\frac{i \in I_2 : \Delta, \alpha; B \vdash_{\mathcal{O}} \tau_i \le \tau_i' \quad i \in I_1 : \Delta, \alpha \vdash_{\mathcal{O}} \tau_i}{\Delta; B \vdash_{\mathcal{O}} \mathsf{objt}[m_i{:}\alpha.\tau_i; f_j{:}\sigma_k]_{i\in I_1,j\in J_1} \le \mathsf{objt}[m_i{:}\alpha.\tau_i'; f_j{:}\sigma_k]_{i\in I_2,j\in J_2}}$$

where I_2 is a prefix of I_1 and J_2 is a prefix of J_1. Reflexivity and transitivity are derivably.

The typing rules for expressions are those in Figure 3 except for the rules for method invocation and method bodies, with Γ replaced by $\Delta; B; \Gamma$, with subtyping judgements in the context $\Delta; B$, and with $\vdash_{\mathcal{O}} s_i' \le s_i$ in the rule for method add/override replaced with $\Delta, \alpha; B \vdash_{\mathcal{O}} \tau_i' \le \tau_i$ where $s_i' = \alpha.\tau_i'$ and $s_i = \alpha.\tau_i$. Define the self-type expose operation as follows:

$$\begin{array}{lcl}
\exp(\alpha, \mathsf{objt}\ r) & = & \mathsf{objt}\ \exp(\alpha, r) \\
\exp(\alpha, [m_i{:}s_i; f_j{:}\sigma_j]_{i\in I,j\in J}) & = & [m_i{:}\exp(\alpha, s_i); f_j{:}\sigma_j]_{i\in I,j\in J} \\
\exp(\alpha, \alpha.\tau) & = & \beta.\tau \qquad \text{where } \beta \notin \mathrm{ftv}(\tau)
\end{array}$$

Then the rule for method bodies is:

$$\frac{\Delta, \alpha; B, \alpha \le \exp(\alpha, \tau); \Gamma, x{:}\alpha \vdash_{\mathcal{O}} e : \sigma}{\Delta; B; \Gamma \vdash^{\mathsf{M}}_{\mathcal{O}} \alpha, x.e{:}\sigma : \tau \rhd \sigma}$$

The rule for method invocation is:

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

The extended language is sound with respect to the operational semantics. To prove this, it helps to make the subtyping rule for objects stronger by replacing the first hypothesis with $\Delta, \alpha; B, \alpha \le \mathsf{objt}[m_i{:}\alpha.\tau_i; f_j{:}\sigma_k]_{i\in I_1,j\in J_1} \vdash_{\mathcal{O}} \tau_i \le \tau_i'$. This stronger rule makes the system undecidable and the translation does not preserve it, but it enables the usual preservation and progress style of type-soundness proof. In addition to the lemmas in Appendix A, which are easily proven for the extended lanugage, a type substitution lemma is needed.

E.2 Translation

The main idea of the translation is to treat the source-level self types just as the self type is already delt with at the target level. This is most directly expressed in the translation of signatures:

$$[\![\alpha.\sigma]\!]_{\mathrm{sig}}(\tau) = (\alpha \to [\![\sigma]\!]_{\mathrm{type}})\{\alpha := \tau\}$$

