Object Closure Conversion *

Neal Glew
Cornell University

24 August 1999

Abstract

An integral part of implementing functional languages is closure conversion—the process of converting code with free variables into closed code and auxiliary data structures. Closure conversion has been extensively studied in this context, but also arises in languages with first-class objects. In fact, one variant of Java's inner classes are an example of objects that need to be closure converted, and the transformation for converting these inner classes into Java Virtual Machine classes is an example of closure conversion.

This paper argues that a direct formulation of object closure conversion is interesting and gives further insight into general closure conversion. It presents a formal closure-conversion translation for a second-order object language and proves it correct. The translation and proof generalise to other object-oriented languages, and the paper gives some examples to support this statement. Finally, the paper discusses the well known connection between function closures and single-method objects. This connection is formalised by showing that an encoding of functions into objects, object closure conversion, and various object encodings compose to give various closure-conversion translations for functions.

1 Introduction

The process of closure conversion and the concept of closures are old and well studied ideas arising in any language with first-class functions. Briefly, if a function f nested within a function g has free variables that are defined in g, the compiler will need to propagate the values of these variables from the time they are computed in g to the times at which f executes. The usual solution is to compile functions to closures, which are data structures that pair closed code with the values of free variables. The application of a function f to an argument a becomes an expression that extracts the closed code for f from the closure and then applies the code to the original argument and a part of the closure containing the values of the free variables.

The necessity for closure conversion is not limited to functional languages. In particular, Abadi and Cardelli's object calculi [AC96] have first-class objects, and an inner-nested object's methods could refer to variables defined in an outer-nested object. It might seem that object closure conversion is less important to mainstream object-oriented languages, so first I shall argue that

*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.

this is not the case. One objection is that mainstream object-oriented languages are class based, and because classes are typically second class, closure conversion is not needed. However, Java was recently extended with *inner classes*, including a form that requires closure conversion. In fact, this form was introduced to alleviate the tedium of manual closure conversion. Another objection is that the combination of an object encoding and functional closure conversion gives object closure conversion. While this indirect approach is adequate, the results of this paper show that this approach misses many important points that a direct account exposes.

First, the indirect approach, if implemented naively, misses opportunities for sharing. Since most object encodings produce separate functions for each method, the methods of an object will be closure converted separately and will not share their environments. A direct approach naturally shares the environments.

Second, the object closure-conversion translation presented here is simpler than typed functional closure-conversion translations. Objects combine code and data in a single construct, so the values of free variables are paired with closed code simply by adding extra fields. Instead of using existential types to hide the types of the environment, the translation uses subsumption, a natural feature of any object calculus.

Third, this paper shows that functional closure conversion is equivalent to the composition of an encoding of functions as objects, object closure conversion, and an object encoding. This result formalises the well known connection between closures and single method objects. We will see that standard choices in functional closure conversion correspond to choices in the object closure conversion and the object encoding.

A difficult and still open issue, even in the functional literature, is the correctness of closure conversion. Minamide *et al.* [MMH95] discuss closure conversion for simply-typed and polymorphically-typed lambda calculi. They describe both conversions as two-step translations, and prove both type preservation and operational correctness. Their notion of correctness is an observational equivalence defined inductively over source types, and their correctness argument is a logical-relations argument. However, this proof does not extend to recursive functions. Morrisett and Harper [MH99] have used a similar technique, but extended with an unwinding lemma, to prove correct a number of typed closure-conversion translations for recursive functions. Their unwinding lemma is proved by defining a model and proving the result there. Unfortunately, it is difficult to define models for languages with recursive types, and harder to prove results in models of languages with state and advanced control features. These proofs are unsatisfying since they do not scale well to real language features.

Steckler and Wand [SW96] describe an optimised closure-conversion process for a simple untyped lambda calculus. They also prove their analysis and transformation correct, and this is a harder task than other proofs described in this paper, as it shows correctness of the optimisations as well as the basic translation. They avoid some of the problems of the above proofs by carefully defining their source semantics and translation so that a simulation argument is possible. Their real contribution, however, is the analysis and optimisation of known closures, and the proof that this optimisation is correct. This proof probably could be used in other settings. As mentioned above, functional closure conversion is object closure conversion composed with an object encoding. Therefore, the proof of object closure conversion's correctness should be simpler. However, it does retain the key difficulty that makes these proofs hard to construct. Thus defining a direct object closure-conversion translation allows us focus on this essential problem.

This paper makes three contributions: it defines a direct object closure-conversion translation, it proves this translation correct using syntactic methods that extend to recursive types, and it relates object closure conversion to functional closure conversion, formalising the well known connection between closures and single-method objects. First, the basic ideas are explained in the setting of a very simple object calculus. The translation is formalised in Section 4, for a second-order object calculus with method parameters described in Section 3. Some extensions of the translation to other language constructs are discussed in Section 5, and the connection between closures and single-method objects is shown in Section 6.

2 The Basic Idea

This section describes the basic idea of a direct object closure-conversion translation. This translation will transform a simple object calculus to itself, taking arbitrary terms as input, and producing closed code as output.

The syntax of the language is:

$$\begin{array}{llll} \text{Types} & \tau, \sigma & ::= & [m_i{:}\tau_i; f_j{:}\sigma_j]_{i \in I, j \in J} \\ \text{Terms} & e, b & ::= & x \mid [m_i = x_i.b_i{:}\tau_i; f_j = e_j]_{i \in I, j \in J} \mid e.m \mid e.f \mid e_1.f \leftarrow e_2 \end{array}$$

The only form of type is the object type $[m_i{:}\tau_i; f_j{:}\sigma_j]_{i \in I, j \in J}$, which is for objects with methods m_i of type τ_i and fields f_j of type σ_j.[1] Objects are created by an object constructor $[m_i = x_i.b_i{:}\tau_i; f_j = e_j]_{i \in I, j \in J}$. The newly created object responds to method m_i by executing b_i with x_i bound to the object, and has e_j as its value for field f_j. Note that e_j is evaluated at the time the object is created, and its free variables do not need to be closed over. Method invocation is written $e.m$, field selection $e.f$, and field update $e_1.f \leftarrow e_2$. I defer the formal semantics and typing rules to the next section.

For functions, the variables that need to be closed over are the free variables of the function. For objects, however, the variables that need to be closed over are not the free variables of the object, but just the free variables of the object's methods. Therefore, I define the notion of closure variables for object constructors:

$$\mathrm{cv}([m_i = x_i.b_i{:}\tau_i; f_j = e_j]_{i \in I, j \in J}) = \bigcup_{i \in I} (\mathrm{fv}(b_i) - \{x_i\})$$

The goal of object closure conversion is to eliminate all closure variables of all object constructors that appear in the program.

The idea behind object closure conversion is simple: In functional closure conversion, closed code is paired with the values of free variables. Since objects already pair code and data, we

[1] The indices i and j range over index sets I and J. For the purposes of this paper, these index sets are unordered and object types and objects are considered equivalent up to reordering. Ordered index sets could also be considered, and the results apply in this case also. The translation defined preserves ordering.

simply add new fields to the object to store the free variables of its methods, and access them through the self variable. For example, the expression

$$[\mathsf{apply} = \mathit{self}_1.[\mathsf{me} = \mathit{self}_2.(\mathit{self}_1.\mathsf{apply}){:}\tau;\,]{:}\tau;\,].\mathsf{apply}$$

where $\tau = [\mathsf{me}{:}[;\,];\,]$ is closure converted to

$$[\mathsf{apply} = \mathit{self}_1.[\mathsf{me} = \mathit{self}_2.(\mathit{self}_2.f.\mathsf{apply}){:}\tau;\, f = \mathit{self}_1]{:}\tau;\,].\mathsf{apply}$$

This idea leads directly to the following syntax-directed translation:

$$\begin{array}{lcl}
|x| & = & x \\
|e.m| & = & |e|.m \\
|e.f| & = & |e|.f \\
|e_1.f \leftarrow e_2| & = & |e_1|.f \leftarrow |e_2| \\
|e| & = & [m_i = x_i.b_i'{:}\tau_i;\, f_j = |e_j|, g_k = y_k]_{i\in I, j\in J, 1\le k\le n} \\
 & & \text{where } \begin{array}[t]{lcl} e & = & [m_i = x_i.b_i{:}\tau_i;\, f_j = e_j]_{i\in I, j\in J} \\ \mathrm{cv}(e) & = & \{y_1, \ldots, y_n\} \\ b_i' & = & |b_i|\{y_1, \ldots, y_n := x_i.g_1, \ldots, x_i.g_n\} \\ g_k \text{ are fresh} & & \end{array}
\end{array}$$

The notation $x\{y := z\}$ denotes capture-avoiding substitution of z for y in z. The translation is straightforward for all expression forms exception object constructors. In particular, notice that method invocation, the analogue of function application, has a trivial translation. For object constructors, the new object has the same methods and fields as the original objects, and some extra fields g_k. There is one extra field for each of the closure variables. After translating the method bodies, each closure variable is replaced by a selection from the self variable of the field that corresponds to the closure variable.

It is worth pointing out that the typing translation is the identity. In functional closure conversion a function type is translated into a more elaborate type, which usually employs an existential to hide the types of the closure variables. In this translation, the new object has the same methods and fields with the same types, and some extra fields. So its principle type is a subtype of the original object's type, and subsumption is used to hide the types of the closure variables. As we shall see, this simplicity is due to the fact that this translation is only half of a functional closure-conversion translation. The other half is an object encoding that does considerable type level translation and further term translation.

Now I will formalise the developments of this section in a scaled up language. This more complex language includes features needed for a comparison with functional closure conversion, in particular, method parameters. Also it supports the claim that the ideas scale to real languages. In particular, the scaled up language will include polymorphism, as past attempts to scale closure conversion to include polymorphism have encountered difficulties.

3 The Object Language

This section formalises an object language. The next section will define a formal translation from this language to itself that takes arbitrary terms to closed terms. The language is a variant of Abadi and Cardelli's second-order object calculus [AC96] with a distinction between methods and fields, variances on fields, method parameters, only unbounded polymorphism, and no method update (lifting the latter two restrictions will be discussed in Section 5). The syntax of the language is:

$$
\begin{array}{llll}
\text{Types} & \tau,\sigma & ::= & \alpha \mid [m_i{:}s_i;\, f_j{:}\sigma_j^{\phi_j}]_{i\in I, j\in J} \mid \forall\alpha.\tau \\
\text{Method Signature} & s & ::= & [\vec{\alpha}](\vec{\tau}) \to \tau \\
\text{Variances} & \phi & ::= & + \mid - \mid \circ \\
\text{Terms} & e, b & ::= & x \mid [m_i = M_i;\, f_j = e_j]_{i\in I, j\in J} \mid e.m[\vec{\tau}](\vec{e}) \mid \\
 & & & e.f \mid e_1.f \leftarrow e_2 \mid \Lambda\alpha.b \mid e[\tau] \\
\text{Method Definition} & M & ::= & x[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).b{:}\tau
\end{array}
$$

The notation $\vec{X}$ denotes a sequence of elements drawn from syntax category X. For example, $\vec{\alpha}$ means $\alpha_1, \ldots, \alpha_n$, and $\overrightarrow{x{:}\tau}$ means $x_1{:}\tau_1, \ldots, x_n{:}\tau_n$.

The types include type variables α, object types $[m_i{:}s_i;\, f_j{:}\sigma_j^{\phi_j}]_{i\in I, j\in J}$, and polymorphic types $\forall\alpha.\tau$. Methods now take both type and value parameters, so have they signatures s instead of types. The signature $[\alpha_1, \ldots, \alpha_m](\tau_1, \ldots, \tau_n) \to \tau$ specifies a methods that takes m type parameters α_1 through α_m and n value parameters of types τ_1 through τ_n and produces a result of type τ. Object types also specify variances ϕ for fields. A read only field has variance $+$; a write only field has variance $-$; a read write field has variance $\circ$.

The terms include variables x, object constructors $[m_i = M_i;\, f_j{=}e_j]_{i\in I, j\in J}$, method invocation $e.m[\vec{\tau}](\vec{e})$, field selection $e.f$, field update $e_1.f \leftarrow e_2$, type abstraction $\Lambda\alpha.b$, and type application $e[\tau]$. An object constructor gives each of its methods a method definition M. The method definition $x[\alpha_1, \ldots, \alpha_m](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).b{:}\tau$ takes type parameters α_1 through α_m and value parameters x_1 through x_n of types τ_1 through τ_n and executes b with x bound to the object. A method invocation $e.m[\vec{\tau}](\vec{e})$ includes both the actual type arguments $\vec{\tau}$ and the actual value arguments $\vec{e}$. I intend a type-erasure interpretation of polymorphism. Consequently, $\Lambda\alpha.b$ does not suspend the execution of b until type application but evaluates it immediately.

There are several syntactic restrictions that simplify the technical development. In particular, the m_i in an object type or an object constructor must be distinct, and similarly for the field names, type parameters, and value parameters. Syntactic objects are equal up to α-equivalence.

The operational semantics appears in Figure 1. The notation $E\langle e\rangle$ denotes the substitution of e for the unique hole $\langle\rangle$ in E. The semantics is a deterministic, left to right, call by value, context based, reduction semantics. Again, note that fields are evaluated to values at object-creation time. The notation $e{\downarrow}$ means that $e \mapsto^* v$ for some v; $e{\uparrow}$ means that e starts an infinite reduction sequence.

The typing rules are standard and appear in Figure 2. The calculus has full breadth and depth subtyping, methods are contravariant in their arguments and covariant in their results, and the depth subtyping of fields is determined by their variance.[2] The notation ϵ is used for an empty typing context. Note that applying a typing rule with Δ, α in a hypothesis, implicitly requires $\alpha \notin \Delta$, and similarly for value contexts.

The typing rules are sound with respect to the operational semantics; this property can be

[2]Abadi and Cardelli [AC96] provide a description of variances and the rules for variance subtyping.

