Object Conversion is Type-Preserving

David McAllester

July 23, 1998

Abstract

Closure conversion is a program transformation widely used in modern compilers. For a variety of reasons it is desirable that compilation preserve types. We consider a closure conversion mapping e to O[e] which is naive in the sense that O[e] does not contain pack or unpack programming constructs. The main theorem states that, for this fixed notion of closure conversion, any "normal" type system T for the lambda calculus can be mechanically converted to an object-oriented type system T' such that for any lambda term e we have that e is typable by T iff O[e] is typable by T'. Simple types, partial types, union types, intersection types, Hindley-Milner types and system F are all normal. The main theorem can also be interpreted as a formalization of the common claim that objects subsume procedures and that first-order object-oriented programs are as expressive as the lambda calculus.

1 Introduction

C++ and Java, the most widely used object-oriented programming languages, do not support first-class procedures. Various proposals have been made for higher-order object-oriented languages, e.g., [12, 20, 1, 17]. However, it is part of the folklore of object-oriented programming that procedures are a special case of objects — a procedure is just an object with an apply method. According to this folklore procedures do not increase the "expressive power" of an object-oriented language. One of the objectives of this paper is to rigorously addresses the relative expressive power of first-order vs. higher-order object-oriented programming languages. The fundamental difficulty in measuring the relative expressive power of programming languages is the so-called Turing tar pit — any two Turing-complete programming languages can express exactly the same class of partial functions on the natural numbers. Intuitively, however, if a program is translated from a high level (expressive) to a low level (inexpressive) language the program becomes less clear — more difficult to read and verify. This can be made rigorous by defining clarity in terms of formal static analysis. We say that a translation from one language to another is analysis-preserving if any statically inferable property of the source program is also statically inferable on the target program. The main theorem of this paper states that a naive translation from the pure lambda calculus to first order object-oriented programs preserves clarity in the sense that, for a broad class of type systems, the source program is typable if and only if its translation is typable. The type systems for which this hold include simple types, partial types, intersection and union types, Hindley-Milner types and implicit system F.

The main theorem given here is also related to type-preservation compilation and proof-carrying code [9, 11]. Minamide, Morrisett, and Harper take a "closures as objects" view of the closure conversion [7]. Here we use the term "object conversion" for a closure conversion process in which closures are viewed as objects. Rather than use an object type system based on record types, Minamide et al. adopt a type system for objects based on existential types [16]. In this approach polymorphism requires special treatment. The main theorem of this paper can be viewed as stating that a naive object conversion preserves typability under an infinite class of type systems including Hindley-Milner types and implicit system F. No special treatment of polymorphism is required.

A third issue addressed by this paper is the expressive power of first order rewrite rules. First order rewrite rules have been widely used as a model of programming, e.g., [13]. A standard objection to first order rewrite rules as a model of programming is the lack of higher order abstraction and first class procedures. The naive object conversion used here translates the lambda calculus into first order rewrite rules. This translation is done in such a way that typability in any normal type system for the lambda calculus can be naturally determined from the translation of a lambda term. The main theorem here can be interpreted as saying that first order rewrite rules can support lambda calculus type systems, e.g., system F.

The main theorem is founded on a simple but apparently novel theory of objects. An object-oriented program is taken to be a set of first order rewrite rule of the following form.

$$f(c(y_1, \ldots, y_n), x_1, \ldots, x_k) \rightarrow b$$

Here the function c is an object-constructor symbol and f is a message name. We assume that object constructors are disjoint from message names. The term $f(c(y_1, \ldots, y_n), x_1, \ldots, x_k)$ might be expressed in a more traditional notation as $(c(y_1, \ldots, y_n).f)(x_1, \ldots, x_k)$. Rewrite rules of this form will be called first order method definitions. We assume there is at most one method definition for a given pair of c and f. The right hand side b can be any first order term constructed from object-constructors and message names whose free variables are among $y_1, \ldots, y_n, x_1, \ldots, x_k$.

It is easy to see that first-order method definitions provide a Turing-complete programming language. In particular one can implement classical conditional expressions using the method `if` as defined by the following method definitions.

$$\texttt{if}(\texttt{true}, x, y) \rightarrow x$$

$$\texttt{if}(\texttt{false}, x, y) \rightarrow y$$

First order method definitions form a "pure" object system in the sense that all values are objects.

Object conversion can be formulated as a translation from the pure lambda calculus into first-order terms plus a set of first-order method definitions. As usual, the lambda-calculus is defined by the following grammar.

$$e ::= x \mid \lambda x.e \mid (e_1\ e_2)$$

The object-conversion $O[e]$ of an expression e is defined by the following equations.

$$O[x] = x$$

$$O[(e_1,\ e_2)] = \mathtt{apply}(O[e_1],\ O[e_2])$$

$$O[\lambda x.e] = c_{\lambda x.e}(y_1,\ \ldots,\ y_n)$$

In the last equation $y_1, \ldots, y_n$ are the free variables of $\lambda x.e$ and $c_{\lambda x.e}$ is a constructor symbol specific to $\lambda x.e$. For theoretical simplicity we can assume an infinite set of such constructor symbols, one for each lambda-expression of the lambda calculus, where for each constructor of the form $c_{\lambda x.e}$ we have the following method definition.

$$\mathtt{apply}(c_{\lambda x.e}(y_1,\ \ldots,\ y_n),\ x) \rightarrow O[e]$$

The main theorem states that this object conversion transformation is type-preserving under a wide variety of type systems for the lambda calculus.