There are two further details. First, method invocation needs to be modified to deal with free self variables. For example, if $e : \mathsf{objt}[\mathtt{bump}{:}\alpha.\alpha;]$ then $e.\mathtt{bump}$ should have type $\mathsf{objt}[\mathtt{bump}{:}\alpha.\alpha;]$. The translation in Section 3.3 translates it to $\mathsf{unpack}\ \alpha, x = [\![e]\!]_{\mathrm{exp}}\ \mathsf{in}\ x.\mathsf{mt}.\mathtt{bump}\ x$. Using the typing so far, the body of this unpack has type α, which is not allowed by the unpack rule. However, the body can be repacked into type $[\![\mathsf{objt}[\mathtt{bump}{:}\alpha.\alpha;]]\!]_{\mathrm{type}}$. In general, the method invocation $e.m$ will have type $\sigma\{\alpha := \tau\}$ where $e : \tau$ and m's signature in τ is $\alpha.\sigma$. In the translation the body of the unpack will have type $[\![\sigma]\!]_{\mathrm{type}}$ where α may appear free but only covariantly. Replacing these covariant occurances of α by $[\![\tau]\!]_{\mathrm{type}}$ by a pack like operation is safe, but the machinery given so far does not allow it. To address this shortcoming, a deeper form of pack is added to the language. This deeper pack is written $\mathsf{pack}^{+}\ v, \tau_1[\alpha = \tau_2]\ \mathsf{as\ self}\ \beta.\sigma$ and changes a value v of type $\tau_1\{\alpha := \tau_2\}$ to type $\tau_1\{\alpha := \mathsf{self}\ \beta.\sigma\}$ so long as α is only positive in τ_1 and $\tau_2 \leq \sigma\{\beta := \tau_2\}$. The old pack can be considered syntactic sugar: $\mathsf{pack}\ v, \tau\ \mathsf{as\ self}\ \alpha.\sigma = \mathsf{pack}^{+}\ v, \beta[\beta = \tau]\ \mathsf{as\ self}\ \alpha.\sigma$. Using the deep pack, the translation of method invocation is $\mathsf{unpack}\ \alpha, x = [\![e]\!]_{\mathrm{exp}}\ \mathsf{in\ pack}^{+}\ x.\mathsf{mt}.m\ x, [\![\sigma]\!]_{\mathrm{type}}[\beta = \alpha]\ \mathsf{as}\ [\![\tau]\!]_{\mathrm{type}}$, where $e : \tau$ and m's signature in τ is $\beta.\sigma$.

Second, the translation of the context used to check method bodies does not agree with the context in which translated methods are checked. For example, in $\mathsf{et} \leftarrow\!\!\!+[\mathtt{bump} = \alpha, x.x.\mathtt{bump}{:}\alpha]$ the method body $x.\mathtt{bump}$ is checked in the context $\alpha; \alpha \leq \mathsf{objt}[\mathtt{bump}{:}\beta.\alpha;]; x{:}\alpha$. Translated this context becomes $\alpha; \alpha \leq \mathsf{self}\ \beta.\langle \mathsf{mt}{:}\langle \mathtt{bump}{:}\beta \to \alpha^{+}\rangle^{\to +}\rangle; x{:}\alpha$. However, the translated method body is checked in the environment $\alpha; \alpha \leq \langle \mathsf{mt}{:}\langle \mathtt{bump}{:}\alpha \to \alpha^{+}\rangle^{\to +}\rangle; x{:}\alpha$. Methods have to be checked in this environment because they must be polymorphic in the type of self to work properly in subclasses. The solution is to witness the subtyping in the source calculus of α to $\exp(\alpha, \tau)$ with a pack coercion in the target calculus. To make this precise, I will add an explicit coercion to $\mathcal{O}$, then show how to translate $\mathcal{O}$ to itself replacing implicit subtyping by the new explicit coercion, and finally show how to translate this extended $\mathcal{O}$.

The explicit coercion is written $\mathsf{tobnd}(e, \tau[\alpha = \beta \leq \sigma])$ where $e : \tau\{\alpha := \beta\}$, α appears only covariantly in τ, and σ is β's bound. The goal is eliminate the use in a subsumption rule of the bound introduced by the method body checking rule and replace it with the explicit coercion. Consider any expression and a typing derivation for it. The expression is transformed by considering each use of the method body checking rule on a method body $\alpha, x.e{:}\tau$. For each use of the subsumption rule within the typing checking of e, if the subsumption rule is used on an expression e' with hypotheses $\Delta; B; \Gamma \vdash_{\mathcal{O}} e_1 : \tau_2$ and $\Delta; B \vdash_{\mathcal{O}} \tau_1 \leq \tau_2$, replace e' by $f(e')$ where $\Delta; B \vdash_{\mathcal{O}} \tau_1 \leq \tau_2 \mapsto^{\alpha,\gamma.\gamma} f$ as determined by the following rules:

$$\frac{}{\Delta; B \vdash_{\mathcal{O}} \beta \leq \beta \mapsto^{\alpha,\gamma.\tau} \lambda x.x}\ (\beta \in \Delta)$$

$$\frac{\Delta; B \vdash_{\mathcal{O}} \tau_1 \leq \tau_2 \mapsto^{\alpha,\gamma.\tau} f}{\Delta; B \vdash_{\mathcal{O}} \beta \leq \tau_2 \mapsto^{\alpha,\gamma.\tau} f}\ (\beta \in \Delta; \beta \leq \tau_1 \in B; \alpha \neq \beta)$$

$$\frac{\Delta; B \vdash_{\mathcal{O}} \tau_1 \leq \tau_2 \mapsto^{\alpha,\gamma.\tau} f}{\Delta; B \vdash_{\mathcal{O}} \alpha \leq \tau_2 \mapsto^{\alpha,\gamma.\tau} \lambda x.f(\mathsf{tobnd}(x, \tau[\gamma = \alpha \leq \tau_1]))}\ (\alpha \in \Delta; \alpha \leq \tau_1 \in B)$$

$$\frac{\Delta \vdash_{\mathcal{O}} \mathsf{tempt}\ r}{\Delta; B \vdash_{\mathcal{O}} \mathsf{tempt}\ r \leq \mathsf{tempt}\ r \mapsto^{\alpha,\gamma.\tau} \lambda x.x}$$

$$\frac{i \in I_2 : \Delta, \beta; B \vdash_{\mathcal{O}} \tau_i \leq \tau_i' \mapsto^{\alpha,\gamma.\tau^i} f_i \quad i \in I_1 : \Delta, \alpha \vdash_{\mathcal{O}} \tau_i}{\Delta; B \vdash_{\mathcal{O}} \mathsf{objt}[m_i{:}\beta.\tau_i; f_j{:}\sigma_j]_{i \in I_1, j \in J_1} \leq \mathsf{objt}[m_i{:}\beta.\tau_i'; f_j{:}\sigma_j]_{i \in I_2, j \in J_2} \mapsto^{\alpha,\gamma.\tau} \circ_{i \in I_2} f_i}$$

where $\tau^k = \tau\{\gamma := \mathsf{objt}[m_i{:}\beta.\tau_i^k; f_j{:}\sigma_j]_{i \in I_1, j \in J_1}\}$, $\tau_k^k = \gamma$, $\tau_i^k = \tau_i$ if $k \neq i$, $\beta \notin \{\alpha, \gamma\}$, I_2 is a prefix of I_1, and J_2 is a prefix of J_1.

Given an expression e in the extended $\mathcal{O}$ such that the subtyping rule for bounds is not used for any bound introduced by the method body checking rule, the translation is as follows:

$$\begin{array}{lcl}
\llbracket \alpha.\sigma \rrbracket_{\mathrm{sig}}(\tau) & = & (\alpha \to \llbracket \sigma \rrbracket_{\mathrm{type}})\{\alpha := \tau\} \\
\llbracket e.m \rrbracket_{\mathrm{exp}} & = & \mathsf{unpack}\ \alpha, x = \llbracket e \rrbracket_{\mathrm{exp}}\ \mathsf{in} \\
 & & \mathsf{pack}^{+}\ x.\mathsf{mt}.m\ x, \llbracket \sigma \rrbracket_{\mathrm{type}}[\beta = \alpha]\ \mathsf{as}\ \llbracket \mathsf{objt}\ r \rrbracket_{\mathrm{type}} \\
 & & \text{where } e \text{ has type } \mathsf{objt}\ r\text{, } m\text{'s signature in } r \text{ is } \beta.\sigma; \\
 & & \qquad \alpha \text{ and } x \text{ are fresh} \\
\llbracket \mathsf{tobnd}(e, \tau[\alpha = \beta \leq \mathsf{objt}\ r]) \rrbracket_{\mathrm{exp}} & = & \mathsf{pack}^{+}\ \llbracket e \rrbracket_{\mathrm{exp}}, \llbracket \tau \rrbracket_{\mathrm{type}}[\alpha = \beta]\ \mathsf{as\ self}\ \gamma.\llbracket r \rrbracket_{\mathrm{full}}(\gamma) \\
\llbracket \alpha, x.e{:}\sigma \rrbracket_{\mathrm{mth}}(\tau) & = & (\lambda x{:}\alpha.\llbracket e \rrbracket_{\mathrm{exp}})\{\alpha := \tau\}
\end{array}$$