Syntax:

$$
\begin{array}{llll}
\text{Contexts} & E & ::= & \langle\rangle \mid [m_i = M_i; \overrightarrow{f = v}, f = E, \overrightarrow{g = e}]_{i \in I} \mid E.m[\vec{\tau}](\vec{e}) \mid \\
& & & v.m[\vec{\tau}](\vec{v}, E, \vec{e}) \mid E.f \mid E.f \leftarrow e \mid v.f \leftarrow E \mid \\
& & & \Lambda\alpha.E \mid E[\tau] \\
\text{Values} & v, w & ::= & [m_i = M_i; f_j = v_j]_{i \in I, j \in J} \mid \forall\alpha.v
\end{array}
$$

Reduction rules:

$$E\langle\iota\rangle \mapsto E\langle e\rangle$$

Where $v = [m_i = M_i; f_j = v_j]_{i \in I, j \in J}$ and:

ι	e	Side Conditions
$v.m_k[\vec{\tau}](v_1, \ldots, v_n)$	$b\{\vec{\alpha}, x, \vec{x} := \vec{\tau}, v, \vec{v}\}$	$k \in I, M_k = x[\vec{\alpha}](x_1{:}\sigma_1, \ldots, x_n{:}\sigma_m).b$
$v.f_k$	v_k	$k \in J$
$v.f_k \leftarrow w$	$[m_i = M_i; f_j = v'_j]_{i \in I, j \in J}$	$k \in J, v'_j = \begin{cases} v_j & j \neq k \\ w & j = k \end{cases}$
$(\Lambda\alpha.w)[\tau]$	$w\{\alpha := \tau\}$	

Figure 1: Operational Semantics

proven by standard techniques. The proof of correctness uses some other properties of the type system, which I state here; these are also proven by standard techniques.

Lemma 3.1 (Free Variables) *If* $\Delta; \Gamma \vdash e : \tau$ *then* $\mathrm{fv}(e) \subseteq \mathrm{dom}(\Gamma)$. *If* $\Delta \vdash \Gamma$ *then* $\mathrm{ftv}(\Gamma) \subseteq \Delta$.

Lemma 3.2 (Context Strengthening) *If* $\Delta \vdash \tau$ *then* $\Delta, \alpha \vdash \tau$. *If* $\Delta \vdash \tau_1 \leq \tau_2$ *then* $\Delta, \alpha \vdash \tau_1 \tau_2$. *If* $\Delta; \Gamma \vdash e : \tau$ *then* $\Delta, \alpha; \Gamma \vdash e : \tau$. *If* $\Delta; \Gamma_1, x{:}\tau_2, \Gamma_2 \vdash e : \tau$ *and* $\Delta \vdash \tau_1 \leq \tau_2$ *then* $\Delta; \Gamma_1, x{:}\tau_1, \Gamma_2 \vdash e : \tau$.

Lemma 3.3 (Substitution) *If* $\Delta; \Gamma \vdash e_1 : \tau$ *and* $\Delta; \Gamma \vdash e_2 : \Gamma(x)$ *then* $\Delta; \Gamma \vdash e_1\{x := e_2\} : \tau$. *If* $\Delta; x{:}\tau_1 \vdash e : \tau$ *and* $\Delta; y{:}\tau_2 \vdash e' : \tau_1$ *then* $\Delta; y{:}\tau_2 \vdash e\{x := e'\} : \tau$. *If* $\Delta, \alpha; \Gamma \vdash e : \tau$ *and* $\Delta \vdash \sigma$ *then* $\Delta; \Gamma\{\alpha := \sigma\} \vdash e\{\alpha := \sigma\} : \tau\{\alpha := \sigma\}$.

The *free variables* of an expression are:

$$
\begin{array}{lcl}
\mathrm{fv}(x) & = & \{x\} \\
\mathrm{fv}([m_i = M_i; f_j = e_j]_{i \in I, j \in J}) & = & \bigcup_{i \in I} \mathrm{fv}(M_i) \cup \bigcup_{j \in J} \mathrm{fv}(e_j) \\
\mathrm{fv}(e.m[\vec{\tau}](e_1, \ldots, e_n)) & = & \mathrm{fv}(e) \cup \bigcup_{1 \leq i \leq n} \mathrm{fv}(e_i) \\
\mathrm{fv}(e.f) & = & \mathrm{fv}(e) \\
\mathrm{fv}(e_1.f \leftarrow e_2) & = & \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \\
\mathrm{fv}(\Lambda\alpha.b) & = & \mathrm{fv}(b) \\
\mathrm{fv}(e[\sigma]) & = & \mathrm{fv}(e) \\
\mathrm{fv}(x[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).b{:}\tau) & = & \mathrm{fv}(b) - \{x, x_1, \ldots, x_n\}
\end{array}
$$

The *closure variables* of an object constructor are:

$$\mathrm{cv}([m_i = M_i; f_j = e_j]_{i \in I, j \in J}) = \bigcup_{i \in I} \mathrm{fv}(M_i)$$

Typing Contexts:

$$\begin{array}{lcll} \Delta & ::= & \alpha_1, \ldots, \alpha_n & \text{where } \alpha_1, \ldots, \alpha_n \text{ are distinct} \\ \Gamma & ::= & x_1{:}\tau_1, \ldots, x_n{:}\tau_n & \text{where } x_1, \ldots, x_n \text{ are distinct} \end{array}$$

Type well formedness, typing context well formedness, and subtyping:

$$\frac{}{\Delta \vdash \tau}\ (\mathrm{ftv}(\tau) \subseteq \Delta) \qquad \frac{\Delta \vdash \tau_i}{\Delta \vdash x_1 : \tau_1, \ldots, x_n : \tau_n} \qquad \frac{}{\Delta \vdash \alpha \leq \alpha}\ (\alpha \in \Delta)$$

$$\frac{\begin{array}{ll} i \in I_2 : & \Delta \vdash s_i \leq s'_i \\ j \in J_2 : & \Delta \vdash \sigma_j^{\phi_j} \leq {\sigma'}_j^{\phi'_j} \\ & \Delta \vdash [m_i{:}s_i;\, f_j{:}\sigma_j^{\phi_j}]_{i \in I_1, j \in J_1} \end{array}}{\Delta \vdash [m_i{:}s_i;\, f_j{:}\sigma_j^{\phi_j}]_{i \in I_1, j \in J_1} \leq [m_i{:}s'_i;\, f_j{:}{\sigma'}_j^{\phi'_j}]_{i \in I_2, j \in J_2}}\ (I_2 \subseteq I_1; J_2 \subseteq J_1)$$

$$\frac{\Delta, \alpha \vdash \tau_1 \leq \tau_2}{\Delta \vdash \forall\alpha.\tau_1 \leq \forall\alpha.\tau_2} \qquad \frac{\Delta, \vec{\alpha} \vdash \sigma_i \leq \tau_i \quad \Delta, \vec{\alpha} \vdash \tau \leq \sigma}{\Delta \vdash [\vec{\alpha}](\tau_1, \ldots, \tau_n) \to \tau \leq [\vec{\alpha}](\sigma_1, \ldots, \sigma_n) \to \sigma}$$

$$\frac{\Delta \vdash \sigma_1 \leq \sigma_2}{\Delta \vdash \sigma_1^{\phi} \leq \sigma_2^{+}}\ (\phi \in \{+, \circ\}) \quad \frac{\Delta \vdash \sigma_2 \leq \sigma_1}{\Delta \vdash \sigma_1^{\phi} \leq \sigma_2^{-}}\ (\phi \in \{-, \circ\}) \quad \frac{\Delta \vdash \sigma}{\Delta \vdash \sigma^{\circ} \leq \sigma^{\circ}}$$

Expression typing:

$$\frac{\Delta; \Gamma \vdash e : \tau_1 \quad \vdash \tau_1 \leq \tau_2}{\Delta; \Gamma \vdash e : \tau_2} \qquad \frac{}{\Delta; \Gamma \vdash x : \tau}\ (\Gamma(x) = \tau)$$

$$\frac{\Delta \vdash \tau \quad \Delta; \Gamma; \tau \vdash M_i : s_i \quad \Delta; \Gamma \vdash e_j : \sigma_j}{\Delta; \Gamma \vdash [m_i = M_i;\, f_j = e_j]_{i \in I, j \in J} : \tau}\ (\tau = [m_i{:}s_i;\, f_j{:}\sigma_j^{\circ}]_{i \in I, j \in J})$$

$$\frac{\Delta; \Gamma \vdash e : [m{:}[\alpha_1, \ldots, \alpha_m](\tau_1, \ldots, \tau_n) \to \tau;\,] \quad \Delta \vdash \sigma_i \quad \Delta; \Gamma \vdash e_i : \tau_i\{\vec{\alpha} := \vec{\sigma}\}}{\Delta; \Gamma \vdash e.m[\sigma_1, \ldots, \sigma_m](e_1, \ldots, e_n) : \tau\{\vec{\alpha} := \vec{\sigma}\}}$$

$$\frac{\Delta; \Gamma \vdash e : [;\, f{:}\sigma^{\phi}]}{\Delta; \Gamma \vdash e.f : \sigma_k}\ (\phi \in \{+, \circ\})$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau \quad \Delta \vdash \tau \leq [;\, f{:}\sigma^{\phi}] \quad \Delta; \Gamma \vdash e_2 : \sigma}{\Delta; \Gamma \vdash e_1.f \leftarrow e_2 : \tau}\ (\phi \in \{-, \circ\})$$

$$\frac{\Delta, \alpha; \Gamma \vdash b : \tau}{\Delta; \Gamma \vdash \Lambda\alpha.b : \forall\alpha.\tau} \qquad \frac{\Delta; \Gamma \vdash e : \forall\alpha.\tau \quad \Delta \vdash \sigma}{\Delta; \Gamma \vdash e[\sigma] : \tau\{\alpha := \sigma\}}$$

$$\frac{\Delta, \vec{\alpha}; \Gamma, x{:}\tau, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash b : \sigma}{\Delta; \Gamma; \tau \vdash x[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).b{:}\sigma : [\vec{\alpha}](\tau_1, \ldots, \tau_n) \to \sigma}$$

Figure 2: Typing Rules

$$
\begin{array}{lcl}
|x| & = & x \\
|e.m[\vec{\tau}](e_1,\ldots,e_n)| & = & |e|.m[\vec{\tau}](|e_1|,\ldots,|e_n|) \\
|e.f| & = & |e|.f \\
|e_1.f \leftarrow e_2| & = & |e_1|.f \leftarrow |e_2| \\
|\Lambda\alpha.b| & = & \Lambda\alpha.|b| \\
|e[\sigma]| & = & |e|[\sigma] \\
|e| & = & e'
\end{array}
$$

$$
\begin{array}{llcl}
\text{where} & e & = & [m_i = M_i; f_j = e_j]_{i\in I, j\in J} \\
 & e' & = & [m_i = M_i'; f_j = |e_j|, g_1 = y_1,\ldots,g_n = y_n]_{i\in I,j\in J} \\
 & \mathrm{cv}(e) & = & \{y_1,\ldots,y_n\} \\
 & M_i & = & x_i[\vec{\alpha}_i](\overrightarrow{x_i{:}\tau_i}).b_i{:}\tau_i \\
 & M_i' & = & x_i[\vec{\alpha}_i](\overrightarrow{x_i{:}\tau_i}).|b_i|\{y_1,\ldots,y_n := x_i.g_1,\ldots,x_i.g_n\}{:}\tau_i \\
 & g_1,\ \ldots,\ g_n & \text{are fresh} &
\end{array}
$$

Figure 3: Object Closure Conversion Translation

Note that the closure variables do not include the free variables of the field initialisers, e_j, because the field initialisers are evaluated at object-creation time. An object constructor is *code closed* if and only if it has no closure variables. An arbitrary expression is *code closed* if and only if every object-constructor subexpression is code closed.

4 Object Closure Conversion

This section formalises the translation described in Section 2 for the object language in Section 3. Exactly the same ideas are used: An object constructor is extended with fields for its closure variables, and method bodies reference these closure variables by field selection of the self variable. The typing translation is the identity translation, and the term translation is defined inductively over the syntax of expressions and appears in Figure 3.

4.1 Observational Equivalence

An important aspect of the correctness of the translation is that it preserves the meaning of expressions. There are a number of ways to define notions of meaning preservation. Unfortunately, the simplest of these, simulation arguments, does not hold for this language. Consider the example given in Section 2. The source term makes this transition:

$$
\begin{array}{ll}
 & [\mathsf{apply} = \mathit{self}_1.[\mathsf{me} = \mathit{self}_2.(\mathit{self}_1.\mathsf{apply}){:}\tau;\,]{:}\tau;\,].\mathsf{apply} \\
\mapsto & [\mathsf{me} = \mathit{self}_2.([\mathsf{apply} = \mathit{self}_1.[\mathsf{me} = \mathit{self}_2.(\mathit{self}_1.\mathsf{apply}){:}\tau;\,]{:}\tau;\,].\mathsf{apply}){:}\tau;\,]
\end{array}
$$

But the translated term makes this transition:

$$
\begin{array}{ll}
 & [\mathsf{apply} = \mathit{self}_1.[\mathsf{me} = \mathit{self}_2.(\mathit{self}_2.f.\mathsf{apply}){:}\tau;\, f = \mathit{self}_1]{:}\tau;\,].\mathsf{apply} \\
\mapsto & [\mathsf{me} = \mathit{self}_2.(\mathit{self}_2.f.\mathsf{apply}){:}\tau; \\
 & \ f = [\mathsf{apply} = \mathit{self}_1.[\mathsf{me} = \mathit{self}_2.(\mathit{self}_2.f.\mathsf{apply}){:}\tau;\, f = \mathit{self}_1]{:}\tau;\,]]
\end{array}
$$

The reduced source term is code closed and translates to itself not the reduced translated term. In particular, notice that the outer object is part of the method body of `me` in the reduced source term but an extra field in the reduced target term. In general, closure conversion shifts free variables to extra fields, but does not shift the values that get substituted for them to extra fields. In effect the environment is in the method bodies in the source term, but in extra fields in the target term. While this does not change the behaviour of terms, a simulation argument cannot prove correctness.

Instead, this paper uses contextual equivalence [Mor68, Plo77]: two terms are equivalent if they are indistinguishable in any context of the language. Following standard practice, termination behaviour is used as the primitive notion of observable difference. The formal definition of contextual equivalence, which I shall call observational equivalence, requires the definition of contexts.

Contexts C are an extension of the syntax category e with holes $\langle\rangle$. Holes may appear in a number of places in C and within type or term variable binders, and these binders capture the free type and term variables of the hole. The notation $C\langle e\rangle$ denotes the expression that results from replacing the holes in C with e. A context C is typed by the judgement $\Delta_1;\Gamma_1 \vdash C : \tau_1\langle\Delta_2;\Gamma_2{:}\tau_2\rangle$, and this holds exactly when $\Delta_1;\Gamma_1 \vdash C : \tau_1$ is derivable with the extra rule $\Delta_1,\Delta_2;\Gamma_1,\Gamma_2 \vdash \langle\rangle : \tau_2$. Clearly if $\Delta_1,\Delta_2;\Gamma_1,\Gamma_2 \vdash e : \tau_2$ then $\Delta_1;\Gamma_1 \vdash C\langle e\rangle : \tau_1$.