Before presenting the technical details we give a brief discussion of previous work on the expressiveness of programming languages. The schematology model of expressiveness treats most program operations, such as +, as symbols of unspecified meaning. The abstract meaning of a program expression then becomes a function from interpretations of the meaning of the unspecified function symbols to the function denoted by the program under that interpretation. This abstract meaning of the program can be represented by "unfolding" the expression into a possibly infinite first order term [14, 6, 2, 19]. The most fundamental result is that a language with general recursion can express abstract meanings not expressible in a language with iteration but not general recursion. However, under the schematology model full recursion does not increase the expressive power of a language with an appropriate fixed notion of a stack data structure.

In the abstraction-preserving model one says that a source language is no more expressive than a target language if the source can be translated to the target in such a way that all semantically equivalent (observationally equivalent) source expressions translate to semantically equivalent (observationally equivalent) target expressions [8, 18, 21]. For example, the continuation-passing style (CPS) conversion preserves abstraction. However, consider a standard compilation of continuations into traditional first order stacks. This translation is not abstraction-preserving because semantically equivalent continuations map to non-equivalent stacks, e.g., stacks of different length. So higher-order languages are more expressive than first-order languages — higher-order languages allow more information hiding than do first-order languages. However, the increased expressive power of higher order languages seems to disappear if we compare higher order languages to *object-oriented* first-order languages. Object-oriented first-order languages provide information hiding in a manner analogous to higher-order languages.

An analysis-preserving model of expressiveness is different from either the schematology model or the abstraction-preserving model. The analysis-preserving model seems to make more distinctions. Intuitively, it is easier to construct abstraction-preserving translations than analysis-preserving translations. For example, there is a simple translation of objects into procedures that take message arguments where the messages are symbolic tokens. This translation can be done in an abstraction-preserving way. However, type systems for object-oriented programs can be viewed as performing a case analysis on the message sent to the procedure. No standard type system for functional programs performs such a case analysis. So, although this natural translation is abstraction-preserving, it is not analysis-preserving. Under the analysis-preserving model of expressiveness it is difficult to argue that procedures subsume objects — objects seem strictly more general than procedures.

In order to formulate a type-preservation theorem for an infinite class of type systems it is necessary to define the concept of a type system in general. Such a definition amounts to the construction of a logical framework — a formalism for expressing (type) inference rules [4, 15, 3, 10]. Here we use a simple framework consisting of a notion of inference rules with built-in notions of free variable and substitution using λ as the only quantifier. It differs from LF [3] and [10] in that it does not directly support α-β-equivalence of lambda expressions and in particular does not provide higher order unification or matching. In the framework used here β-conversion must be handled explicitly. However, the framework used here does support implicit substitution. We call it a semi-explicit framework.

2 The Semi-Explicit Framework (SEF)

A type system is a set of inference rules. The simplest formal model of an inference rule is a first order Horn clause. It is well known that first order Horn clauses (logic programs) can express any RE set and hence can express any RE notion of typability. However, inference rules are commonly phrased using side conditions of the form $x \notin FV(u)$, stating that the variable x does not occur free in the expression u. Also, it is common for rules to use substitution expressions of the form $u[w/x]$. For example, consider the following two inference rules of implicit system F.

$$\frac{\Phi \vdash e{:}\, \tau,\ \ \alpha \notin FV(\Phi)}{\Phi \vdash e{:}\, \forall\alpha\, \tau} \qquad\qquad \frac{\Phi \vdash e{:}\, \forall\alpha\, \tau \quad \mathtt{type}(\sigma)}{\Phi \vdash e{::}\, \tau[\sigma/\alpha]}$$

If we ignore the side condition $\alpha \notin FV(\Phi)$ then the first rule can be viewed as a Horn clause with variables Φ, e, τ, and α and using predicate $\vdash$ and term constructors ":" and $\forall$. The variables in the rule are best viewed as meta-variables. In the first rule α is a meta-variable which ranges over variables of the object language. It is not difficult to implement the side conditions and substitution operation explicitly in first order Horn clauses, i.e., as a logic program. However, the above rules can be phrased more naturally, and the main theorem of this paper proved more easily, if we use an enriched notion of inference rule. In this section we formalize a notion of Horn clause with side conditions of the form $x \notin FV(u)$ and substitution expressions of the form $u[w/x]$.