The proofs of type preservation and operational correctness could probably be extended to this translation.

E.3 Structural Method Invocation

A structural rule for method invocation would look like:

$$\frac{\Delta; B; \Gamma \vdash_{\mathcal{O}} e : \tau \quad \Delta; B \vdash_{\mathcal{O}} \tau \leq \mathsf{objt}[m_i{:}\alpha_i.\tau_i; f_j{:}\sigma_j]_{i \in I, j \in J}}{\Delta; B; \Gamma \vdash_{\mathcal{O}} e.m_k : \tau_k\{\alpha_k := \tau\}} \ (k \in I)$$

To translate this variant, a structural unpack rule is needed. This best expressed if unpack also repacks its body:

$$\mathsf{unpack}\ \alpha, x = \mathsf{pack}^{+}\ v, \beta[\beta = \tau]\ \mathsf{as\ self}\ \alpha.\sigma\ \mathsf{in}\ e \mapsto \mathsf{pack}^{+}\ e\{\alpha, x := \tau, v\}, \tau'[\beta = \tau]\ \mathsf{as\ self}\ \alpha.\sigma$$

where e has type τ'. (To formalise this properly, we would need to annotate unpack with e's type.) The typing rule now becomes:

$$\frac{\Delta; B; \Gamma \vdash_{\mathcal{F}} e_1 : \tau_1 \quad \Delta; B \vdash_{\mathcal{F}} \alpha_1 \leq \mathsf{self}\ \alpha.\tau \quad \Delta, \alpha; B, \alpha \leq \tau; \Gamma, x{:}\alpha \vdash_{\mathcal{F}} e_2 : \tau_2 \quad \vdash_{\mathcal{F}} \tau_2\{\alpha^{+}\}}{\Delta; B; \Gamma \vdash_{\mathcal{F}} \mathsf{unpack}\ \alpha, x = e_1\ \mathsf{in}\ e_2 : \tau_2\{\alpha := \tau_1\}}$$

The new translation now does not do the repack in method translation, as the unpack does it:

$$\llbracket e.m \rrbracket_{\mathrm{exp}} = \mathsf{unpack}\ \alpha, x, \llbracket e \rrbracket_{\mathrm{exp}} = x.\mathsf{mt}.m\ x\ \mathsf{in}$$

The rest of the translation is the same.

E.4 Target Language Details

I will give a few more details of the deep pack coercions. The pack operation is replaced with the deep one:

$$\begin{array}{lcl} e & ::= & \cdots \mid \mathsf{pack}^{\phi}\ e, \tau_1[\beta = \tau_2]\ \mathsf{as\ self}\ \alpha.\tau \\ v & ::= & \cdots \mid \mathsf{pack}^{+}\ v, \beta[\beta = \tau]\ \mathsf{as\ self}\ \alpha.\sigma \\ E & ::= & \cdots \mid \mathsf{pack}^{\phi}\ E, \tau_1[\beta = \tau_2]\ \mathsf{as\ self}\ \alpha.\tau \end{array}$$

where $\phi \in \{+, -\}$. The pack^- form is like the opposite of pack^+ and is needed to deal with contravariant positions such as function arguments.