Most previous work, which considered untyped languages, used an untyped version of equivalence. However, closure conversion is not correct under this version. A translated object constructor has extra fields, so a context that selects these extra fields will for the source term get stuck, and for the translated term converge. Therefore, the correctness criteria must rule out run-time type errors by using a typed version of equivalence. A term and its translation will be equivalent at the type of the source term.

Definition 4.1 (Observational Equivalence)

- *e_1 and e_2 are Kleene equivalent, $e_1 \sim e_2$, iff:* [3]

$$\epsilon;\epsilon \vdash e_1 : [;] \quad\wedge\quad \epsilon;\epsilon \vdash e_2 : [;] \quad\wedge\quad e_1\!\downarrow \Leftrightarrow e_2\!\downarrow \quad\wedge\quad e_1\!\uparrow \Leftrightarrow e_2\!\uparrow$$

- *e_1 and e_2 are $\Delta;\Gamma$-observationally equivalent at τ, $e_1 \equiv_{\Delta;\Gamma:\tau} e_2$, iff:*

$$\Delta;\Gamma \vdash e_1 : \tau \quad\wedge\quad \Delta;\Gamma \vdash e_2 : \tau \quad\wedge\quad \forall C : \epsilon;\epsilon \vdash C : [;]\langle\Delta;\Gamma{:}\tau\rangle \Rightarrow C\langle e_1\rangle \sim C\langle e_2\rangle$$

Note that the first two clauses of observational equivalence imply the first two clauses of the Kleene equivalence of the third clause, so proofs of observational equivalence need only establish the first two clauses of observational equivalence and the third and fourth clause of the Kleene equivalence for all appropriate contexts.

The above relations are equivalence relations and observational equivalence is a congruence. These properties are proven in Appendix B, as part of a basic theory of observational equivalence. Some basic properties are proven: equivalence is preserved under equivalent substitutions and under reduction, and equivalence at a subtype is finer than equivalence at a supertype.

[3] The third and fourth clauses are redundant in a deterministic language, such as the one of this paper. I state the more general definition to be consistent with my scalability theme.

Some principles for establishing equivalence are also proven. First, to show arbitrary open terms are equivalent it suffices to show that under all appropriate substitutions they are equivalent. Second, to show closed terms are equivalent it suffices to show that the values they evaluate to are equivalent.

Third, a coinduction principle is proven. To motivate this principle, consider proving v_1 and v_2 equivalent at $[m{:}\tau;]$. Placing v_1 and v_2 in arbitrary contexts results in arbitrary reductions, but such reductions will not observe a difference unless it includes a method invocation of m. Thus, most previous work includes a theorem of the form that v_1 and v_2 are equivalent if they are Kleene equivalent in all contexts of the form $C\langle\langle\rangle.m\rangle$. However, if $v_1.m \mapsto e_1$ and $v_2.m \mapsto e_2$ this still requires reasoning about e_1 and e_2 in arbitrary contexts. My coinduction principle goes further and considers just contexts of the form $\langle\rangle.m$. It requires showing that $(v_1, v_2, [m{:}\tau;])$ is a member of a set of triples that is closed under reduction. For this example, closed under reduction means that e_1 and e_2 are either equivalent or that (e_1, e_2, τ) is another triple in the set. In general the type at which v_1 and v_2 are to be equivalent determines the contexts that need to be considered, details are in Appendices A and B.

This coinduction principle is not enough. Consider proving $[m = x.b_1;]$ equivalent to $[m = x.b_2;]$ as part of a proof by induction on the structure of expressions. Using the coinduction principle this requires showing that $b_i\{x := [m = x.b_i;]\}$ for $i \in \{1, 2\}$ are equivalent or in some set of triples. The induction hypothesis says that b_1 and b_2 are equivalent assuming x is substituted with equivalent values, but x is substituted by the values we are trying to prove equivalent. This is the fundamental problem with extending the proof of Minamide *et al.* [MMH95] to recursive functions. Morrisett and Harper [MH99] use an unwinding lemma to solve this problem. The 0th unwinding of $[m = x.b_i;]$ for $i \in \{1, 2\}$ diverge when method m is invoked so they are clearly equivalent. The $n+1$th unwinding reduces to b_i with the nth unwinding substituted for x, so by induction on n, we can establish that the nth unwindings of the objects are equivalent for all n. The unwinding lemma then tells us that the objects themselves are equivalent. The basic theory mentioned above proves an unwinding lemma for the object language, and a more powerful coinduction principle.

4.2 Correctness

There are three aspects to correctness: the translation produces code closed expressions, preserves types, and preserves the meaning of expressions.

Theorem 4.1 (Translation Correctness) *If* $\Delta \vdash \Gamma$ *and* $\Delta; \Gamma \vdash e : \tau$ *then:*

- $|e|$ *is code closed*
- $\Delta; \Gamma \vdash |e| : \tau$
- e *and* $|e|$ *are* $\Delta; \Gamma$*-observationally equivalent at* τ

Before proving correctness, I will prove some auxiliary lemmas.

Lemma 4.2 *If $\Delta;\Gamma \vdash e : \tau$ and $\Delta;\Gamma \vdash y_k : \sigma'_k$ then $e \equiv_{\Delta;\Gamma:\tau} e'$ where:*

$$\begin{array}{rcl} e & = & [m_i = M_i; f_j = e_j]_{i\in I, j\in J} \\ e' & = & [m_i = M_i; f_j = e_j, g_k = y_k]_{i\in I, j\in J, k\in K} \\ \tau & = & [m_i{:}s_i; f_j{:}\sigma_j^\circ]_{i\in I, j\in J} \end{array}$$

Proof: The result follows by Lemma B.10 if the right hand side of that lemma holds. Let ρ be a value substitution such that $\Delta \vdash \rho : \Gamma$. If $e\{\rho\}\uparrow$ then by inspection of the reduction rules, there must be $j \in J$ such that $e_j\{\rho\}\uparrow$. Therefore $e'\{\rho\}\uparrow$ also. Similarly $e'\{\rho\}\uparrow$ implies $e'\{\rho\}\uparrow$. Now if either $e\{\rho\}$ or $e'\{\rho\}$ converges then the other also converges and they converge to values of the form $v = [m_i = M_i\{\rho\}; f_j = v_j]_{i\in I, j\in J}$ and $v' = [m_i = M_i\{\rho\}; f_j = v_j, g_k = \rho(y_k)]_{i\in I, j\in J, k\in K}$ respectively. It remains to show that $v \equiv_{\Delta;\epsilon:\tau} v'$. By Lemma B.16 this follows if A is closed under reductions where:

$$A = \left\{ (v_1, v_2, \tau) \;\middle|\; \begin{array}{l} v_1 = [m_i = M_i\{\rho\}; f_j = v_{1j}]_{i\in I, j\in J} \\ v_2 = [m_i = M_i\{\rho\}; f_j = v_{2j}, g_k = \rho(y_k)]_{i\in I, j\in J, k\in K} \\ v_{1j} \equiv_{\Delta;\epsilon:\sigma_j} v_{2j} \end{array} \right\}$$

The reductions that apply are $\langle\rangle.m_i[\vec{\sigma}](\vec{v})$ for $i \in I$, $\langle\rangle.f_j$ for $j \in J$, and $\langle\rangle.f_j \leftarrow v$ for $j \in J$.

For the first reduction assume $M_i = x[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).b{:}\tau'$ and $s_i = [\vec{\alpha}](\tau_1, \ldots, \tau_n) \to \tau'$. Note that $\Delta, \vec{\alpha}; \Gamma, x{:}\tau, \overrightarrow{x{:}\tau} \vdash b : \tau'$. Let $\tau''_i = \tau_i\{\vec{\alpha} := \vec{\sigma}\}$, $\tau'' = \tau'\{\vec{\alpha} := \vec{\sigma}\}$, and $\Gamma' = \Gamma, x{:}\tau, \overrightarrow{x{:}\tau''}$. Let $v_{1h} \equiv_{\Delta;\epsilon:\tau''_h} v_{2h}$ for $1 \le h \le n$ then:

$$\begin{array}{rcl} v_1.m_i[\vec{\sigma}](\vec{v_1}) & \mapsto & b\{\rho\}\{\vec{\alpha}, x, \vec{x} := \vec{\sigma}, v_1, \vec{v_1}\} \\ & = & b\{\vec{\alpha} := \vec{\sigma}\}\{\rho, \vec{x} := \vec{v_1}\}\{x := v_1\} \\ v_2.m_i[\vec{\sigma}](\vec{v_2}) & \mapsto & b\{\rho\}\{\vec{\alpha}, x, \vec{x} := \vec{\sigma}, v_2, \vec{v_2}\} \\ & = & b\{\vec{\alpha} := \vec{\sigma}\}\{\rho, \vec{x} := \vec{v_2}\}\{x := v_2\} \end{array}$$

By Substitution $\Delta; \Gamma, x{:}\tau, \overrightarrow{x{:}\tau''} \vdash b\{\vec{\alpha} := \vec{\sigma}\} : \tau'\{\vec{\alpha}' := \vec{\sigma}\}$, so by reflexivity $b\{\vec{\alpha} := \vec{\sigma}\} \equiv_{\Delta;\Gamma':\tau''} b\{\vec{\alpha} := \vec{\sigma}\}$. Clearly, $\rho, \vec{x} := \vec{v_1} \equiv_{\Delta:\Gamma,\overrightarrow{x:\tau''}} \rho, \vec{x} := \vec{v_2}$, so by Lemma B.14 $b\{\vec{\alpha} := \vec{\sigma}\}\{\rho_1, \vec{x} := \vec{v_1}\} \equiv_{\Delta;x:\tau:\tau''} b\{\vec{\alpha} := \vec{\sigma}\}\{\rho_2, \vec{x} := \vec{v_2}\}$ as required.

For the second reduction $v_1.f_j \mapsto v_{1j}$, $v_2.f_j \mapsto v_{2j}$, and $v_{1j} \equiv_{\Delta;\epsilon:\sigma_j} v_{2j}$ as required. For the third reduction let $v\{f := v'\}$ denote the value v with its f field replaced by v'. Let $v'_1 \equiv_{\Delta;\epsilon:\sigma_j} v'_2$ then $v_1.f_j \leftarrow v'_1 \mapsto v_1\{f_j := v'_1\}$ and $v_2.f_j \leftarrow v'_2 \mapsto v_2\{f_j := v'_2\}$. Clearly $(v_1\{f_j := v'_1\}, v_2\{f_j := v'_2\}, \tau) \in A$. □

Lemma 4.3 *If $\Delta;\Gamma \vdash e_1 : \tau$ and $b_1 \equiv_{\Delta';\Gamma':\sigma} b_2$ then $e_1 \equiv_{\Delta;\Gamma:\tau} e_2$ where:*

$$\begin{array}{rcl} e_1 & = & [m = x[\vec{\alpha}](\overrightarrow{x{:}\tau}).b_1{:}\sigma, m_i = M_i; f_j = e_j, g_k = y_k]_{i\in I, j\in J, k\in K} \\ e_2 & = & [m = x[\vec{\alpha}](\overrightarrow{x{:}\tau}).b_2\{\rho_2\}{:}\sigma, m_i = M_i; f_j = e_j, g_k = y_k]_{i\in I, j\in J, k\in K} \\ \rho_2 & = & \{y_k := x.g_k\}_{k\in K} \\ \tau & = & [m{:}[\vec{\alpha}](\vec{\tau}) \to \sigma, m_i{:}s_i; f_j{:}\sigma_j^\circ, g_k{:}{\sigma'_k}^+]_{i\in I, j\in J, k\in K} \\ \Delta' & = & \Delta, \vec{\alpha} \\ \Gamma' & = & \Gamma, x{:}\tau, \overrightarrow{x{:}\tau} \end{array}$$

Proof: Let $X(b) = x[\vec{\alpha}](\overrightarrow{x{:}\tau}).b{:}\sigma$. The proof is very similar to the one for Lemma 4.2. Instead of A the following set suffices:

$$\left\{ (v_1, v_2, \tau) \;\middle|\; \begin{array}{l} v_1 = [m = X(b_1\{\rho\}), m_i = M_i\{\rho\}; f_j = v_{1j}, g_k = \rho(y_k)]_{i\in I, j\in J, k\in K} \\ v_2 = [m = X(b_2\{\rho_2\}\{\rho\}), m_i = M_i\{\rho\}; f_j = v_{2j}, g_k = \rho(y_k)]_{i\in I, j\in J, k\in K} \\ v_{1j} \equiv_{\Delta;\epsilon:\sigma_j} v_{2j} \end{array} \right\}$$

The reasoning is very similar to that in Lemma 4.2 except for the case of invocation of method m. Assuming n value parameters, again with $\tau''_i = \tau_i\{\vec{\alpha} := \vec{\sigma}\}$, let $v_{1i} \equiv_{\Delta;\epsilon:\tau''_i} v_{2i}$. Then:

$$\begin{array}{rcl} v_1.m[\vec{\sigma}](\vec{v_1}) & \mapsto & b_1\{\rho\}\{\vec{\alpha}, x, \vec{x} := \vec{\sigma}, v_1, \vec{v_1}\} \\ & = & b_1\{\vec{\alpha} := \vec{\sigma}\}\{\rho\}\{\vec{x} := \vec{v_1}\}\{x := v_1\} \\ v_2.m[\vec{\sigma}](\vec{v_2}) & \mapsto & b_2\{\rho_2\}\{\rho\}\{\vec{\alpha}, x, \vec{x} := \vec{\sigma}, v_2, \vec{v_2}\} \\ & = & b_2\{\vec{\alpha} := \vec{\sigma}\}\{\rho'\}\{\vec{x} := \vec{v_2}\}\{x := v_2\} \\ \text{where } \rho'(z) & = & \left\{ \begin{array}{ll} \rho(z) & \forall k \in K : z \neq y_k \\ v_2.g_k & z = y_k \end{array} \right. \end{array}$$