We first define an object language independent of the meta-language. The object-expressions are defined by the following grammar.

$$\mathtt{e} ::= \mathtt{x} \mid \mathtt{f}(\mathtt{e}_1,\ \ldots,\ \mathtt{e}_n) \mid \mathtt{g}(\lambda\mathtt{x}.e_1,\ e_2,\ \ldots,\ e_k)$$

The expression $\mathtt{x}$ in the above grammar represents an arbitrary object-level variable (as opposed to the meta-level variables used in rules). In this grammar $\mathtt{f}$ denotes an object level constructor of arity (number of arguments) n and $\mathtt{g}$ denotes a "binding construct" of arity k where the first argument of a binding construct must be a lambda expression. We assume an infinite set of constructors and binding constructs of every arity. All quantification is reduced to lambda expressions appearing as the first argument of binding constructs. For example, a universally quantified formula is represented as $\forall(\lambda x.\Phi)$ where $\forall$ is a monadic binding construct. A procedure with an explicitly stated input type can be written as $\mathtt{typed\_proc}(\lambda x.u,\ \tau)$ where τ is the input type. The notions of free and bound variable and α-equivalence are defined as in the lambda calculus. For any object-expressions $\mathtt{e}$ and $\mathtt{u}$ and object-variable $\mathtt{x}$ we define $\mathtt{e}[\mathtt{u}/\mathtt{x}]$ in the usual way as the result of replacing all free occurrences of $\mathtt{x}$ in $\mathtt{e}$ by $\mathtt{u}$ with renaming of bound variables in $\mathtt{e}$ to avoid the capture of free variables in $\mathtt{u}$. We let $FV(\mathtt{e})$ denote the set of free variables in the object-expression $\mathtt{e}$. The definition of an object expression does not involve types — any expression generated by the above grammar is an object-expression. As discussed below, typability is defined by a set of inference rules.

In the formulation of inference rules we use three kinds of meta-variables — unconstrained meta-variables, abstraction meta-variables, and variable meta-variables. A meta-substitution ξ is a mapping from meta-variables to object-expressions such that for every abstraction meta-variable f we have that $\xi(f)$ is an object-level lambda-expression and for every variable meta-variable x we have that $\xi(x)$ is an object-level variable. Meta-expressions are defined by the following grammar where x, f, and e represent arbitrary variable meta-variables, abstraction meta-variables, and unconstrained meta-variables respectively and $\mathtt{x}$ represents an arbitrary object-level variable.

$$V ::= \mathtt{x} \mid x$$

$$F ::= f \mid \lambda V.M$$

$$M ::= V \mid F \mid e \mid \mathtt{f}(M_1, \ldots, M_n) \mid g(F, M_1, \ldots, M_k) \mid \mathtt{mlet}(F, M)$$

The special meta-level binding construct mlet is used to represent substitution expressions. The expression $u[w/x]$ is written as $\mathtt{mlet}(\lambda x.u, w)$. If ξ is a meta-substitution and m is a meta-expression then we let $\xi[m]$ be the result of replacing every meta-variable in m by the value assigned by ξ. There is no notion of free or bound meta-variable and no renaming is done in the process of meta-substitution. For example, if $\xi(x)$ is $\mathtt{y}$ and $\xi(e)$ is $\mathtt{f(y)}$ then $\xi[\lambda x.e]$ is $\lambda\mathtt{y}.\mathtt{f}(\mathtt{y})$. For any meta-substitution ξ and any meta-expression m we have that $\xi[m]$ is an expression generated by the following grammar.

$$\mathtt{e} ::= \mathtt{x} \mid \mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n) \mid g(\lambda\mathtt{x}.e_1, e_2, \ldots, e_k) \mid \mathtt{mlet}(\lambda\mathtt{x}.e_1, e_2)$$

For any such expression $\mathtt{e}$ we define $\downarrow\mathtt{e}$ to be the result of repeatedly replacing subexpressions of the form $\mathtt{mlet}(\lambda\mathtt{x}.\mathtt{u}, \mathtt{w})$ by $\mathtt{u}[\mathtt{w}/\mathtt{x}]$. It is possible to show that any such process of let-reduction terminates and that the final result is independent of the order of the reductions. For any set $\mathtt{B}$ of object-expression we let $\downarrow\mathtt{B}$ be the result of replacing each element $\mathtt{e}$ of $\mathtt{B}$ by $\downarrow\mathtt{e}$.

A SEF inference rule is a triple $B, C \Rightarrow H$ where H is a meta-expression, B is a finite set of meta-expressions, and C is a (possibly empty) set of side conditions of the form $x \notin FV(m)$ where x is a variable meta-variable and m is a meta-expression. Let $B, C \rightarrow H$ be a SEF inference rule and let ξ be a meta-substitution such that for each side condition $x \notin FV(m)$ in C we have that $\xi(x)$ is not in $FV(\xi[m])$. The pair $\downarrow\xi[B] \Rightarrow \downarrow\xi[H]$ is called an *instance* of $B, C \Rightarrow H$. An object-expression $\mathtt{e}$ is said to be derivable by a set R of SEF inference rules, written $R \mathrel{\vdash\!\!\sim} \mathtt{e}$, if there is an instance $\mathtt{B} \rightarrow \mathtt{e}'$ of a rule in R such that $R \mathrel{\vdash\!\!\sim} \mathtt{w}$ for each $\mathtt{w}$ in $\mathtt{B}$ and where $\mathtt{e}'$ is α-equivalent to $\mathtt{e}$. The base case occurs when $\mathtt{B}$ is empty.