The old pack is treated as syntactic sugar for a specific deep pack: $\mathsf{pack}\ v, \tau\ \mathsf{as\ self}\ \alpha.\sigma = \mathsf{pack}^{+}\ v, \beta[\beta = \tau]\ \mathsf{as\ self}\ \alpha.\sigma$. Using this sugar, the operational semantics of Figure 2 gives unpack the semantics (I am not considering the structural pack rule from the previous subsection):

$$\mathsf{unpack}\ \alpha, x = \mathsf{pack}^{+}\ v, \beta[\beta = \tau]\ \mathsf{as\ self}\ \alpha.\sigma\ \mathsf{in}\ e \mapsto e\{\alpha, x := \tau, v\}$$

In addition there are operational rules for the other nonvalue pack forms. These rules just push the pack coercions deeper into the term structure:

$$\mathsf{pack}^{\phi}\ v, \beta[\alpha = \sigma_1]\ \mathsf{as}\ \sigma_2 \mapsto v \qquad \alpha \neq \beta$$

$$\begin{array}{l} \mathsf{pack}^{+}\ \lambda x{:}\tau'.e, (\tau_1 \rightarrow \tau_2)[\alpha = \sigma_1]\ \mathsf{as}\ \sigma_2 \mapsto \\ \qquad \lambda x{:}\tau_1\{\alpha := \sigma_2\}.\mathsf{pack}^{+}\ e\{x := \mathsf{pack}^{-}\ x, \tau_1[\alpha = \sigma_1]\ \mathsf{as}\ \sigma_2\}, \tau_2[\alpha = \sigma_1]\ \mathsf{as}\ \sigma_2 \end{array}$$

$$\begin{array}{l} \mathsf{pack}^{-}\ \lambda x{:}\tau'.e, (\tau_1 \rightarrow \tau_2)[\alpha = \sigma_1]\ \mathsf{as}\ \sigma_2 \mapsto \\ \qquad \lambda x{:}\tau_1\{\alpha := \sigma_1\}.\mathsf{pack}^{-}\ e\{x := \mathsf{pack}^{+}\ x, \tau_1[\alpha = \sigma_1]\ \mathsf{as}\ \sigma_2\}, \tau_2[\alpha = \sigma_1]\ \mathsf{as}\ \sigma_2 \end{array}$$

$$\mathsf{pack}^{\phi}\ \langle \ell_i = v_i \rangle_{i \in I_1}, (\langle \ell_i{:}\tau_i^{\phi_i} \rangle_{i \in I_2})[\alpha = \sigma_1]\ \mathsf{as}\ \sigma_2 \mapsto \langle \ell_i = \mathit{dopack}^{\phi}(v_i, \tau_i^{\phi_i}, \alpha, \sigma_1, \sigma_2, i \in I_2) \rangle_{\ell \in I_1}$$

$$\mathit{dopack}^{\phi}(v, \tau^{\phi'}, \alpha, \sigma_1, \sigma_2, b) = \begin{cases} v & b \Rightarrow \phi' = \circ \\ \mathsf{pack}^{\phi.\phi'}\ v, \tau[\alpha = \sigma_1]\ \mathsf{as}\ \sigma_2 & b \wedge \phi' \neq \circ \end{cases}$$

$$\mathsf{pack}^{\phi}\ \Lambda\alpha \leq \tau.v, (\forall \alpha \leq \tau_1.\tau_2)[\alpha = \sigma_1]\ \mathsf{as}\ \sigma_2 \mapsto \Lambda\alpha \leq \tau.\mathsf{pack}^{\phi}\ v, \tau_2[\alpha = \sigma_1]\ \mathsf{as}\ \sigma_2$$

$$\mathsf{pack}^{\phi}\ v, (\mathsf{rec}\ \beta.\tau)[\alpha = \sigma_1]\ \mathsf{as}\ \sigma_2 \mapsto v \qquad \text{not } \beta \downarrow \tau$$

$$\mathsf{pack}^{\phi}\ v, (\mathsf{rec}\ \beta.\tau)[\alpha = \sigma_1]\ \mathsf{as}\ \sigma_2 \mapsto \mathsf{pack}^{\phi}\ v, (\tau\{\beta := \mathsf{rec}\ \beta.\tau\})[\alpha = \sigma_1]\ \mathsf{as}\ \sigma_2 \qquad \beta \downarrow \tau$$