Since $v_2.g_k \mapsto \rho(y_k)$, by Lemma B.9 $\rho(y_k) \equiv_{\Delta;\epsilon:\sigma'_k} v_2.g_k$. Therefore $\rho \equiv_{\Delta:\Gamma} \rho'$. Let $\Gamma'' = x{:}\tau, \overrightarrow{x{:}\tau''}$. By assumption and Lemma B.11 $b_1\{\vec{\alpha} := \vec{\sigma}\} \equiv_{\Delta;\Gamma,\Gamma'':\sigma\{\vec{\alpha}:=\vec{\sigma}\}} b_2\{\vec{\alpha} := \vec{\sigma}\}$. By Lemma B.14 $b_1\{\vec{\alpha} := \vec{\sigma}\}\{\rho\} \equiv_{\Delta;\Gamma'':\sigma\{\vec{\alpha}:=\vec{\sigma}\}} b_2\{\vec{\alpha} := \vec{\sigma}\}\{\rho'\}$. Clearly $\vec{x} := \vec{v_1} \equiv_{\Delta:\overrightarrow{x:\tau''}} \vec{x} := \vec{v_2}$ so by Lemma B.14 $b_1\{\vec{\alpha} := \vec{\sigma}\}\{\rho\}\{\vec{x} := \vec{v_1}\} \equiv_{\Delta;x:\tau:\sigma\{\vec{\alpha}:=\vec{\sigma}\}} b_2\{\vec{\alpha} := \vec{\sigma}\}\{\rho'\}\{\vec{x} := \vec{v_2}\}$ as required. □

Proof (Correctness): First, I will show that $|e|$ is code closed and that $\mathrm{fv}(|e|) \subseteq \mathrm{fv}(e)$. The proof proceeds by induction on the structure of e, and splits into cases based on the outer form of e. All cases are straightforward except for e an object constructor. In this case:

$$\begin{array}{rcl} \mathrm{cv}(|e|) & = & \cup_{i\in I}\mathrm{fv}(M'_i) \\ & = & \cup_{i\in I}(\mathrm{fv}(b'_i) - \{x_i, \vec{x_i}\}) \\ & \subseteq & \cup_{i\in I}(((\mathrm{fv}(|b_i|) - \{y_k\}_{1\le k\le n}) \cup \{x_i\}) - \{x_i, \vec{x_i}\}) \\ & \subseteq & \cup_{i\in I}(((\mathrm{fv}(b_i) - \{y_k\}_{1\le k\le n}) \cup \{x_i\}) - \{x_i, \vec{x_i}\}) \\ & \subseteq & \cup_{i\in I}((\{x_i, \vec{x_i}\} \cup \{x_i\}) - \{x_i, \vec{x_i}\}) \\ & = & \emptyset \end{array}$$

The subterms of $|e|$ are code closed by the induction hypothesis and inspection (note that the substitutions $|b_i|\{y_k := x_i.g_k\}$ do not introduce new closure variables in code closed subterms). So $|e|$ is code closed. Finally, $\mathrm{fv}(|e|) = \mathrm{cv}(e) \cup_{j\in J} \mathrm{fv}(|e_j|) \cup \{y_k\}_{1\le k\le n} \subseteq \emptyset \cup_{j\in J} \mathrm{fv}(e_j) \cup \{y_k\}_{1\le k\le n} \subseteq \mathrm{fv}(e)$.

Next, I will establish that $\Delta; \Gamma \vdash |e| : \tau$. The proof proceeds by induction on the derivation of $\Delta; \Gamma \vdash e : \tau$, and splits into cases based on the last rule used:

Subsumption Rule In this case, $\tau = \tau_2$, $\Delta; \Gamma \vdash e : \tau_1$, and $\Delta \vdash \tau_1 \le \tau_2$. By the induction hypothesis $\Delta; \Gamma \vdash |e| : \tau_1$ and so by subsumption $\Delta; \Gamma \vdash |e| : \tau_2$ as required.

Variable Rule In this case, $e = x = |e|$, so the result follows immediately from the assumption.

Object Rule In this case, $e = [m_i = M_i; f_j = e_j]_{i\in I, j\in J}$, $\tau = [m_i : s_i; f_j : \sigma^\circ_j]_{i\in I, j\in J}$, $\Delta; \Gamma; \tau \vdash M_i : s_i$, and $\Delta; \Gamma \vdash e_j : \sigma_j$. Let $y_1, \ldots, y_n$ be the closure variables for e. By Free Variables

$y_i \in \mathrm{dom}(\Gamma)$. Let $\sigma'_k = \Gamma(y_k)$, and $\tau' = [m_i : s_i; f_j : \sigma_j^\circ, g_k : {\sigma'_k}^\circ]_{i \in I, j \in J, 1 \leq k \leq n}$. Since $g_{1 \leq k \leq p}$ were chosen to be fresh, it is easy to establish that $\Delta \vdash \tau'$ and that $\Delta \vdash \tau' \leq \tau$. Assume for the moment that $\Delta; \Gamma; \tau' \vdash M'_i : s_i$. By the induction hypothesis $\Delta; \Gamma \vdash |e_j| : \sigma_j$. Clearly, $\Delta; \Gamma \vdash y_k : \sigma'_k$, so by the Object Rule $\Delta; \Gamma \vdash |e| : \tau'$. The result follows by subsumption.

It remains to show that $\Delta; \Gamma \vdash M'_i : s_i$. Consider an arbitrary $M_i = x[\vec{\alpha}](\overrightarrow{x{:}\tau}).b{:}\sigma$. By the Method Definition Rule $s_i = [\vec{\alpha}](\vec{\tau}) \to \sigma$ and $\Delta, \vec{\alpha}; \Gamma, x{:}\tau, \overrightarrow{x{:}\tau} \vdash b : \sigma$. By the induction hypothesis $\Delta, \vec{\alpha}; \Gamma, x{:}\tau, \overrightarrow{x{:}\tau} \vdash |b| : \sigma$. By Context Strengthening $\Delta, \vec{\alpha}; \Gamma, x{:}\tau', \overrightarrow{x{:}\tau} \vdash |b| : \sigma$. By the Variable Rule and the Field Selection Rule $\Delta, \vec{\alpha}; \Gamma, x{:}\tau', \overrightarrow{x{:}\tau} \vdash x.g_k : \sigma'_k$, so by Substitution $\Delta, \vec{\alpha}; \Gamma, x{:}\tau', \overrightarrow{x{:}\tau} \vdash |b|\{y_1, \ldots, y_n := x.g_1, \ldots, x.g_n\} : \sigma$. So by the Method Definition Rule $\Delta; \Gamma; \tau' \vdash M_i : s_i$.

Method Invocation Rule In this case, $e = e'.m[\vec{\sigma}](\vec{e})$, $\tau = \tau'\{\vec{\alpha} := \vec{\sigma}\}$, $\Delta \vdash \sigma_i$, $\Delta; \Gamma \vdash e' : [m{:}[\vec{\alpha}](\vec{\tau}) \to \tau';]$, and $\Delta; \Gamma \vdash e_i : \tau_i\{\vec{\alpha} := \vec{\sigma}\}$. By the induction hypothesis $\Delta; \Gamma \vdash |e'| : [m{:}[\vec{\alpha}](\vec{\tau}) \to \tau';]$ and $\Delta; \Gamma \vdash |e_i| : \tau_i\{\vec{\alpha} := \vec{\sigma}\}$. So by the Method Invocation Rule $\Delta; \Gamma \vdash |e'|.m[\vec{\sigma}](\overrightarrow{|e|}) : \tau$. The result follows since $|e'|.m[\vec{\sigma}](\overrightarrow{|e|}) = |e.m[\vec{\sigma}](\vec{e})| = |e|$.

Field Selection Rule In this case, $e = e'.f$, $\Delta; \Gamma \vdash e' : [; f : \tau^\phi]$, and $\phi \in \{+, \circ\}$. By the induction hypothesis $\Delta; \Gamma \vdash |e'| : [; f : \tau^\phi]$. By the field selection rule $\Delta; \Gamma \vdash |e'|.f : \tau$. The result follows since $|e'|.f = |e'.f| = |e|$.

Field Update Rule In this case, $e = e_1.f \leftarrow e_2$, $\tau = [; f : \sigma^\phi]$, $\Delta; \Gamma \vdash e_1 : \tau$, $\Delta; \Gamma \vdash e_2 : \sigma$, and $\phi \in \{-, \circ\}$. By the induction hypothesis twice, $\Delta; \Gamma \vdash |e_1| : \tau$ and $\Delta; \Gamma \vdash |e_2| : \sigma$. By the field update rule $\Delta; \Gamma \vdash |e_1|.f \leftarrow |e_2| : \tau$. The result follows since $|e_1|.f \leftarrow |e_2| = |e_1.f \leftarrow e_2| = |e|$.

Type Abstraction Rule In this case, $e = \Lambda\alpha.e'$, $\tau = \forall\alpha.\tau'$, and $\Delta, \alpha; \Gamma \vdash e' : \tau'$. By the induction hypothesis $\Delta, \alpha; \Gamma \vdash |e'| : \tau'$, so by the Type Abstraction Rule $\Delta; \Gamma \vdash \Lambda\alpha.|e'| : \tau$. The result follows since $\Lambda\alpha.|e'| = |\Lambda\alpha.e'| = |e|$.

Type Application Rule In this case, $e = e'[\sigma]$, $\tau = \tau'\{\alpha := \sigma\}$, $\Delta; \Gamma \vdash e' : \forall\alpha.\tau'$, and $\Delta \vdash \sigma$. By the induction hypothesis $\Delta; \Gamma \vdash |e'| : \forall\alpha.\tau'$, so by the Type Application Rule $\Delta; \Gamma \vdash |e'|[\sigma] : \tau'\{\alpha := \sigma\}$. The result follows since $|e'|[\sigma] = |e'[\sigma]| = |e|$.

Finally, I will show that $e \equiv_{\Delta;\Gamma:\tau} |e|$. Note that only the third clause of the definition of observational equivalence needs to be established. The proof proceeds by induction of the structure of e and then on the height of the derivation of $\Delta; \Gamma \vdash e : \tau$, and splits into cases based on the last rule in the derivation:

Subsumption Rule In this case, $\tau = \tau_2$, $\Delta; \Gamma \vdash e : \tau_1$, and $\Delta \vdash \tau_1 \leq \tau_2$. By the induction hypothesis $e \equiv_{\Delta;\Gamma:\tau_1} |e|$. By OE Refinement $e \equiv_{\Delta;\Gamma:\tau_2} |e|$ as required.

Variable Rule In this case, $e = x = |e|$ and $\tau = \Gamma(x)$. The result follows by reflexivity as proven in Lemma B.1.

Object Rule For this case:

$$
\begin{array}{lcl}
e & = & [m_i = M_i; f_j = e_j]_{1\le i\le m, 1\le j\le n} \\
|e| & = & [m_i = M_i'; f_j = |e_j|, g_p = y_p]_{1\le i\le m, 1\le j\le n, 1\le k\le p} \\
\mathrm{cv}(e) & = & \{y_1, \ldots, y_p\} \\
M_i & = & x_i[\vec{\alpha_i}](\overrightarrow{x_i{:}\tau_i}).b_i{:}\tau_i \\
M_i' & = & x_i[\vec{\alpha_i}](\overrightarrow{x_i{:}\tau_i}).b_i'{:}\tau_i \\
b_i' & = & |b_i|\{y_1, \ldots, y_p := x_i.g_1, \ldots, x_i.g_p\} \\
\tau & = & [m_i{:}s_i; f_j{:}\sigma_j^\circ]_{1\le i\le m, 1\le j\le n} \\
\tau' & = & [m_i{:}s_i; f_j{:}\sigma_j^\circ, g_k{:}\sigma_k'^{+}]_{1\le i\le m, 1\le j\le n, 1\le k\le p} \\
s_i & = & [\vec{\alpha_i}](\vec{\tau_i}) \to \tau_i \\
\sigma_k' & = & \Gamma(y_k) \\
\Delta_i & = & \Delta, \vec{\alpha_i} \\
\Gamma_i & = & \Gamma, x_i{:}\tau, \overrightarrow{x_i{:}\tau_i} \\
\Gamma_i' & = & \Gamma, x_i{:}\tau', \overrightarrow{x_i{:}\tau_i}
\end{array}
$$

An argument similar to the Field Selection Rule case will be denoted FSR argument. The result follows by transitivity from:

$$
\begin{array}{ll}
 & e \\
\equiv_{\Delta;\Gamma:\tau} & \quad \langle \text{FSR argument} \rangle \\
 & [m_i = M_i; f_1 = |e_1|, f_j = e_j]_{1\le i\le m, 2\le j\le n} \\
\equiv_{\Delta;\Gamma:\tau} & \cdots \quad \langle \text{FSR argument} \rangle \\
 & [m_i = M_i; f_j = |e_j|]_{1\le i\le m, 1\le j\le n} \\
\equiv_{\Delta;\Gamma:\tau} & \quad \langle \text{Lemma 4.2} \rangle \\
 & [m_i = M_i; f_j = |e_j|, g_k = y_k]_{1\le i\le m, 1\le j\le n, 1\le k\le p} \\
\equiv_{\Delta;\Gamma:\tau} & \quad \langle \text{See Below} \rangle \\
 & [m_1 = M_1', m_i = M_i; f_j = |e_j|, g_k = y_k]_{2\le i\le m, 1\le j\le n, 1\le k\le p} \\
\equiv_{\Delta;\Gamma:\tau} & \cdots \quad \langle \text{See Below} \rangle \\
 & [m_i = M_i'; f_j = |e_j|, g_k = y_k]_{1\le i\le m, 1\le j\le n, 1\le k\le p} \\
= & \quad \langle \text{Definition of } |\cdot| \rangle \\
 & |e|
\end{array}
$$

Just need to show that $[m_1 = M_1', \ldots, m_{i-1} = M_{i-1}', m_i = M_i, \ldots, m_n = M_n; f_j = |e_j|, g_k = y_k]_{j\in J, k\in K} \equiv_{\Delta;\Gamma:\tau} [m_1 = M_1', \ldots, m_i = M_i', m_{i+1} = M_{i+1}, \ldots, m_n = M_n; f_j = |e_j|, g_k = y_k]_{j\in J, k\in K}$.

By the typing rule for method bodies, it must be that $\Delta_i; \Gamma_i \vdash b_i : \tau_i$. As argued for type preservation $\Delta \vdash \tau' \le \tau$, so by Context Strengthening $\Delta_i; \Gamma_i' \vdash b_i : \tau_i$. By the induction hypothesis $b_i \equiv_{\Delta_i;\Gamma_i':\tau'} |b_i|$. So by Lemma 4.3 $[m_1 = M_1', \ldots, m_{i-1} = M_{i-1}', m_i = M_i, \ldots, m_n = M_n; f_j = |e_j|, g_k = y_k]_{j\in J, k\in K} \equiv_{\Delta;\Gamma:\tau'} [m_1 = M_1', \ldots, m_i = M_i', m_{i+1} = M_{i+1}, \ldots, m_n = M_n; f_j = |e_j|, g_k = y_k]_{j\in J, k\in K}$. The result follows by OE Refinement.