For example, the inference rules at the beginning of this section can be written as the following SEF rules.

$$\{\Phi \vdash e\colon \tau\},\ \{x \notin FV[\Phi]\} \quad\Rightarrow\quad \Phi \vdash e\colon \mathtt{type\_forall}(\lambda x.\tau)$$

$$\{\Phi \vdash e\colon \mathtt{type\_forall}(f),\ \mathtt{type}(\sigma)\},\ \{\} \quad\Rightarrow\quad \Phi \vdash e\colon \mathtt{mlet}(f, \sigma)$$

These rules have the following two instances respectively.

$$\{T \vdash \mathtt{proc}(\lambda x.x)\colon \alpha \rightarrow \alpha\} \quad\Rightarrow\quad T \vdash \mathtt{proc}(\lambda\mathtt{x}.\mathtt{x})\colon \mathtt{type\_forall}(\lambda\alpha.\alpha \rightarrow \alpha)$$

$$\{T \vdash \mathtt{proc}(\lambda\mathtt{x}.\mathtt{x})\colon \mathtt{type\_forall}(\lambda\alpha.\alpha \rightarrow \alpha),\ \mathtt{type}(\mathtt{int})\}$$

$$\Rightarrow\quad T \vdash \mathtt{proc}(\lambda\mathtt{x}.\mathtt{x})\colon \mathtt{int} \rightarrow \mathtt{int}$$

3 The Main Theorem

For any set R of SEF inference rules we say that an expression $\mathtt{e}$ is typable by R if there exists a Φ and τ such that $R \mathrel{\vert\!\sim} \Phi \vdash e\colon \tau$. We define the pure lambda calculus to be the set of object-level expressions generated by the following grammar where `proc` is a binding construct and `apply_proc` is a constructor.

$$e ::= x \mid \mathtt{proc}(\lambda x.e) \mid \mathtt{apply\_proc}(e_1,\ e_2)$$

Figure 1 shows the fundamental SEF inference rules for typing procedures. Let R_P be the rules in figure 1. Figure 2 gives a set of optional rules. $R_P \cup \{S1,\ S2\}$ type exactly those expression of the pure lambda calculus which are typable by simple types. By selecting appropriate additional rules from figure 2 we can construct type systems for intersection types, partial types, and implicit system F.

As stated in the introduction we model object-oriented programming with first order rewrite rules of the following form where f is viewed as a message name and c as an object class.

$$f(c(y_1,\ \ldots,\ y_n),\ x_1,\ \ldots,\ x_k) \to e$$

Here e can be any object-level expression all of whose free variables are among y_1, …, y_n, x_1, …, x_k. A rewrite rule of this form will be called a method definition on class c. In this rule both f and c are taken to be constructors of the SEF object language. We assume an infinite set of message name constructors f and object constructors c. Furthermore, we assume that each object constructor c is associated with a given finite set of method definitions on c.

Figure 3 gives basic rules for objects. Let R_O be the set of rules in figure 3. The rule set R_O can also be combined with subsets of the rules from figure 2. For example, $R_O \cup \{S1,\ S2,\ S3,\ S4,\ S5,\ S6\}$ gives an object type system with subtyping on object types.

We now define object conversion to be a transformation on arbitrary object-level expressions as given by the following equations.

$$O[\mathtt{x}] = \mathtt{x}$$

$$O[\mathtt{proc}(\lambda\mathtt{x.e})] = c_{\mathtt{proc}(\lambda\mathtt{x.e})}(y_1,\ \ldots,\ y_k)$$

$$O[\sigma \to \tau] = \mathtt{apply\_message}().O[\sigma] \rightharpoonup O[\tau]$$

$$O[\mathtt{apply\_proc}(u,\ w)] = \mathtt{apply\_message}(O[u],\ O[w])$$

$$O[f(e_1,\ \ldots,\ e_n)] = f(O[e_1],\ \ldots,\ O[e_n])$$

$$O[g(\lambda\mathtt{x.e}_1,\ \mathtt{e}_2,\ \ldots,\ \mathtt{e}_k)] = g(\lambda\mathtt{x}.O[\mathtt{e}_1],\ O[\mathtt{e}_2],\ \ldots,\ O[\mathtt{e}_k])$$

In the second equation y_1, …, y_k are all the free variables of $\lambda\mathtt{x.e}$. In the second to last equation f is any constructor other than `apply_proc` or $\to$ and in the last equation g is any binding construct other than `proc`. Constructors of the form $c_{\mathtt{proc}(\lambda\mathtt{x.e})}$ will be called closure constructors. Note that closure constructors are a special case of object constructors. Each such closure constructor is associated with the following single method definition.

$$\texttt{apply\_message}(c_{\texttt{proc}(\lambda\texttt{x.e})}(y_1,\ \ldots,\ y_n),\ \texttt{x}) \rightarrow O[\texttt{e}]$$

The main theorem implies that for any subset S of the rules in figure 2 containing at least S1 and S2, and any expression e of the pure lambda calculus, we have that e is typable by $R_P \cup S$ if and only if $O[e]$ is typable by $R_O \cup S$. It is interesting to note that, for example, $R_P \cup \{S1,\ S2,\ S3,\ S4,\ S5,\ S6\}$ types only those lambda terms typable by simple types while $R_O \cup \{S1,\ S2,\ S3,\ S4,\ S5,\ S6\}$ is an object type system with subtyping on object types. In spite of the apparent differences between these systems, if e is a pure lambda term then $O[e]$ is typable by $R_O \cup \{S1,\ S2,\ S3,\ S4,\ S5,\ S6\}$ if and only if e is typable by simple types.