The rule for pushing a pack through a pack is a little complicated. In the encoding presented in this paper, the actual self type is always $\mathsf{rec}\ \alpha.\tau$ or some type variable that came from an unpack. Therefore, at run time, the only self types in a pack form being evaluated are recursive types corresponding to the self types. Consider $\mathsf{pack}^{+}\ \mathsf{pack}^{+}\ v, \alpha[\alpha = \mathsf{rec}\ \gamma.\tau_1]\ \mathsf{as\ self}\ \gamma.\tau_1, (\mathsf{self}\ \beta.\sigma_2)[\alpha = \mathsf{rec}\ \gamma.\tau_2]\ \mathsf{as\ self}\ \gamma.\tau_2$. The typing rules ensure that $\mathsf{self}\ \gamma.\tau_1 \leq (\mathsf{self}\ \beta.\sigma_2)\{\alpha := \mathsf{rec}\ \gamma.\tau_2\}$. In fact, the encodings presented in this paper are such that $\mathsf{self}\ \gamma.\tau_1 = (\mathsf{self}\ \gamma.\sigma_1)\{\alpha := \mathsf{rec}\ \gamma.\tau_2\}$ and $\mathsf{self}\ \gamma.\sigma_1 \leq \mathsf{self}\ \gamma.\sigma_2$. Therefore the following rule suffices for our purposes:

$$\begin{array}{l} \mathsf{pack}^{+}\ \mathsf{pack}^{+}\ v, \alpha[\alpha = \mathsf{rec}\ \gamma.\tau_1]\ \mathsf{as\ self}\ \gamma.\tau_1, (\mathsf{self}\ \beta.\sigma_2)[\alpha = \mathsf{rec}\ \gamma.\tau_2]\ \mathsf{as\ self}\ \gamma.\tau_2 \\ \qquad\qquad\qquad\qquad\qquad\qquad \mapsto \\ \qquad\qquad\qquad \mathsf{pack}^{+} \\ \qquad\qquad\qquad\quad \mathsf{pack}^{+}\ v, (\mathsf{rec}\ \gamma.\sigma_1)[\alpha = \mathsf{rec}\ \gamma.\tau_2]\ \mathsf{as\ self}\ \gamma.\tau_2, \\ \qquad\qquad\qquad\quad \alpha[\alpha = (\mathsf{rec}\ \gamma.\sigma_1)\{\alpha := \mathsf{self}\ \gamma.\tau_2\}]\ \mathsf{as} \\ \qquad\qquad\qquad\quad (\mathsf{self}\ \gamma.\sigma_1)\{\alpha := \mathsf{self}\ \gamma.\tau_2\} \end{array}$$

where $\alpha \neq \gamma$, $\Delta_E, \alpha; B_E \vdash_{\mathcal{F}} \sigma_1 \leq \sigma_2$, and $\sigma_1\{\alpha := \mathsf{rec}\ \gamma.\tau_2\} = \tau_1$. Similarly:

$$\begin{array}{l}
\mathsf{pack}^{-}\ \mathsf{pack}^{+}\ v, \alpha[\alpha = \mathsf{rec}\ \gamma.\tau_1]\ \mathsf{as\ self}\ \gamma.\tau_1, (\mathsf{self}\ \beta.\sigma_2)[\alpha = \mathsf{rec}\ \gamma.\tau_2]\ \mathsf{as\ self}\ \gamma.\tau_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mapsto \\
\qquad\qquad\qquad \mathsf{pack}^{+} \\
\qquad\qquad\qquad\quad \mathsf{pack}^{-}\ v, (\mathsf{rec}\ \gamma.\sigma_1)[\alpha = \mathsf{rec}\ \gamma.\tau_2]\ \mathsf{as\ self}\ \gamma.\tau_2, \\
\qquad\qquad\qquad\quad \alpha[\alpha = (\mathsf{rec}\ \gamma.\sigma_1)\{\alpha := \mathsf{rec}\ \gamma.\tau_2\}]\ \mathsf{as} \\
\qquad\qquad\qquad\quad (\mathsf{self}\ \gamma.\sigma_1)\{\alpha := \mathsf{rec}\ \gamma.\tau_2\}
\end{array}$$

