Object Closure Conversion

Neal Glew [1]

Department of Computer Science
Cornell University
Ithaca, USA

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

[1] 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 a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

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 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. Because of space limitations, the correctness theorem is stated here, and the proof appears in a companion technical report [Gle99]. 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:

$$\text{Types } \tau, \sigma ::= [m_i{:}\tau_i; f_j{:}\sigma_j]_{i \in I, j \in J}$$

$$\text{Terms } e, b ::= x \mid [m_i = x_i.b_i{:}\tau_i; f_j = e_j]_{i \in I, j \in J} \mid e.m \mid e.f \mid e_1.f \leftarrow e_2$$

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.[2] 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 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:

$$\begin{aligned}
|x| &= x \\
|e.m| &= |e|.m \\
|e.f| &= |e|.f \\
|e_1.f \leftarrow e_2| &= |e_1|.f \leftarrow |e_2| \\
|e| &= [m_i = x_i.b_i'{:}\tau_i; f_j = |e_j|, g_k = y_k]_{i\in I, j\in J, 1\le k\le n} \\
&\quad \text{where } e = [m_i = x_i.b_i{:}\tau_i; f_j = e_j]_{i\in I, j\in J} \\
&\qquad \mathrm{cv}(e) = \{y_1, \ldots, y_n\} \\
&\qquad b_i' = |b_i|\{y_1, \ldots, y_n := x_i.g_1, \ldots, x_i.g_n\} \\
&\qquad g_k \text{ are fresh}
\end{aligned}$$

[2] 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.

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:

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

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 they have 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.[3] 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 proven by standard techniques. Type soundness and several related properties of the typing rules are used in the proof of correctness; details appear in the companion technical report [Gle99].

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

Syntax:

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

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	$k \in J$
$v.f_k \leftarrow w$	$[m_i = M_i; f_j = v'_j]_{i \in I, j \in J}$	$k \in J, v'_j = \begin{cases} v_j & j \neq k \\ w & j = k \end{cases}$
$(\Lambda\alpha.w)[\tau]$	$w\{\alpha := \tau\}$	

Fig. 1. Operational Semantics

The *free variables* of an expression are:

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

The *closure variables* of an object constructor are:

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

Again note that only the free variables of methods are included. 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

Typing Contexts:

$$\Delta ::= \alpha_1, \ldots, \alpha_n \qquad \text{where } \alpha_1, \ldots, \alpha_n \text{ are distinct}$$

$$\Gamma ::= x_1{:}\tau_1, \ldots, x_n{:}\tau_n \text{ where } x_1, \ldots, x_n \text{ are distinct}$$

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

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

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

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

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

Expression typing:

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

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

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

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

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

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

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

Fig. 2. Typing Rules

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

$$
\begin{array}{lll}
|x| & = x \\
|e.m[\vec{\tau}](e_1, \ldots, e_n)| & = |e|.m[\vec{\tau}](|e_1|, \ldots, |e_n|) \\
|e.f| & = |e|.f \\
|e_1.f \leftarrow e_2| & = |e_1|.f \leftarrow |e_2| \\
|\Lambda\alpha.b| & = \Lambda\alpha.|b| \\
|e[\sigma]| & = |e|[\sigma] \\
|e| & = e' \\
\quad \text{where } e & = [m_i = M_i; f_j = e_j]_{i\in I, j\in J} \\
\qquad e' & = [m_i = M_i'; f_j = |e_j|, g_1 = y_1, \ldots, g_n = y_n]_{i\in I, j\in J} \\
\qquad \mathrm{cv}(e) & = \{y_1, \ldots, y_n\} \\
\qquad M_i & = x_i[\vec{\alpha}_i](\overrightarrow{x_i{:}\tau_i}).b_i{:}\tau_i \\
\qquad M_i' & = x_i[\vec{\alpha}_i](\overrightarrow{x_i{:}\tau_i}).|b_i|\{y_1, \ldots, y_n := x_i.g_1, \ldots, x_i.g_n\}{:}\tau_i \\
\qquad g_1, \ldots, g_n & \text{ are fresh}
\end{array}
$$

Fig. 3. Object Closure Conversion Translation

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:

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

But the translated term makes this transition:

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

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:* [4]

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

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

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

The above relations are equivalence relations and observational equivalence is a congruence. These properties are proven in the companion technical report [Gle99], 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. Some principles for establishing equivalence are also proven.

[4] 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.

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 the companion technical report [Gle99].

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

The proof appears in the companion technical report [Gle99], and is by induction on the structure of e or its typing derivation. For all cases other than object constructors, the proof is straightforward, and uses the congruence property of equivalence. To prove operational correctness for object constructors requires two key lemmas. The first shows that an object is equivalent to itself with some extra fields. Intuitively this is true because the extra fields are just carried around, and contexts cannot see the extra fields at the original type. The proof uses the coinduction principle in a straightforward manner. The other lemma shows that one method body can be replaced with its translation. The coinduction principle is used again. The substitution $\vec{y} := \overrightarrow{x_i.g}$ preserves equivalence because $x_i.g_k \mapsto y_k$, so one of the basic lemmas says that $x_i.g_k$ and y_k are equivalent.

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 including an imperative semantics, right-extension subtyping, integers, products, sums, arrays, recursive types, and functions. Details appear in the companion technical report [Gle99].

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.

The translation can also be extended to handle method update, field extension, and method extension. All of these require field extension, which interacts poorly with breadth subtyping. Thus, either a different operational semantics is required, such as the dictionary semantics [RS98], or a more complicated type system, such as described by Fisher [Fis96]. Details appear in the companion technical report [Gle99].

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.

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:

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

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

$$\begin{aligned} |\mathsf{int}| &= \mathsf{int} \\ |\tau_1 \to \tau_2| &= [\mathsf{apply}{:}[\,](|\tau_1|) \to |\tau_2|;\,] \end{aligned}$$

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

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

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

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, and $e_1.f \leftarrow e_2$ denotes field update. First, consider an encoding based on the self-application semantics [Kam88]. Ignoring typing, the interesting composed translation rules are:

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

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]. Ignoring typing, the interesting composed translation rules are:

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

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 gen-

eral 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.

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

[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`.

[Gle99] Neal Glew. Object closure conversion. Technical Report TR99-1763, Department of Computer Science, Cornell University, 4130 Upson Hall, Ithaca, NY 14853-7501, USA, August 1999. Available at `http://www.cs.cornell.edu/glew/paper-list.html`.

[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, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA, 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.