An expression Φ will be called a well formed antecedent if it is of the form $T \wedge x_1\colon \sigma_1 \wedge \ldots \wedge x_n\colon \sigma_n$ where $x_1, \ldots, x_n$ are distinct variables not occurring in any σ_j. We say that a well formed antecedent Ψ contains a well formed antecedent Φ if every type assignment in Ψ is a type assignment in Φ. We call a set of inference rules S *normal* if it satisfies the following conditions.

1. S contains the rules S1 and S2.

2. If $R_P \cup S \mathrel{|\!\!\sim} \Phi\ \vdash\ e\colon \tau$ then Φ is a well formed antecedent and every free variable of e is assigned a type in Φ.

3. If there exists an n-step $R_P \cup S$ derivation of $\Phi\ \vdash\ e\colon \tau$ and Ψ contains Φ, then there exists an n-step $R_P \cup S$ derivation, using the same final inference rule, of $\Psi\ \vdash\ e\colon \tau$.

4. If there exists an n-step $R_P \cup S$ derivation of $\Phi\ \vdash\ e\colon \tau$ then there exists an n-step $R_P \cup S$ derivation, using the same final rule, of the unique sequent $T \wedge y_1\colon \sigma_1 \wedge \ldots \wedge y_n : \sigma_n\ \vdash\ e\colon \tau$ where $y_1, \ldots, y_n$ are the free variables of e, in the order in which they first appear free in e, and such that Φ contains these type assignments.

5. If $R_P \cup S \mathrel{|\!\!\sim} T \wedge x_1 : \sigma_1 \wedge \ldots \wedge x_n : \sigma_n\ \vdash\ e : \tau$ and $R_P \cup S \mathrel{|\!\!\sim} \Phi\ \vdash\ u_1\colon \sigma_1,\ \ldots, u_n\colon \sigma_n$ then $R_P \cup S \mathrel{|\!\!\sim} \Phi\ \vdash\ e[u_1/x_1,\ \ldots,\ u_n/x_n]\colon \tau$.

6. Conditions 3, 4, and 5 also hold for $R_O \cup S$ derivations.

7. No rule in S mentions `proc`, `apply_proc`, $\rightharpoonup$, or "::" or object constructors or message names.

8. If e is typable by $R_O \cup S$ and e is constructed entirely from variables, closure-constructors and the message name `apply_message` then there exists a derivation of some formula of the form $\Psi\ \vdash\ e : \sigma$ such that the only message name in the derivation is `apply_message` and all object-constructors in the derivation are closure-constructors.

9. For each rule $B, C \to H$ in S, for each meta-substitution ξ satisfying the side conditions C and such that $R_P \cup S \mathrel{|\!\sim} \downarrow\!\xi[B]$, and for each expression of the form $\mathtt{mlet}(\lambda x.u,\ w)$ occurring in B or H we have that $\xi[u]$ does not contain `proc`.

10. If $B, C \to H$ is a rule in $R_O \cup S$ and ξ is a meta substitution such that $R_O \cup S \mathrel{|\!\sim} \downarrow\!\xi[B]$ then for any meta-variable x appearing C we have that $\xi(x)$ does not contain "::".

Although we will not prove it here, any subset of the rules in figure 2 including S1 and S2 satisfies these conditions. All the conditions are fairly natural except conditions 9 and 10. Condition 9, although complex to state, is trivially satisfied in any type system with a clear distinction between terms and types where terms never appear in types and where substitution operations in the inference rules are only done on type expressions, as in rule S10. Condition 10 is trivially satisfied in any type system where the special constructor "::" can not occur below the top level in derived formulas and all meta-variables in all rules appear below the top level in some derived formula. The main theorem does not apply to dependent type systems in which terms can appear in types although some appropriate generalization covering dependent types may be possible. Many other natural rule sets also satisfy the conditions. For example, we could add rules for a `let` binding construct and let-polymorphism. We could add rules for records or pairs or arrays.

> **Main Theorem:** If S is a normal rule set then for any term e of the pure lambda calculus we have that e is typable by $R_P \cup S$ if and only if $O[e]$ is typable by $R_O \cup S$.

The proof is given in an appendix. The basic idea is that one can translate a $R_P \cup S$ derivation of a typing of e to an $R_O \cup S$ derivation of a typing of $O[e]$ where any use of inference rules P1, P2, P3, or P4 in R_P translates to a use of O1, O2, O3&O3$'$, and O4 in R_O respectively, and each use of a rule in S translates to use of the same rule. Derivations of typing for $O[e]$ are translated to derivation of e in an analogous manner. More specifically, one defines an inverse conversion O^{-1} such that for any term e of the pure lambda calculus we have that $O^{-1}[O[e]] = e$. Condition 8 above implies that if there exists a typing of $O[e]$ then there exists a "pure" derivation of a typing of $O[e]$, i.e., one not involving messages other than `apply_message` or object constructors other than closure constructors. We then show that any pure $R_O \cup S$ derivation of a typing for a pure expression w can be translated step by step to an $R_P \cup S$ derivation of a typing for $O^{-1}[w]$.