where $\alpha \neq \gamma$, $\Delta_E, \alpha; B_E \vdash_{\mathcal{F}} \sigma_1 \leq \sigma_2$, and $\sigma_1\{\alpha := \mathsf{self}\ \gamma.\tau_2\} = \tau_1$.

The typing rules for deep packs are straighforward:

$$\frac{\Delta; B; \Gamma \vdash_{\mathcal{F}} e : \tau_1\{\alpha := \tau_2\} \quad \Delta; B \vdash_{\mathcal{F}} \tau_2 \leq \sigma\{\alpha := \tau_2\}}{\Delta; B; \Gamma \vdash_{\mathcal{F}} \mathsf{pack}^{+}\ e, \tau_1[\alpha = \tau_2]\ \mathsf{as\ self}\ \alpha.\sigma : \tau_1\{\alpha := \mathsf{self}\ \alpha.\sigma\}}\ (\tau_1\{\alpha^{+}\})$$

$$\frac{\Delta; B; \Gamma \vdash_{\mathcal{F}} e : \tau_1\{\alpha := \mathsf{self}\ \alpha.\sigma\} \quad \Delta; B \vdash_{\mathcal{F}} \tau_2 \leq \sigma\{\alpha := \tau_2\}}{\Delta; B; \Gamma \vdash_{\mathcal{F}} \mathsf{pack}^{-}\ e, \tau_1[\alpha = \tau_2]\ \mathsf{as\ self}\ \alpha.\sigma : \tau_1\{\alpha := \tau_2\}}\ (\tau_1\{\alpha^{-}\})$$

where $\tau\{\alpha^{\phi}\}$ means that α appears only positively, if $\phi = +$, or negatively, if $\phi = -$, in τ. Its definition is standard and omitted. It is straightfoward to extend the proof of soundness of the target language to these deep packs. Essentially, Decomposition and Canonical Forms need to be reformulated, but otherwise all the lemmas up to and including Canonical Forms still go through. Type Preservation has a bunch of extra cases for all the pack rules, but these are all just type chasing. Progress also has an extra case, but this follows by noting that the rules partition the possible forms of the type τ and ϕ in $\mathsf{pack}^{\phi}\ v, \tau[\alpha = \sigma_1]\ \mathsf{as}\ \sigma_2$ and that Canonical Forms implies that v has an appropriate form for the form of τ. I omit the many details. The proof could be extended to handle the structural unpack operation as well.

F Closure Conversion

This section details a closure-passing-style closure-conversion translation based on my previous object closure conversion [Gle99a] and the encoding in this paper.

The source language is a typed lambda calculus with records:

$$\begin{array}{lcl}
\tau & ::= & \tau_1 \to \tau_2 \mid \langle \ell_i{:}\tau_i \rangle_{i \in I} \\
e & ::= & x \mid \mathsf{fix}\ f(x{:}\tau_1){:}\tau_2.e \mid e_1\ e_2 \mid \langle \ell_i = e_i \rangle_{i \in I} \mid e.\ell
\end{array}$$