Method Invocation Rule In this case $e = e'.m[\vec{\sigma}](e_1, \ldots, e_n)$. Arguments similar to the Field Selection Rule case establish each of the following, from which the result follows by transitivity.

$$
\begin{array}{rll}
e & \equiv_{\Delta;\Gamma:\tau} & |e'|.m[\vec{\sigma}](e_1, \ldots, e_n) \\
 & \equiv_{\Delta;\Gamma:\tau} & |e'|.m[\vec{\sigma}](|e_1|, e_2, \ldots, e_n) \\
 & \equiv_{\Delta;\Gamma:\tau} & \cdots \\
 & \equiv_{\Delta;\Gamma:\tau} & |e'|.m[\vec{\sigma}](|e_1|, \ldots, |e_n|) \\
 & = & |e|
\end{array}
$$

Field Selection Rule In this case $e = e'.f$ and $\Delta; \Gamma \vdash e' : \tau'$ where $\tau' = [; f : \tau^\phi]$. By the induction hypothesis $e' \equiv_{\Delta;\Gamma:\tau'} |e'|$ $(*)$. Let $C = \langle\rangle.f$, then $\Delta; \Gamma \vdash C : \tau\langle\epsilon; \epsilon{:}\tau'\rangle$, $e = C\langle e'\rangle$ and $|e| = E\langle|e'|\rangle$. By $(*)$ and OE Congruence $C\langle e'\rangle \equiv_{\Delta;\Gamma:\tau} C\langle|e'|\rangle$ and the result follows.

Field Update Rule In this case, $e = e_1.f \leftarrow e_2$. An argument similar to the Field Selection Rule case establishes $e \equiv_{\Delta;\Gamma:\tau} |e_1|.f \leftarrow e_2$, and a similar argument establishes $|e_1|.f \leftarrow e_2 \equiv_{\Delta;\Gamma:\tau} |e_1|.f \leftarrow |e_2| = |e|$. The result follows by transitivity.

Type Abstraction Rule This case is similar to the Field Selection Rule case.

Type Application Rule This case is similar to the Field Selection Rule case.

$\square$

A couple of points are worth mentioning. First, the proof actually proves full abstraction. Full abstraction is the property that the target language cannot distinguish the translation of equivalent source terms, formally: $e_1 \equiv_{\Delta;\Gamma:\tau} e_2$ implies $|e_1| \equiv_{\Delta;\Gamma:\tau} |e_2|$. This holds for object closure conversion, for if $e_1 \equiv_{\Delta;\Gamma:\tau} e_2$ then $|e_1| \equiv_{\Delta;\Gamma:\tau} e_1 \equiv_{\Delta;\Gamma:\tau} e_2 \equiv_{\Delta;\Gamma:\tau} |e_2|$.

Second, the proof should extend to a number of other features. Consider recursive types. For equirecursive types, the definitions and proofs remain unchanged. For isorecursive types, the definition of closed under reduction needs to include unroll, but otherwise the proof goes through. The proof should also easily extend to cover all of the features mentioned in the next section, thus supporting my claim that the proof really does scale to real languages.

5 Some Extensions

The translation scales to a number of other language features and variations in the language semantics; in some sense, it is a canonical object closure-conversion translation for object calculi without method update or method extension. The translation, as stated, works for the same language but with field update interpreted imperatively rather than applicatively. It also works for the same language but with right-extension subtyping (*i.e.*, methods and fields in an object constructor are ordered, and a subtype has more methods and fields on the right). The translation preserves the right-extension property. As right-extension subtyping can be implemented more efficiently, a source language with both first-class objects and right-extension subtyping has an efficient implementation. If other noncode[4] constructs such as integers, products, sums, and arrays, are added to the language, the translation can be extended to these constructs by adding trivial rules such as $|i| = i$, $|e_1 + e_2| = |e_1| + |e_2|$, $|(e_1, e_2)| = (|e_1|, |e_2|)$, $|e.1| = |e|.1$, et cetera. Recursive types and their roll and unroll coercions could be added with similar trivial rules. This addition would not require serious changes to the proof of correctness, as is the case for the all the proofs described in the introduction. If functions are added to the language, then the translation could be extended with functional closure conversion or with the ideas of Section 6.

[4] A code construct is one that suspends expressions, for example, functions and method definitions. In general code constructs require closure conversion, while noncode constructs do not.

On polymorphism, the translation already includes unbounded parametric polymorphism both as a separate construct and for methods. Extending this to ordinary-bounded, F-bounded, or matching-based parametric polymorphism is straightforward. It is worth noting that in implementing the object language, after converting objects to records of functions, and before lifting these functions to the top level, the free type variables must be closed over using the ideas of Morrisett *et al.* [MWCG98]. If a type-passing interpretation is desired, the translation must be changed to close over type variables as well. There are two approaches: The first uses the ideas of Crary, Weirich, and Morrisett [CWM98] to convert type variables to value variables and a type-erasure interpretation. This paper's translation would then close over the value variables. The other approach involves adding type fields to an object constructor to store the free type variables. This would require a type system for objects with type fields, which likely would involve the complexity described by Minamide *et al.* [MMH96], Leroy [Ler94], and Harper and Lillibridge [HL94]. Otherwise, I believe the translation would look much the same, although the proof would be considerably more complicated.

Next, consider object calculi with method update and method extension. A method update operation $e.m \leftarrow M$ changes (either imperatively or applicatively, depending on the desired semantics) object e's response to method m to be M. The translation of this expression needs to close over the closure variables of M. To achieve this e must be extended with new fields to store the free variables, so a field-extension operation is needed. Consider a language with all of these operations:

$$e ::= \ldots \mid e.m \leftarrow M \mid e_1 + f = e_2 \mid e + m = M$$

The expression $e_1 + f = e_2$ is the extension of object e_1 with a new field f with initial value e_2; similarly, $e + m = M$ is the extension of e with a new method m with definition M. The intended operational semantics of the extended language is the dictionary semantics of Riecke and Stone [RS98].

The translation extends as follows:

$$\begin{array}{lcl}
|e_1 + f = e_2| & = & |e_1| + f = |e_2| \\
|e.m \leftarrow M| & = & (|e| +_{1 \leq i \leq n} g_i = y_i).m \leftarrow M' \\
|e + m = M| & = & (|e| +_{1 \leq i \leq n} g_i = y_i) + m \leftarrow M' \\
\quad \text{where } M & = & x[\vec{\alpha}](\overrightarrow{x{:}\tau}).b{:}\tau \\
\qquad\quad\;\; M' & = & x[\vec{\alpha}](\overrightarrow{x{:}\tau}).|b|\{y_1, \ldots, y_n := x.g_1, \ldots, x.g_n\}{:}\tau \\
\qquad\quad\;\; \mathrm{cv}(M) & = & \{y_1, \ldots, y_n\} \\
\qquad\quad\;\; g_1, \ldots, g_n \text{ are fresh} & &
\end{array}$$

Finally, as well as extensions to other language features, the translation has variants that express other environment representations. The translation of method bodies is $b_i' = |b_i|\{\overrightarrow{y := x_i.g}\}$. It could also be $b_i' = |b_i\{\overrightarrow{y := x_i.g}\}|$, which would result in a different environment representation. The first choice leaves y_1 through y_p free in all inner-nested object constructors and so these variables will be closed over in the inner objects, resulting in a flat environment representation. The second choice replaces the y_p in an inner-nested object constructor with a (free) reference to x_i which will be closed over, resulting in a linked environment representation.

$$
\begin{array}{lcl}
|x| & = & x \\
|i| & = & i \\
|[m_i = M_i; f_j = e_j]_{i \in I, j \in J}| & = & \langle m_i = |M_i|, f_j = |e_j| \rangle_{i \in I, j \in J} \\
|e.m[\vec{\tau}](\vec{e})| & = & \mathsf{let}\ x = |e|\ \mathsf{in}\ x.m(x, \overrightarrow{|e|}) \\
|e.f| & = & |e|.f \\
|e_1.f \leftarrow e_2| & = & |e_1|.f \leftarrow |e_2| \\
|\Lambda\alpha.b| & = & |b| \\
|e[\sigma]| & = & |e| \\
|x[\vec{\alpha}](\overrightarrow{x{:}\tau}){:}\tau.b| & = & \lambda(x, \vec{x}).|b|
\end{array}
$$

Figure 4: Untyped Self Application Semantics Translation

6 Closures and Functional Closure Conversion

We are finally in a position to compare object closure conversion and functional closure conversion, and to formalise the well known connection between closures and single-method objects. This section will show this connection by providing a translation from a typed lambda calculus to the language of this paper, and show that the encoding composed with object closure conversion and some object encoding results in a functional closure conversion. I will demonstrate this for two particular object encodings, showing their equivalence to a closure passing and environment passing style of functional closure conversion.

Consider a simply-typed lambda calculus with recursive functions:

$$
\begin{array}{llcl}
\text{Types} & \tau, \sigma & ::= & \mathsf{int} \mid \tau_1 \to \tau_2 \\
\text{Expressions} & e, b & ::= & x \mid i \mid \mathsf{fix}\ f(x_1{:}\tau_1){:}\tau_2.b \mid e_1\ e_2
\end{array}
$$

For the remainder of this section, assume an object language with integers. The lambda calculus can be encoded into the object language as follows:

$$
\begin{array}{lcl}
|\mathsf{int}| & = & \mathsf{int} \\
|\tau_1 \to \tau_2| & = & [\mathsf{apply}{:}[\,](|\tau_1|) \to |\tau_2|;\,] \\
|x| & = & x \\
|i| & = & i \\
|\mathsf{fix}\ f(x_1{:}\tau_1){:}\tau_2.b| & = & [\mathsf{apply} = f[\,](x_1{:}|\tau_1|).|b|{:}|\tau_2|;\,] \\
|e_1\ e_2| & = & |e_1|.\mathsf{apply}[\,](|e_2|)
\end{array}
$$

If the object closure conversion translation is composed with the above encoding, the combined rule for functions is:

$$
\begin{array}{l}
|\mathsf{fix}\ f(x_1{:}\tau_1){:}\tau_2.b| \ = \ [\mathsf{apply} = f[\,](x_1{:}|\tau_1|).|b|\{y_i := f.g_i\}; g_i = y_i] \\
\qquad \text{where } \mathrm{fv}(b) - \{f, x_1\} = \{y_i\}
\end{array}
$$

Now, consider converting the object calculi back into a functional calculi via an object encoding. The notation $\langle \overrightarrow{f = e} \rangle$ denotes a record with fields $\vec{f}$ and values $\vec{e}$, $e.f$ denotes field projection,

$$
\begin{array}{lcl}
|\alpha| & = & \alpha \\
|[m_i{:}s_i;]_{i\in I}| & = & \exists\alpha.\langle\langle m_i = |s_i|(\alpha)\rangle^+_{i\in I}, \alpha^+\rangle \\
|\forall\alpha.\tau| & = & \forall\alpha.|\tau| \\
|[\vec{\alpha}](\vec{\tau}) \to \tau|(\sigma) & = & \forall[\vec{\alpha}](\sigma, \vec{|\tau|}) \to |\tau| \\
 & & \\
|x| & = & x \\
|i| & = & i \\
|[m_i = M_i; f_j = e_j]_{i\in I, j\in J}{:}\tau| & = & \mathsf{letrec}\ m_i = |M_i|(\tau, \tau_e)\ \mathsf{in} \\
 & & \mathsf{pack}\ [\tau_e, \langle\langle m_i = m_i\rangle_{i\in I}, \langle f_j = |e_j|\rangle_{j\in J}\rangle]\ \mathsf{as}\ |\tau| \\
 & & \text{where}\ \ \tau \ = \ [m_i{:}s_i; f_j{:}\sigma_j^{\phi_j}]_{i\in I, j\in J} \\
 & & \qquad\quad \tau_e \ = \ \langle f_j{:}|\sigma_j|^{\phi_j}\rangle_{j\in J} \\
|e.m[\vec{\tau}](\vec{e})| & = & \mathsf{unpack}\ [\alpha, x] = |e|\ \mathsf{in}\ x.1.m[\vec{|\tau|}](x.2, \vec{|e|}) \\
|x.f| & = & x_{\mathsf{e}}.f \\
|x.f \leftarrow e_2| & = & \mathsf{pack}\ [\tau_e, \langle\langle m_i = m_i\rangle_{i\in I}, x_{\mathsf{e}}.f \leftarrow |e_2|\rangle]\ \mathsf{as}\ |\tau| \\
|\Lambda\alpha.b| & = & \Lambda\alpha.|b| \\
|e[\sigma]| & = & |e|[|\sigma|] \\
|x[\vec{\alpha}](\overrightarrow{x{:}\tau}){:}\tau.b|(\tau, \tau_e) & = & \lambda[\vec{\alpha}](x_{\mathsf{e}}{:}\tau_e, \overrightarrow{x{:}|\tau|}){:}|\tau|. \\
 & & \mathsf{let}\ x = \mathsf{pack}\ [\tau_e, \langle\langle m_i = m_i\rangle_{i\in I}, x_{\mathsf{e}}\rangle]\ \mathsf{as}\ |\tau|\ \mathsf{in} \\
 & & |b|
\end{array}
$$

Where x maps 1-1 to x_{e} and all x_{e} are unnamable in the source calculus.

Figure 5: Typed Pierce and Turner Translation

and $e_1.f \leftarrow e_2$ denotes field update. First, consider an encoding based on the self-application semantics [Kam88]. An untyped version of this translation appears in Figure 4. Ignoring typing, the interesting composed translation rules are:

$$
\begin{array}{lcl}
|\mathsf{fix}\ f(x_1{:}\tau_1){:}\tau_2.b| & = & \langle \mathsf{apply} = \lambda(f, x_1).|b|\{y_i := f.g_i\}, g_i = y_i\rangle \\
 & & \qquad \text{where } \mathrm{fv}(b) - \{f, x_1\} = \{y_i\} \\
|e_1\ e_2| & = & \mathsf{let}\ x = |e_1|\ \mathsf{in}\ x.\mathsf{apply}(x, |e_2|)
\end{array}
$$

These rules are just functional closure conversion with a closure-passing style and a flat environment representation.