4 Conclusions

We have shown that a naive object conversion mapping the lambda calculus to first order rewrite rules preserves typability under a wide variety of type systems. This has potential applications in type-preserving compilation and supports the position that first-order object-oriented programming languages are as expressive as the lambda calculus.

Acknowledgements: Kathleen Fisher and Sergei Vorobyov provided considerable useful discussion and analysis of the issues in this paper.

References

[1] François Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In *POPL '97: 24th ACM Symposium on Principles of Programming Languages*, pages 302–315, Paris, January 1997. ACM Press.

[2] Bruno Courcelle. Recursive applicative program schemes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science:Volume B*, pages 459–492. MIT Press, 1990.

[3] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[4] G. Huet and G. Plotkin, editors. *Logical Frameworks*. Cambridge University Press, 1991.

[5] IEEE Computer Society Press. *Proceedings, Thirteenth Annual IEEE Symposium on Logic in Computer Science*, 1998.

[6] Zohar Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1978.

[7] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *POPL96*, 1996.

[8] J.C Mitchell. On abstraction and the expressive power of programming languages. *Science of Computer Programming*, 212:141–163, 1993. Preliminary version appeared *Proc. Theor. Aspects of Computer Software,* Springer LNCS 526, September 1991, pages 750–770.

[9] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. In *POPL98*, 1998.

[10] Gopalan Nadathur and Dale Miller. Higher-order horn clauses. *JACM*, 37(4):777–814, 1990.

[11] George C. Necula and Peter Lee. Efficient representation and validation of proofs. In *Proceedings, Thirteenth Annual IEEE Symposium on Logic in Computer Science* [5], pages 105–117.

[12] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In *24'th ACM Symposium on Principles of Programming Languages*, Paris, January 1997.

[13] Michael O'Donnel. *Equational Logic as a Programming Language.* MIT Press, 1985.

[14] M. Paterson. Equivalence problems in a model of computation. Ph.D. Thesis, University of Cambridge, England, 1967.

[15] Frank Pfenning. Home page for logical frameworks. http://www.cs.cmu.edu/afs/cs.cmu.edu/user/fp/www/lfs.html.

[16] B.C. Pierce and D. N. Turner. Simple type-theoretic foundations for object-oriented programming. *jounral of Functional Programming*, 4(2):207–247, 1994.

[17] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):25–50, 1998. A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[18] Jon G. Riecke. Fully abstract translations between functional languages. *Mathematical Structures in Computer Science*, 3:387–415, 1993. Preliminary version appears in *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 245–254, ACM, 1991.

[19] Geraud Senizergues. The equivalence problem for deterministic pushdown automata is decidable. In *ICALP97*, 1997.

[20] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[21] Ramesh Viswanathan. Full abstraction for first-order object with recursive types and subtyping. In *Proceedings, Thirteenth Annual IEEE Symposium on Logic in Computer Science* [5], pages 380–393.

5 Appendix: The Proof of the Main Theorem

First, suppose that e is typable by $R_P \cup S$. By definition this implies that $R_P \cup S \mathrel{|\!\!\sim} \Psi \vdash e\colon \tau$ for some Ψ and τ. To prove that $O[e]$ is typable by R it suffices to prove that $R_P \cup S \mathrel{|\!\!\sim} O[\Psi] \vdash O[e]\colon O[\tau]$. We will prove the more general statement that if $R_P \cup S \mathrel{|\!\!\sim} \Phi$ then $R_O \cup S \mathrel{|\!\!\sim} O[\Phi]$. The proof is by induction on the size of the derivation of Φ using rules in $R_P \cup S$.

First, suppose the last inference in the derivation uses rule P3 (the rule for typing procedure abstractions) to derive $\Psi \vdash \mathtt{proc}(\lambda x.e)\colon \sigma \to \tau$. Condition

4 in the definition of normality implies there exists a derivation of the same length, and using P3 as the final rule, of the unique sequent of the form

$$T\ y_1 \colon \gamma_1 \wedge \ldots \wedge y_n \colon \gamma_n \ \vdash \ \mathtt{proc}(\lambda x.e) \colon \sigma \to \tau$$

where $y_1, \ldots, y_n$ are the free variables of $\lambda x.e$ and where Ψ contains these bindings. Since this sequent is derived using P3 we that have there exists a smaller derivation of

$$T \wedge y_1 \colon \gamma_1 \wedge \ldots \wedge y_n \gamma_n \wedge x \colon \sigma \ \vdash \ e \colon \tau$$

By the induction hypothesis we then have that the following is derivable using $R_O \cup S$.

$$T \wedge y_1 \colon O[\gamma_1] \wedge \ldots \wedge y_n \colon O[\gamma_n] \wedge x \colon O[\sigma] \ \vdash \ O[e] \colon O[\tau]$$