The idea is to treat functions as single-method objects, closure convert the resulting object language, and then encode these into records and functions (for single-method objects the method table might as well be collapsed into the object). The combined translation is:

$$
\begin{array}{lll}
\llbracket \tau_1 \to \tau_2 \rrbracket_{\mathrm{type}} & = & \mathsf{self}\ \alpha.\langle \mathsf{apply} = ((\alpha, \llbracket \tau_1 \rrbracket_{\mathrm{type}}) \to \llbracket \tau_2 \rrbracket_{\mathrm{type}})^{+} \rangle^{\to} \\
\llbracket \langle \ell_i{:}\tau_i \rangle_{i \in I} \rrbracket_{\mathrm{type}} & = & \langle \ell_i{:}\llbracket \tau_i \rrbracket^{+}_{\mathrm{type}} \rangle^{\to} \\
 & & \\
\llbracket x \rrbracket_{\mathrm{exp}} & = & x \\
\llbracket \mathsf{fix}\ f(x{:}\tau_1){:}\tau_2.e \rrbracket_{\mathrm{exp}} & = & \mathsf{pack}\ \langle \mathsf{apply} = \lambda(f'{:}\tau, x{:}\llbracket \tau_1 \rrbracket_{\mathrm{type}}).e', g_i = y_i \rangle_{1 \le i \le n}, \tau\ \mathsf{as}\ \llbracket \tau_1 \to \tau_2 \rrbracket_{\mathrm{type}} \\
\quad \text{where} \quad \{y_i{:}\sigma_i\}_{1 \le i \le n} & = & \mathrm{fv}(e) - \{f, x\} \\
\qquad\qquad\quad e' & = & \mathsf{let}\ f = \mathsf{pack}\ f', \tau\ \mathsf{as}\ \llbracket \tau_1 \to \tau_2 \rrbracket_{\mathrm{type}}\ \mathsf{in}\ (\llbracket e \rrbracket_{\mathrm{exp}} \{y_i := f'.g_i\}_{1 \le i \le n}) \\
\qquad\qquad\quad \tau & = & \mathsf{rec}\ \alpha.\langle \mathsf{apply}{:}((\alpha, \llbracket \tau_1 \rrbracket_{\mathrm{type}}) \to \llbracket \tau_2 \rrbracket_{\mathrm{type}})^{+}, g_i{:}\llbracket \sigma_i \rrbracket^{+}_{\mathrm{type}} \rangle^{\to}_{1 \le i \le n} \\
\qquad\qquad\quad g_1, \ldots, g_n \text{ are fresh} & & \\
\llbracket e_1\ e_2 \rrbracket_{\mathrm{exp}} & = & \mathsf{unpack}\ \alpha, x = \llbracket e_1 \rrbracket_{\mathrm{exp}}\ \mathsf{in}\ x.\mathsf{apply}(x, \llbracket e_2 \rrbracket_{\mathrm{exp}}) \\
\llbracket \langle \ell_i = e_i \rangle_{i \in I} \rrbracket_{\mathrm{exp}} & = & \langle \ell_i = \llbracket e_i \rrbracket_{\mathrm{exp}} \rangle_{i \in I} \\
\llbracket e.\ell \rrbracket_{\mathrm{exp}} & = & \llbracket e \rrbracket_{\mathrm{exp}}.\ell
\end{array}
$$

It is typical to follow closure conversion by hoisting, where all functions are lifted to the top level. To perform hoisting after the above translation requires closing over type variables. In particular, every term of the form $\lambda(\overline{x{:}\vec{\tau}}).e$ is replaced by $x[\vec{\alpha}]$, where x is a fresh global variable, $\vec{\alpha}$ are $\mathrm{ftv}(\lambda(\overline{x{:}\vec{\tau}}).e)$ plus the free variables of their bounds and their bounds *et cetera*, $\vec{\sigma}$ are the bounds of $\vec{\alpha}$, and the global binding $x = \Lambda\overrightarrow{\alpha \le \sigma}.\lambda(\overline{x{:}\vec{\tau}}).e$ is introduced.

This translation could be proven correct by combining a proof of correctness for the embedding of functions into single-method objects, the proof of correctness for object closure conversion [Gle99b], and the proofs in Appendices C and D.

This translation along with Crary's [Cra99] provide the first typed closure-passing-style closure-conversion translations.