Second, consider Pierce and Turner's object encoding [PT94]. A typed version of this translation appears in Figure 5, assuming that methods are public and fields are private. Ignoring typing, the interesting composed translation rules are:

$$
\begin{array}{lcl}
|\mathsf{fix}\ f(x_1{:}\tau_1){:}\tau_2.b| & = & \mathsf{let\ fix}\ \mathit{fcode}(\mathit{fenv}, x_1) = \\
 & & \quad \mathsf{let}\ f = \langle\langle \mathsf{apply} = \mathit{fcode}\rangle, \mathit{fenv}\rangle\ \mathsf{in}\ |b|\{y_i := \mathit{fenv}.g_i\}\ \mathsf{in} \\
 & & \langle\langle \mathsf{apply} = \mathit{fcode}\rangle, \langle g_i = y_i\rangle\rangle \\
 & & \qquad \text{where } \mathrm{fv}(b) - \{f, x_1\} = \{y_i\} \\
|e_1\ e_2| & = & \mathsf{let}\ x = |e_1|\ \mathsf{in}\ x.1.\mathsf{apply}(x.2, |e_2|)
\end{array}
$$

These rules are just functional closure conversion with an environment-passing style and a flat environment representation.

Thus we see that closures and single method objects are equivalent, as witnessed by the translation given at the beginning of this section. We also see that functional closure conversion factors into this equivalence translation, object closure conversion, and an object encoding.

Other schemes mentioned in Morrisett and Harper [MH99] could also be described as specialised object encodings for single-method objects composed with object closure conversion. In fact, the ideas of this section suggest a general framework for explaining functional closure conversion, as follows. First, a closure language would be defined, which would be the object language of this paper restricted to single methods. Second, a closure-conversion translation would be defined on this language, and various choices for environment representation would be described as variations on this translation. Third, a closure-representation translation would be presented, and various passing styles would be described as various closure representations. This general framework then allows Java's inner-class transformation to be explained as a generalisation to a full object language.

7 Related Work and Summary

Closure conversion has been studied extensively for functions [AJ89, Han95, KKR$^+$86, Lan64, MMH96, MWCG98, Rey72, Ste78, SW96]. To the best of my knowledge, object closure conversion has not been described before.

The connection between closures and objects is well known, and has been hinted at in the literature. As far as I am aware, this is the first paper to formalise the connection explicitly. Reddy [Red98] discuss objects as closures. Minamide *et al.* [MMH96] informally discuss closures as objects, but do not formalise the connection. Abadi and Cardelli [AC96] give several versions of an encoding of functional calculi into object calculi. Their encoding is different from the one here as they do not consider method parameters. Also they do not talk at all about closure conversion nor make connections between functional calculi and object calculi at the level of closures.

Correctness proofs for functional closure conversion are given by Minamide *et al.* [MMH95], Stekler and Wand [SW96], Morrisett and Harper [MH99], and possibly other authors. Only the latter considers recursive functions, and none of them consider recursive types. Observational equivalence has been studied and used to prove correctness in both functional settings (*e.g.*, [Mor68, Plo77, MST96]) and in object settings (*e.g.*, [GHL98]). My results use similar tools and similar proof techniques, further supporting the claim that my proof will scale to real languages.

This paper has presented an object language with first-class objects and a closure-conversion translation for it. This translation was used to show a formal connection between closures and single-method objects, and that functional closure conversion factors through object closure conversion. These results lend further insight into the general problem of closure conversion and foundations for the implementation of object-oriented languages.

References

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

[AJ89] Andrew Appel and Trevor Jim. Continuation-passing, closure-passing style. In *16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 293–302, Austin, Texas, USA, January 1989.

[CWM98] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *1998 ACM SIGPLAN International Conference on Functional Programming*, pages 301–312, Baltimore Maryland, USA, September 1998.

[GHL98] Andrew Gordon, Paul Hankin, and Søren Lassen. Compilation and equivalence of imperative objects. Technical Report RS-98-55, BRICS, Department of Computer Science, University of Aarhus, Ny Munkegade, building 540, DK-8000 Aarhus C, Denmark, December 1998. URL: `http://www.brics.dk/RS/98/55`.

[Han95] J. Hannan. A type system for closure conversion. In *The Workshop on Types for Program Analysis*, 1995.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137, Portland Oregon, USA, January 1994.

[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.

[KKR+86] David Kranz, R. Kelsey, J. Rees, P. R. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233, June 1986.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 109–122, Portland, OR, USA, January 1994.

[MH99] Greg Morrisett and Robert Harper. Simply-typed closure conversion. Unpublished, authors contact: `jgm@cs.cornell.edu`, April 1999.

[MMH95] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. Technical Report CMU-CS-95-171, Carnegie Mellon University, Pittsburgh, PA 15213, July 1995.

[MMH96] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, January 1996.

[Mor68] J. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, Massachuetts Institute of Technology, 1968.

[MST96] Ian Mason, Scott Smith, and Carolyn Talcott. From operational semantics to domain theory. *Information and Computation*, 128:26–47, 1996.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego California, USA, January 1998. ACM Press.

[Plo77] Gordon Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[PT94] Benjamin Pierce and David Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[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.

[Rey72] John Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the Annual ACM Conference*, pages 717–740, 1972.

[RS98] Jon Riecke and Christopher Stone. Privacy via subsumption. In *5th International Workshop on Foundations of Object Oriented Programming Languages*, San Diego, CA, USA, January 1998.

[Ste78] Guy Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, MIT, 1978.

[SW96] Paul Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Progamming Languages and Systems*, pages 48–86, January 1996.

A Unwinding

Consider an extension of the object language where object constructors have the form:

$$[m_i = M_i; f_j = e_j]^n_{i \in I, j \in J} \quad n \in \mathbb{N} \cup \{\omega\}$$

The n is called an unwinding number, and limits the number of recursive invocations of a method on the object. The old language can be mapped to the new one by decorating all object constructors with ω. The new language can be mapped to the old one by erasing the unwinding numbers. It is easy to verify that these mappings form an embedding-projection pair.

The operational semantics for the new language consists of the old rules with the rule for method invocation replaced by:

$$\begin{array}{rcl} v^n.m[\vec{\tau}](\vec{v}) & \mapsto & b\{\vec{\alpha}, x, \vec{x} := \vec{\tau}, v^{n-1}, \vec{v}\} \quad n > 0 \\ v^0.m[\vec{\tau}](\vec{v}) & \mapsto & v^0.m[\vec{\tau}](\vec{v}) \end{array}$$

Where $\omega - 1 = \omega$. It is easy to verify that the embedding (mentioned above) commutes with reduction. Thus an expression is equivalent in behaviour to its embedding.

If $e_1 \mapsto e_2$ then by inspection of the reduction rules, observe that each unwinding number in e_2 corresponds to one of the unwinding numbers of e_1, and is equal to or 1 less than that unwinding number. In the theorems below we will be interested in some set of a term's unwinding numbers and will write $e[\cdot]$ to denote a term e with some set of unwinding numbers distinguished. Then $e[n]$ will be e with the distinguished unwinding numbers replaced by n, $e[\geq n]$ will mean the distinguished unwinding numbers are all at least n, and if $e_1 \mapsto e_2$ then $e_2[\cdot]$ will be e_2 with the unwinding numbers that correspond to e_1's distinguished unwinding numbers distinguished.

Lemma A.1 *If $e_1[\geq n]$, $n \geq 1$, and $e_1 \mapsto e_2$ then $e_2[\geq n-1]$.*

Proof: By inspection of the reduction rules, no unwinding numbers are introduced, and no unwinding numbers are changed except by the method invocation rule, which reduces one unwinding number by 1. □

Lemma A.2 *If $e[\omega]\!\downarrow^n$ then $e[n]\!\downarrow$. If $e[n]\!\downarrow$ then $e[\omega]\!\downarrow$.*

Proof: Observe that $e[\omega]$ and $e[n]$ differ only in that the former has unwinding numbers ω is some places and the latter has n instead. Therefore, if $n \geq 1$ the same reduction rules apply to both, and if $e[\omega] \mapsto e'[\omega]$ then $e[n] \mapsto e'[\geq n-1]$. Let the reduction sequence for $e[\omega]\!\downarrow^n$ be $e_1[\omega] \mapsto \cdots \mapsto e_n[\omega]$. Then by induction and Lemma A.1, $e_1[\geq n] \mapsto \cdots \mapsto e_n[\geq 0]$. Note that changing unwinding numbers in a value still makes it a value, so $e[n]\!\downarrow$.

If the second new rule mentioned above is ever used in a reduction sequence, that rule is used for the rest of the reduction sequence, and an infinite loop results. Therefore if $e[n]\!\downarrow$ that rule is never used. By similar reasoning to above, the same rules apply to $e[n]$ as do to $e[\omega]$. So if the reduction sequence for $e[n]\!\downarrow$ is $e_1[n_1] \mapsto \cdots \mapsto e_m[n_m]$ then $e[\omega] = e_1[\omega] \mapsto \cdots \mapsto e_m[\omega]$, that is, $e[\omega]\!\downarrow$. □

Lemma A.3 *If $\forall n \in \mathbb{N} : e_1[n] \equiv_{\Delta;\epsilon:\tau} e_2[n]$ then $e_1[\omega] \equiv_{\Delta;\epsilon:\tau} e_2[\omega]$.*

Proof: Since the typing rules ignore unwinding numbers, the typing requirements are easy to establish. Let C be such that $\epsilon;\epsilon \vdash C : [;]\langle\Delta;\epsilon{:}\tau\rangle$, need to show that $C\langle e_1[\omega]\rangle \sim C\langle e_2[\omega]\rangle$. Assume $C\langle e_1[\omega]\rangle\!\uparrow$, then by Lemma A.2 $C\langle e_1[n]\rangle\!\uparrow$ for all n. By assumption $C\langle e_2[n]\rangle\!\uparrow$ for all n, so by Lemma A.2 $C\langle e_2[\omega]\rangle\!\uparrow$. Assume $C\langle e_1[\omega]\rangle\!\downarrow$, then it must do so in some number of steps, say n. Then by Lemma A.2 $C\langle e_1[n]\rangle\!\downarrow$, so by assumption $C\langle e_2[n]\rangle\!\downarrow$. By Lemma A.2 $C\langle e_2[\omega]\rangle\!\downarrow$. □

B Observational Equivalence

This appendix contains some basic theory on observational equivalence. It includes some basic results: observational equivalence is an equivalence and is a congruence, observational equivalence at a subtype is a refinement of observational equivalence at a supertype, and observational equivalence is preserved under type substitutions and substitutions of convergent observationally equivalent terms. It also shows several characterisations of observational equivalence: two terms are observationally equivalent if they have the same termination behaviour and values they evaluate to are observationally equivalent, two terms are observationally equivalent if they have the same termination behaviour under all closed instances and uses (see [MST96]), and two terms are observationally equivalent if they are observationally equivalent under all appropriate substitutions. Finally, I show a coinduction principle for concluding that two terms are observationally equivalent by constructing a set that is closed under reductions (defined later). The results are stated generally, but several of the proofs assume that reduction is deterministic, which it is for the object language. The results can be proven for nondeterministic reduction systems, but the proofs are more complicated.

B.1 Basics

Definition B.1

- *A type substitution* ϱ *is a mapping from type variables to types.*
- $\Delta_1 \vdash \varrho : \Delta_2$ *if and only if* $\mathrm{dom}(\varrho) = \Delta_2$ *and* $\forall \alpha \in \Delta_2 : \Delta_1 \vdash \varrho(\alpha)$*.*

Definition B.2

- *A (term) substitution* ρ *is a mapping from term variables to expressions. A value substitution is a mapping from term variables to values.*
- $\Delta; \Gamma_1 \vdash \rho : \Gamma_2$ *if and only if* $\mathrm{dom}(\rho) = \mathrm{dom}(\Gamma_2)$ *and* $\forall x \in \mathrm{dom}(\Gamma_2) : \Delta; \Gamma_1 \vdash \rho(x) : \Gamma_2(x)$*.*

Definition B.3 $e_1 \equiv^{\mathsf{CIU}}_{\Delta;\Gamma:\tau} e_2$ *if and only if* $\Delta; \Gamma \vdash e_1 : \tau$, $\Delta; \Gamma \vdash e_2 : \tau$, *and for all type substitutions* ϱ*, value substitutions* ρ*, and* E *such that* $\epsilon \vdash \varrho : \Delta$, $\epsilon; \epsilon \vdash \rho : \Gamma\{\varrho\}$*, and* $\epsilon; \epsilon \vdash E : [;]\langle \epsilon; \epsilon{:}\tau\{\varrho\}\rangle$, $E\langle e_1\{\varrho\}\{\rho\}\rangle \sim E\langle e_2\{\varrho\}\{\rho\}\rangle$*.*

Lemma B.1 $\sim$ *is an equivalence relation on* $\{e | \epsilon; \epsilon \vdash e : [;]\}$*.* $\equiv_{\Delta;\Gamma:\tau}$ *is an equivalence relation on* $\{e | \Delta; \Gamma \vdash e : \tau\}$*.* $\equiv^{\mathsf{CIU}}_{\Delta;\Gamma:\tau}$ *is an equivalence relation on* $\{e | \Delta; \Gamma \vdash e : \tau\}$*.*

Proof: The latter results follow from the first result. The first result follows from the reflexive, symmetric, and transitive nature of $e_1\uparrow \Leftrightarrow e_2\uparrow$ and $e_1\downarrow \Leftrightarrow e_2\downarrow$. □

Lemma B.2 (OE Congruence) *If* $e_1 \equiv_{\Delta_1,\Delta_2;\Gamma_1,\Gamma_2:\tau_2} e_2$ *and* $\Delta_1; \Gamma_1 \vdash C : \tau_1\langle \Delta_2; \Gamma_2{:}\tau_2\rangle$ *then* $C\langle e_1\rangle \equiv_{\Delta_1;\Gamma_1:\tau_1} C\langle e_2\rangle$*.*

Proof: Let $\epsilon; \epsilon \vdash C' : [;]\langle \Delta_1; \Gamma_1{:}\tau_1\rangle$. By replacing the use of the hole rule in the derivation for C' with the derivation for C, is easy to establish that $\epsilon; \epsilon \vdash C'\langle C\rangle : [;]\langle \Delta_1, \Delta_2; \Gamma_1, \Gamma_2{:}\tau_2\rangle$ thus $C'\langle C\rangle\langle e_1\rangle \sim C'\langle C\rangle\langle e_2\rangle$. The result follows since $C'\langle C\rangle\langle e_i\rangle = C'\langle C\langle e_i\rangle\rangle$. □

Lemma B.3 (OE Refinement) *If* $\Delta \vdash \tau_1 \leq \tau_2$ *and* $e_1 \equiv_{\Delta;\Gamma:\tau_1} e_2$ *then* $e_1 \equiv_{\Delta;\Gamma:\tau_2} e_2$*.*

Proof: Let $\Delta \vdash \tau_1 \leq \tau_2$ (1) and $e_1 \equiv_{\Delta;\Gamma:\tau_1} e_2$ (2). By (2) $\Delta;\Gamma \vdash e_i : \tau_1$ so by (1) and subsumption $\Delta;\Gamma \vdash e_i : \tau_2$. Let C be such that $\epsilon;\epsilon \vdash C : [;]\langle\Delta;\Gamma{:}\tau_2\rangle$. By modifying the derivation of $\epsilon;\epsilon \vdash C : [;]\langle\Delta;\Gamma{:}\tau_2\rangle$ to use the subsumption rule with (1) after the rule for holes, we can establish that $\epsilon;\epsilon \vdash C : [;]\langle\Delta;\Gamma{:}\tau_1\rangle$. By (2) $C\langle e_1\rangle \sim C\langle e_2\rangle$ as required. $\Box$

Lemma B.4 *Let $O(b) = [m = x[\vec{\alpha}](\overline{x{:}\vec{\tau}}){:}\tau'.b, m_i = M_i; f_j = e_j]_{i\in I, j\in J}$ be such that $\epsilon;\epsilon \vdash O(b) : \tau$ if $\vec{\alpha};\Gamma \vdash b : \tau'$ where $\Gamma = x{:}\tau, \overline{x{:}\vec{\tau}}$. Then, if $b_1 \equiv^{\mathsf{CIU}}_{\vec{\alpha};\Gamma:\tau'} b_2$ then $O(b_1) \equiv^{\mathsf{CIU}}_{\epsilon;\epsilon:\tau} O(b_2)$.*

Proof: (Assume deterministic reduction.)