We will now apply the inference rule O3 to this sequent. Condition 2 and the definition of a well-formed antecedent imply that the side conditions of O3 are satisfied. So $R_O \cup S$ proves the following.

$$c_{\mathtt{proc}(\lambda x.e)}(O[\gamma_1], \ \ldots, \ O[\gamma_n]) \colon\colon \mathtt{apply\_message}().O[\gamma] \to O[\sigma]$$

Since Ψ contains the bindings $y_i \colon \gamma_i$ we have that $O[\Psi]$ contains the bindings $y_i \colon O[\gamma_i]$. By condition 1 S contains the rule S0 and hence we have that $R_O \cup S$ derives $O[\Psi] \ \vdash \ y_i : O[\gamma_i]$. Hence by rule O3$'$ we have that $R_O \cup S$ derives $O[\Psi] \ \vdash \ c_{\mathtt{proc}(\lambda x.e)}(y_1, \ \ldots, \ y_n) \colon O[\tau]$ which is the desired formula.

The cases where the final inference rule is P1, P2, or P4 is similar to, but simpler than, the case for P3. A use of P1 translates to use of O1; a use of P2 translates to a use of O2; and a use of P4 translates to a use of O4. Finally, consider a derivation whose last step uses an instance $\downarrow\xi[B] \Rightarrow \downarrow\xi[H]$ of a rule $B, C \to H$ in S. Let $O[\xi]$ be the substitution mapping x to $O[\xi(x)]$. We will first show that the substitution $O[\xi]$ satisfies the side conditions C. Consider a side condition $x \notin FV[e]$ in C. Condition 7 guarantees that e does not mention `proc`, `apply_proc`, or $\to$. If a meta-expression e does not mention `proc`, `apply_proc`, or $\to$ then $O[\xi[e]] = O[\xi][e]$, e.g., $O[\xi[f(x)]] = f(O[\xi(x)]) = O[\xi][f(x)]$. We also have that $FV[O[e]] = FV[e]$. Hence $FV[O[\xi][e]] = FV[O[\xi[e]]] = FV[\xi[e]]$. Since we have $x \notin FV[\xi[e]]$ we must also have $x \notin FV[O[\xi][e]]$ and hence $O[\xi]$ satisfies the side conditions. Now consider the instance of this same rule that can be written as $\downarrow O[\xi][B] \to \downarrow O[\xi][H]$. By the preceding argument this instance can be written as $\downarrow O[\xi[B]] \to \downarrow O[\xi[H]]$. Condition 9 states that for every expression of the form $\mathtt{mlet}(\lambda x.w, \ u)$ in $\xi[H]$ or $\xi[B]$ we have that w does not contain `proc`. For any expression e with this property we have that $\downarrow O[e] = O[\downarrow e]$. So this instance of the rule can be rewritten as $O[\downarrow\xi[B]] \Rightarrow O[\downarrow\xi[H]]$. By the induction hypothesis each assertion in $O[\downarrow\xi[B]]$ is derivable by $R_O \cup S$ and hence $O[\downarrow\xi[H]]$ is also derivable. This is the desired formula. This finishes the proof that typability of e implies typability of $O[e]$.

Now we show the converse — that if $O[e]$ is typable by $R_O \cup S$ then e is typable by $R_P \cup S$. First we define an inverse of object conversion. More specifically, the operation O^{-1} is defined on an arbitrary object-level expression by the following equations.

$O^{-1}[x] = x$
$O^{-1}[c_{\mathtt{proc}(\lambda x.e)}(u_1, \ldots, u_n)] = \mathtt{proc}(\lambda x.e)[u_1/y_1, \ldots, u_n/y_n]$
$O^{-1}[\mathtt{apply\_message}().\sigma \rightharpoonup \tau] = O^{-1}[\sigma] \rightarrow O^{-1}[\tau]$
$O^{-1}[c_{\mathtt{proc}(\lambda x.e)}(\sigma_1, \ldots, \sigma_n){::}\, \tau] = T \ \wedge y_1{:}\, O^{-1}[\sigma_1] \wedge \ldots \wedge y_n{:}\, O^{-1}[\sigma_n] \vdash \mathtt{proc}(\lambda x.e){:}\, \tau$
$O^{-1}[f(e_1, \ldots, e_n)] = f(O^{-1}[e_1], \ldots, O^{-1}[e_n])$
$O^{-1}[g(\lambda x.e_1, e_2, \ldots, e_n)] = g(\lambda x.O^{-1}[e_1], O[e_2], \ldots, O^{-1}[e_n])$