Let E be such that $\epsilon;\epsilon \vdash E : [;]\langle\epsilon;\epsilon{:}\tau\rangle$ (the substitutions do not matter in the empty contexts). Need to show that $E\langle O(b_1)\rangle \sim E\langle O(b_2)\rangle$. If both $E\langle O(b_1)\rangle$ and $E\langle O(b_2)\rangle$ diverge then the result follows. Otherwise, without loss of generality assume that $E\langle O(b_1)\rangle\!\downarrow$. I will show by induction on computation length that $e\{z := O(b_1)\}\!\downarrow$ implies $e\{y := O(b_2)\}\!\downarrow$ where $\epsilon; y{:}\tau \vdash e : [[];]$ ($E\langle y\rangle$ is such an e). If $e\{z := O(b)\} \mapsto e'\{z := O(b)\}$ for all appropriate b then the result follows from the induction hypothesis. Otherwise, by inspection of the reduction rules there exists E', m, $\vec{\tau}$, and $\vec{v}$ (actually E' and $\vec{v}$ may have z where values are normally required such that $E'\{z := v\}$ is an evaluation context and similarly for $\vec{v}$) such that for all appropriate b, $e = E\langle z.m[\vec{\tau}](\overrightarrow{(v)})\rangle$ and $e\{z := O(b)\} \mapsto E'\{z := O(b)\}\langle b\{\vec{\alpha}, x, \vec{x} := \vec{\tau}, O(b), \overrightarrow{v\{z := O(b)\}}\}\rangle$. By the induction hypothesis $E'\{z := O(b_2)\}\langle b_1\{\vec{\alpha}, x, \vec{x} := \vec{\tau}, O(b_2), \overrightarrow{v\{z := O(b_2)\}}\}\rangle\!\downarrow$. Then by assumption $E'\{z := O(b_2)\}\langle b_2\{\vec{\alpha}, x, \vec{x} := \vec{\tau}, O(b_2), \overrightarrow{v\{z := O(b_2)\}}\}\rangle\!\downarrow$ as required. $\Box$

Lemma B.5 *$e_1 \equiv^{\mathsf{CIU}}_{\Delta;\Gamma:\tau} e_2$ if and only if for all ϱ and ρ such that $\epsilon \vdash \varrho : \Delta$ and $\epsilon \vdash \rho : \Gamma\{\rho\}$, $e_1\{\varrho\}\{\rho\} \equiv^{\mathsf{CIU}}_{\epsilon;\epsilon:\tau\{\varrho\}} e_2\{\varrho\}\{\rho\}$.*

Proof: By the definitions and the observation that substitutions are irrelevant for empty contexts. $\Box$

Lemma B.6 (CIU Congruence) *If $e_1 \equiv^{\mathsf{CIU}}_{\Delta_1,\Delta_2;\Gamma_1,\Gamma_2:\tau_2} e_2$ and $\Delta_1;\Gamma_1 \vdash C : \tau_1\langle\Delta_2;\Gamma_2{:}\tau_2\rangle$ then $C\langle e_1\rangle \equiv^{\mathsf{CIU}}_{\Delta_1;\Gamma_1:\tau_1} C\langle e_2\rangle$.*

Proof: (Assume deterministic reduction.) By Lemma B.5 it suffices to show the result for $\Delta_1 = \epsilon$ and $\Gamma_1 = \epsilon$.

The proof produces by induction on the structure of C. If $C = \langle\rangle$ then the result follows by assumption. If $C = x$ then the result follows by reflexivity. Next consider $C = C_1.f \leftarrow C_2$. The result follows by transitivity from $C_1\langle e_1\rangle.f \leftarrow C_2\langle e_1\rangle \equiv^{\mathsf{CIU}}_{\epsilon;\epsilon:\tau_1} C_2\langle e_1\rangle.f \leftarrow C_2\langle e_2\rangle$ (1) and $C_2\langle e_1\rangle.f \leftarrow C_2\langle e_2\rangle \equiv^{\mathsf{CIU}}_{\epsilon;\epsilon:\tau_1} C_2\langle e_2\rangle.f \leftarrow C_2\langle e_2\rangle$ (2). By the induction hypothesis $C_1\langle e_1\rangle \equiv^{\mathsf{CIU}}_{\epsilon;\epsilon:\tau} C_2\langle e_2\rangle$ (3). Let E be such that $\epsilon;\epsilon \vdash E : [;]\langle\epsilon;\epsilon{:}\tau\rangle$. Let $E' = E\langle\langle\rangle.f \leftarrow C_2\langle e_1\rangle\rangle$. Then $\epsilon;\epsilon \vdash E' : [;]\langle\epsilon;\epsilon{:}\tau\rangle$, so by (3) $E'\langle e_1\rangle \sim E'\langle e_2\rangle$. Since $E'\langle e_i\rangle = E\langle C_1\langle e_i\rangle.f \leftarrow C_2\langle e_1\rangle\rangle$ (1) follows. For (2), by the induction hypothesis $C_2\langle e_1\rangle \equiv^{\mathsf{CIU}}_{\epsilon;\epsilon:\tau'} C_2\langle e_2\rangle$ (4) for some τ'. Let E be such that $\epsilon;\epsilon \vdash E : [;]\langle\epsilon;\epsilon{:}\tau\rangle$. Let $e'_i = E\langle C_1\langle e_2\rangle.f \leftarrow C_2\langle e_i\rangle\rangle$. If both e'_1 and e'_2 diverge the result follows. Otherwise without loss of generality assume $e'_1\!\downarrow$. Then by induction on the length of $e'_1 \mapsto^* v$, I will show $e'_2\!\downarrow$. If $C_1\langle e_2\rangle \mapsto e'$ then $e'_i \mapsto E\langle e'.f \leftarrow C_2\langle e_i\rangle\rangle$ and the result follows by the induction hypothesis. Otherwise $C_1\langle e_2\rangle$ is a value, so let $E' = E\langle C_1\langle e_2\rangle.f \leftarrow \langle\rangle\rangle$ and the result follows by (4) in a similar manner as argued above. The cases for C a method invocation, field selection, type abstraction, or type application are similar. For C an object constructor, a similar argument shows that the field initialisers can be replaced, then the result follows by multiple applications of Lemma B.4. $\Box$

Lemma B.7 (CIU) *$e_1 \equiv_{\Delta;\Gamma:\tau} e_2$ if and only if $e_1 \equiv^{\mathsf{CIU}}_{\Delta;\Gamma:\tau} e_2$.*

Proof: ($\Rightarrow$) Assume $e_1 \equiv_{\Delta;\Gamma:\tau} e_2$. Let ϱ, ρ, and E be such that $\epsilon \vdash \varrho : \Delta$, $\epsilon;\epsilon \vdash \rho : \Gamma\{\varrho\}$, and $\epsilon;\epsilon \vdash E : [;]\langle\epsilon;\epsilon{:}\tau\{\varrho\}\rangle$. Let $C = E\langle[\mathsf{apply} = x[\Delta](\Gamma){:}\tau.\langle\rangle;].\mathsf{apply}[\varrho(\Delta)](\rho(\mathrm{dom}(\Gamma)))\rangle$. Clearly, $\epsilon;\epsilon \vdash C : [;]\langle\Delta;\Gamma{:}\tau\rangle$, and $C\langle e_i\rangle \mapsto E\langle e_i\{\varrho\}\{\rho\}\rangle$. By assumption $C\{e_1\} \sim C\{e_2\}$ so $E\langle e_1\{\varrho\}\{\rho\}\rangle \sim E\langle e_2\{\varrho\}\{\rho\}\rangle$ as required.

($\Leftarrow$) Assume $e_1 \equiv^{\mathsf{CIU}}_{\Delta;\Gamma:\tau} e_2$. Let C be such that $\epsilon;\epsilon \vdash C : [;]\langle\Delta;\Gamma{:}\tau\rangle$. Then by CIU Congruence $C\langle e_1\rangle \equiv^{\mathsf{CIU}}_{\epsilon;\epsilon:[;]} C\langle e_2\rangle$. Thus for $\varrho = \epsilon$, $\rho = \epsilon$, and $E = \langle\rangle$, $E\langle C\langle e_1\rangle\{\varrho\}\{\rho\}\rangle \sim E\langle C\langle e_2\rangle\{\varrho\}\{\rho\}\rangle$, that is, $C\langle e_1\rangle \sim C\langle e_2\rangle$ as required. $\square$

Lemma B.8 *If r is a sequence of expressions all of type τ in context $\Delta;\epsilon$, $\epsilon;\epsilon \vdash E : [;]\langle\Delta;\epsilon{:}\tau\rangle$, and $\epsilon \vdash \varrho : \Delta$ then: $E\langle r\rangle$ is a reduction sequence if and only if r is a reduction sequence if and only if $r\{\varrho\}$ is a reduction sequence.*

Proof: By inspection of the reduction rules. $\square$

Lemma B.9 *If all reduction sequences for e_1 go through e_2 and $\Delta;\epsilon \vdash e_1 : \tau$ then $e_1 \equiv_{\Delta;\epsilon:\tau} e_2$.*

Proof: By type soundness $\Delta;\epsilon \vdash e_2 : \tau$. By CIU and Lemma B.5 if suffices to show the result for $\Delta = \epsilon$. Let E be such that $\epsilon;\epsilon \vdash E : [;]\langle\epsilon;\epsilon{:}\tau\rangle$ then need to show that $E\langle e_1\rangle \sim E\langle e_2\rangle$. By assumption and Lemma B.8 all reduction sequences of $E\langle e_1\rangle$ go through $E\langle e_2\rangle$. $\square$

Lemma B.10 *If $\Delta;\Gamma \vdash e_1 : \tau$ and $\Delta;\Gamma \vdash e_2 : \tau$ then: $e_1 \equiv_{\Delta;\Gamma:\tau} e_2$ if and only if for all value substitutions ρ such that $\Delta \vdash \rho : \Gamma$:*

- $e_1\{\rho\}\uparrow \Leftrightarrow e_2\{\rho\}\uparrow$
- $e_1\{\rho\} \mapsto^* v_1$ *implies* $\exists v_2 : e_2\{\rho\} \mapsto^* v_2 \wedge v_1 \equiv_{\Delta;\epsilon:\tau} v_2$
- $e_2\{\rho\} \mapsto^* v_2$ *implies* $\exists v_1 : e_1\{\rho\} \mapsto^* v_1 \wedge v_1 \equiv_{\Delta;\epsilon:\tau} v_2$

Proof: By CIU and Lemma B.5 twice it remains to show that for all value substitutions ρ such that $\Delta;\epsilon \vdash \rho : \Gamma$, $e_1\{\rho_1\} \equiv_{\Delta;\epsilon:\tau} e_2\{\rho\}$ if and only if the right hand side. Let ρ be such that $\Delta;\epsilon \vdash \rho : \Gamma$, $e_1' = e_1\{\rho\}$, and $e_2' = e_2\{\rho\}$.

(Assume reduction is deterministic.)

($\Rightarrow$) Let $e_1' \equiv_{\Delta;\epsilon:\tau} e_2'$ (1). Consider $C = [;1 = [\mathsf{apply} = x[\Delta]():\tau.\langle\rangle;].\mathsf{apply}[\vec{[;]}]()]$ then $e_1\uparrow \Leftrightarrow C\langle e_1\rangle\uparrow$ and $e_2\uparrow \Leftrightarrow C\langle e_2\rangle\uparrow$. It is easy to establish that $\epsilon;\epsilon \vdash C : [;]\langle\Delta;\epsilon{:}\tau\rangle$, so by (1) $C\{e_1'\}\uparrow \Leftrightarrow C\{e_2'\}\uparrow$. Now, let $e_1' \mapsto^* v_1$. Since reduction is deterministic $e_2' \mapsto^* v_2$ for some v_2, so it remains to show that $v_1 \equiv_{\Delta;\epsilon:\tau} v_2$. By Lemma B.9 $e_i' \equiv_{\Delta;\epsilon:\tau} v_i$, the result follows by transitivity and (1). The argument for $e_2' \mapsto^* v_2$ is symmetric.

($\Leftarrow$) Since reduction is deterministic and by the first part of the assumption, either both e_1' and e_2' converge or both diverge. In the former case $e_1' \mapsto^* v_1$, $e_2' \mapsto^* v_2$, and $v_1 \equiv_{\Delta;\epsilon:\tau} v_2$ by either of the other two parts. By Lemma B.9 $e_i' \equiv_{\Delta;\epsilon:\tau} v_i$, so the result follows by transitivity. In the latter case, by CIU let ϱ and E be such that $\epsilon \vdash \varrho : \Delta$ and $\epsilon;\epsilon \vdash E : [;]\langle\epsilon;\epsilon{:}\tau\{\varrho\}\rangle$. Then by Lemma B.8 both $E\langle e_1\{\varrho\}\rangle$ and $E\langle e_2\{\varrho\}\rangle$ diverges as required. $\square$

Lemma B.11 *If $\Delta_1 \vdash \varrho : \Delta_2$ and $e_1 \equiv_{\Delta_1,\Delta_2;\Gamma:\tau} e_2$ then $e_1\{\varrho\} \equiv_{\Delta_1;\Gamma\{\varrho\}:\tau\{\varrho\}} e_2\{\varrho\}$.*

Proof: Follows by Lemma B.5 and the fact that a Δ_1 type substitution and a Δ_2 type substitution can be spliced together to form a Δ_1, Δ_2 type substitution. □

Lemma B.12 *If $e \mapsto v$, $\epsilon; \epsilon \vdash e : \tau$, and $\epsilon; x{:}\tau \vdash e' : \sigma$ then $e'\{x := e\} \equiv_{\epsilon;\epsilon:\sigma} e'\{x := v\}$.*

Proof: By CIU, let E be such that $\epsilon; \epsilon \vdash E : [;]\langle\epsilon; \epsilon{:}\sigma\rangle$. If both $E\langle e'\{x := e\}\rangle$ and $E\langle e'\{x := v\}\rangle$ diverge, then the result follows. Otherwise, consider $e_1\{x := e\}$ and $e_1\{x := v\}$ where one converges, the goal is show that both converge by induction on the least convergence length. If $e_1\{x := e_2\} \mapsto e_3\{x := e_2\}$ for all e_2 and some e_4 then the result follows by the induction hypothesis. Otherwise $e_1\{x := e\} = E\langle x\rangle\{x := e\}$ and $e_1\{x := v\} = E\langle x\rangle\{x := v\}$. Then $e_1\{x := v\} \mapsto^* E\{x := e\}\langle v\rangle$, so by induction on the number of occurrences of x in E, the other case eventually holds. □

Lemma B.13 *If $e_1 \equiv_{\epsilon;x:\tau:\sigma} e_2$ and $v_1 \equiv_{\epsilon;\epsilon:\tau} v_2$ then $e_1\{x := v_1\} \equiv_{\epsilon;\epsilon:\sigma} e_2\{x := v_2\}$.*

Proof: By CIU Congruence and CIU $[\mathsf{apply} = \_[](x{:}\tau){:}\sigma.e_1;] \equiv_{\epsilon;\epsilon:[\mathsf{apply}:[](\tau):\sigma]} [\mathsf{apply} = \_[](x{:}\tau){:}\sigma.e_2;]$. Since for any appropriate E $E\langle[;1 = v_1, 2 = v_2]\rangle \mapsto v_i$, it is easy to establish that $[;1 = [\mathsf{apply} = \_[](x{:}\tau){:}\sigma.e_1;], 2 = v_1] \equiv_{\epsilon;\epsilon:\tau'} [;1 = [\mathsf{apply} = \_[](x{:}\tau){:}\sigma.e_2;], 2 = v_2]$ for $\tau' = [;1 = [\mathsf{apply}{:}[](\tau){:}\sigma]^+, 2 = \tau^+]$. By CIU Congruence for $C = \langle\rangle.1.\mathsf{apply}[](\langle\rangle.2)$, $C\langle[;1 = [\mathsf{apply} = \_[](x{:}\tau){:}\sigma.e_1;], 2 = v_1]\rangle \equiv_{\epsilon;\epsilon:\sigma} C\langle[;1 = [\mathsf{apply} = \_[](x{:}\tau){:}\sigma.e_2;], 2 = v_2]\rangle$. The result follows since $C\langle[;1 = [\mathsf{apply} = \_[](x{:}\tau){:}\sigma.e_i;], 2 = v_i]\rangle \mapsto^3 e_i\{x := v_i\}$. □

Definition B.4 $\Gamma = \Gamma_1 + \Gamma_2$ *if and only if* $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma_1) \cup \mathrm{dom}(\Gamma_2) \wedge (x{:}\tau \in \Gamma \Leftrightarrow (x{:}\tau \in \Gamma_1 \text{ xor } x{:}\tau \in \Gamma_2))$

Lemma B.14 *For term substitutions ρ_1 and ρ_2 such that for all $x \in \mathrm{dom}(\Gamma)_1$ both $\rho_1(x)$ and $\rho_2(x)$ converge, if $\Gamma = \Gamma_1 + \Gamma_2$, $\rho_1 \equiv_{\Delta:\Gamma_1} \rho_2$, and $e_1 \equiv_{\Delta;\Gamma:\tau} e_2$ then $e_1\{\rho_1\} \equiv_{\Delta;\Gamma_2:\tau} e_2\{\rho_2\}$.*

Proof: By CIU and Lemma B.5 it suffices to show the result for $\Delta = \epsilon$ and $\Gamma_2 = \epsilon$. The result follows by induction on the size of Γ_1. Let $e_1 \equiv_{\epsilon;x:\tau':\tau} e_2$ (1), $e'_1 \equiv_{\epsilon;\epsilon:\tau'} e'_2$ (2), $e'_1 \mapsto^* v_1$, and $e'_2 \mapsto^* v_2$. By (2) and Lemma B.9 $v_1 \equiv_{\epsilon;\epsilon:\tau} v_2$. Then by Lemma B.12 $e_i\{x := e'_i\} \equiv_{\epsilon;\epsilon:\sigma} e_1\{x := v_i\}$. The result follows by Lemma B.13. □

B.2 Closed Under Reduction

Definition B.5

- *A Δ-TP (typed pair) is a triple (v_1, v_2, τ) such that $\Delta; \epsilon \vdash v_1 : \tau$ and $\Delta; \epsilon \vdash v_2 : \tau$.*
- *A $\Delta; \Gamma$-STP (set of typed pairs) is a set of Δ-TPs.*
- *A Δ-STP $\{(v_1^i, v_2^i, \tau^i)\}$ is Δ-TOE (typed observationally equivalent) if and only if*

$$v_1^i \equiv_{\Delta;\epsilon:\tau^i} v_2^i$$

- *A pair of contexts E_1 and E_2 is a $\Delta{:}\tau_1{\Rightarrow}\tau_2$-reduction pair if and only if they satisfy one of the forms in the table below.*

E_1/E_2	τ_1/τ_2	*Side Conditions*
$\langle\rangle.m[\vec{\sigma}](v_{11},\ldots,v_{1n})$ $\langle\rangle.m[\vec{\sigma}](v_{21},\ldots,v_{2n})$	$[m{:}[\vec{\alpha}](\tau'_1,\ldots,\tau'_n)\to\tau']$ $\tau'\{\vec{\alpha}:=\vec{\sigma}\}$	$v_{1i}\equiv_{\Delta;\epsilon:\tau'\{\vec{\alpha}:=\vec{\sigma}\}} v_{2i}$
$\langle\rangle.f$ $\langle\rangle.f$	$[;f{:}\tau_2^{\phi}]$	$\phi\in\{+,\circ\}$
$\langle\rangle.f\leftarrow v_1$ $\langle\rangle.f\leftarrow v_2$	$[;f{:}\tau^{\phi}]$ τ_1	$\phi\in\{-,\circ\}; v_1\equiv_{\Delta;\epsilon:\tau} v_2$
$\langle\rangle[\sigma]$ $\langle\rangle[\sigma]$	$\forall\alpha.\tau'$ $\tau'\{\alpha:=\sigma\}$	

- *For a Δ-STP A: $v_1\equiv_{\Delta;\epsilon;A:\tau} v_2$ if and only if either $v_1\equiv_{\Delta;\epsilon:\tau} v_2$ or there exists τ' such that $(v_1,v_2,\tau')\in A$ and $\Delta\vdash\tau'\leq\tau$.*

- *A Δ-STP A is strictly closed under reduction if and only if for each Δ-TP $(v_1,v_2,\tau)\in A$ and every $\Delta{:}\tau_1{\Rightarrow}\tau_2$-reduction pair E_1 and E_2 such that $\Delta\vdash\tau\leq\tau_1$:*
  - $E_1\langle v_1\rangle\mapsto v'_1$ *implies* $\exists v'_2 : E_2\langle v_2\rangle\mapsto v'_2\wedge(v'_1\equiv_{\Delta;\epsilon;A:\tau_2} v'_2\vee(v'_1=E_1\langle v_1\rangle\wedge v'_2=E_2\langle v_2\rangle))$
  - $E_2\langle v_2\rangle\mapsto v'_2$ *implies* $\exists v'_1 : E_1\langle v_2\rangle\mapsto v'_1\wedge(v'_1\equiv_{\Delta;\epsilon;A:\tau_2} v'_2\vee(v'_1=E_1\langle v_1\rangle\wedge v'_2=E_2\langle v_2\rangle))$

- *Closed under reduction is the same as strictly closed under reduction except that the self variable is ignored in the method invocation rule. To be precise, if by the method invocation rule $E_1\langle v_1\rangle\mapsto v'_1\{x:=v_1\}$ where x is the self variable, and $E_2\langle v_2\rangle\mapsto v'_2\{x:=v_2\}$ then $v'_1\equiv_{\Delta;x:\tau:\tau_2} v'_2$ must hold.*

- *Two value substitutions ρ_1 and ρ_2 are Γ-observational equivalent substitutions of A, $\rho_1\equiv^A_\Gamma\rho_2$, if and only if $\forall x\in\mathrm{dom}(\Gamma) : (\rho_1(x),\rho_2(x),\Gamma(x))\in A$.*

Lemma B.15 *If a Δ-STP A is strictly closed under reduction then A is Δ-TOE.*

Proof: (Assume reduction is deterministic.)

By CIU and Lemma B.5 it suffices to show that for all ϱ such that $\epsilon\vdash\varrho:\Delta$ that $A\{\varrho\}$ is ϵ-TOE. Let $A'=A\{\varrho\}\cup\{(v_1,v_2,\tau)|v_1\equiv_{\epsilon;\epsilon:\tau} v_2\}$ for an arbitrary ϱ such that $\epsilon\vdash\varrho:\Delta$. Clearly the union of two sets that are strictly closed under reductions is also strictly closed under reductions. By a tedious inspection of the definitions, A' is ϵ-STP that is strictly closed under reduction. Consider an arbitrary $(v_1,v_2,\tau)\in A'$, let C be such that $\epsilon;\epsilon\vdash C:[;]\langle\epsilon;\epsilon{:}\tau\rangle$, need to show that $C\langle v_1\rangle\sim C\langle v_2\rangle$. If both these sequences diverge the result follows, otherwise without loss of generality assume that $C\langle v_1\rangle\downarrow$, need to show that $C\langle v_2\rangle\downarrow$. I will show by induction on the length of $e\{\rho_1\}\mapsto^* v$ that $e\{\rho_2\}\downarrow$ where $\epsilon;\Gamma\vdash e:$ and $\rho_1\equiv^{A'}_\Gamma\rho_2$ for some Γ. If $e\{\rho_1\}$ is a value then clearly so is $e\{\rho_2\}$ as both ρ_1 and ρ_2 are value substitutions. If $e\{\rho\}\mapsto e'\{\rho\}$ for all appropriate ρ then the result follows by the induction hypothesis. Otherwise, by inspection of the reduction rules there exists E, E', and $x\in\mathrm{dom}(\Gamma)$ (but, see proof of Lemma B.4) such that $e\{\rho\}=E'\langle E\langle x\rangle\rangle\{\rho\}$ for all appropriate ρ and where E has one of the forms $\langle\rangle.m[\vec{\tau}](\vec{e})$, $\langle\rangle.f$, $\langle\rangle.f\leftarrow e$, or $\langle\rangle[\tau]$. Since $(\rho_1(x),\rho_2(x),\Gamma(x))\in A'$ and A' is strictly closed under reductions, $E\langle x\rangle\{\rho_i\}\mapsto v_i$ and either $v_i=E\langle x\rangle\{\rho_i\}$, $v_1\equiv_{\epsilon;\epsilon:\sigma} v_2$, or $(v_1,v_2,\sigma)\in A'$ for some σ. In the first case clearly $e\{\rho_i\}\uparrow$ as required. In the other two cases $(v_1,v_2,\sigma)\in A'$. Let y be fresh,

$\rho_1' = \rho_1, y := v_1$, $\rho_2' = \rho_2, y := v_2$, $\Gamma' = \Gamma, y{:}\sigma'$ and the result follows from the induction hypothesis on $E'\langle y\rangle\{\rho_i'\}$. $\square$

Lemma B.16 *If a Δ-STP A is closed under reduction then A is Δ-TOE.*

Proof: Let $A[n]$ be the set of Δ-TPs obtained from A by inserting unwinding numbers as follows: on object constructors that are values of a triple place n, and on all other object constructors place ω. I will prove by induction on $i \in \mathbb{N}$ that $A[i]$ is Δ-TOE, the result follows by Lemma A.3. I claim that $A[i]$ is strictly closed under reduction. Let $(v_1^i, v_2^i, \tau) \in A$, E_1 and E_2 be a $\Delta{:}\tau_1{\Rightarrow}\tau_2$-reduction pair, and $\Delta \vdash \tau \leq \tau_1$. Then since A is closed under reduction, either the required conditions hold, or $E_1\langle v_1^\omega\rangle \mapsto v_1'\{x := v_1^\omega\}$, $E_2\langle v_2^\omega\rangle \mapsto v_2'\{x := v_2^\omega\}$, and $v_1' \equiv_{\Delta;x:\tau:\tau_2} v_2'$ (1). If $i > 0$ then $E_1\langle v_1^i\rangle \mapsto v_1'\{x := v_1^{i-1}\}$ and similarly for v_2. Since $(v_1^{i-1}, v_2^{i-1}, \tau) \in A[i-1]$, by the induction hypothesis $v_1^{i-1} \equiv_{\Delta;\epsilon:\tau} v_2^{i-1}$. Thus by (1) $v_1'\{x := v_1^{i-1}\} \equiv_{\Delta;\epsilon:\tau_2} v_2'\{x := v_2^{i-1}\}$ satisfying the first part of $\equiv_{\Delta;\epsilon;A:\tau_2}$. If $i = 0$ then $E_1\langle v_1^0\rangle \mapsto E_1\langle v_1^0\rangle$ and $E_2\langle v_2^0\rangle \mapsto E_2\langle v_2^0\rangle$ as required. Thus $A[i]$ is strictly closed under reduction, so by Lemma B.15 $A[i]$ is Δ-TOE. $\square$