In the second and fourth equation $y_1, \ldots, y_n$ must be the free variables of $\lambda x.e$. The last two equations are used when none of the first four apply. It is easy to show by structural induction on expressions that if e is a pure lambda term then $O^{-1}[O[e]] = e$. Furthermore, for any Φ, e and τ we have $O^{-1}[\Phi \vdash e{:}\, \tau]$ equals $O^{-1}[\Phi] \vdash O^{-1}[e]{:}\, O^{-1}[\tau]$. We define a derivation to be *pure* if the only object constructors appearing in the derivation are closure constructors and the only message name appearing in the derivation is `apply_message`. By condition 8 we have that if $O[e]$ is typable by $R_O \cup S$ then there exists a pure derivation of a sequent of the form $\Psi \vdash e{:}\, \tau$. We now show by induction on derivation size that if there exists a pure derivation of Φ using $R_O \cup S$ then there exists a derivation of $O^{-1}[\Phi]$ using $R_P \cup S$. The basic idea is that uses of O1, O2, O3, and O4 are translated to uses of P1, P2, P3, and P4 respectively. The most difficult cases are are when the final inference is a use of O3 or O3''. Suppose the final inference uses an instance of O3$'$. Since the derivation is pure, the derived sequent must be of the form $\Psi \vdash c_{\mathtt{proc}(\lambda x.e)}(e_1, \ldots e_n){:}\, \tau$. Since this assertion is derived by O3$'$, there exists a smaller derivations of $\Phi \vdash e_1{:}\, \sigma_1$, $\ldots\ \Phi \vdash e_n{:}\, \sigma_n$, and $c_{\mathtt{proc}(\lambda x.e)}(\sigma_1, \ldots, \sigma_n){::}\, \tau$. By the induction hypothesis there exists a derivation using $R_P \cup S$ of $O^-1[\Psi] \vdash O^{-1}[e_1]{:}\, O^{-1}[\sigma_1]$, $\ldots$ $O^{-1}[\Phi] \vdash O^{-1}[e_1]{:}\, O^{-1}[\sigma_1]$ and the following.

$$T \wedge y_1{:}\, O^{-1}[\sigma_1] \wedge \ldots \wedge y_n{:}\, O^{-1}[\sigma_n] \vdash \mathtt{proc}(\lambda x.e){:}\, \tau$$

Now by condition 5 of normality we have that there exists an $R_P \cup S$ derivation of $O^{-1}[\Psi] \vdash \mathtt{proc}(\lambda x.e)[O^{-1}[e_1]/y_1, \ldots, O^-1[e_n]/y_n]{:}\, \tau$ which can be written as $O^{-1}[\Phi] \vdash O^{-1}[c_{\mathtt{proc}(\lambda x.e)}(e_1, \ldots, e_n)]{:}\, O^{-1}[\tau]$ which is the desired formula.

Now suppose the final rule is O3. Since the derivation is pure and since O3 requires that the f_i be distinct and that $m \geq 1$ we have that the conclusion must be of the following form.

$$c_{\mathtt{proc}(\lambda x.e)}(\gamma_1, \ldots, \gamma_n){::}\, \mathtt{apply\_message}().\sigma \rightharpoonup \tau$$

By the induction hypothesis we have that the following can be derived using $R_P \cup S$.

$$T \wedge y_1{:}\, O^{-1}[\gamma_1] \wedge \ldots \wedge y_n{:}\, O^{-1}[\gamma_m] \wedge x{:}\, O^{-1}[\sigma]$$

We now use rule P3 on this sequent. To do this we must check that the side condition on rule P3 is satisfied. The operation O^{-1} does not alter the free variables of an expression not containing "::". So by condition 10 we have that O^{-1} does not alter the free variables of the values of meta-variables so the side conditions on rule O3 now imply that the side conditions on rule P3. So by rule P3 we have that there exists a derivation using $R_P \cup S$ of the following.

$$T \wedge y_1\colon O^{-1}[\gamma_1] \wedge \ldots \wedge y_n\colon O^{-1}[\gamma_m] \vdash \texttt{proc}(\lambda x.e)\colon O^{-1}[\sigma] \rightarrow O^{-1}[\tau]$$

This can be rewritten as follows.

$$T \wedge y_1\colon O^{-1}[\gamma_1] \wedge \ldots \wedge y_n\colon O^{-1}[\gamma_m] \vdash \texttt{proc}(\lambda x.e)\colon O^{-1}[\texttt{apply\_message}().\sigma \rightharpoonup \tau]$$

This is the desired sequent and we are done with this case.

The cases where the final rules are O1, O2, or O4 are similar but simpler. A use of O1 translates to a use of P1; a use of O2 to a use of P2 and a use of O4 to a use of P4. A use of a rule in S translates to a use of that same rule. The proof is directly analogous to the corresponding proof for the in showing that typability of e implies typability of $O[e]$. Condition 7 plays the same role here as in the previous proof — O^{-1} commutes with all constructors other than $\rightharpoonup$, "::!", object constructors, or message names. Unlike object conversion, inverse object conversion commutes with let-reduction in general, i.e., $O^{-1}[\downarrow e] = \downarrow O^{-1}[e]$. This implies that no condition analogous to condition 9 is required for $R_O \cup S$ derivations. However, to ensure that the side conditions of rules in S remain satisfied it is important that O^{-1} not change the free variables of the values of meta-variables. This is ensured by condition 10. The proof of the main theorem is now complete.

P1

$$\frac{\texttt{type}(\tau) \quad \texttt{type}(\sigma)}{\texttt{type}(\tau \rightarrow \sigma)}$$

P2

$$\frac{\sigma_2 \leq \sigma_1,\ \tau_1 \leq \tau_2}{\sigma_1 \rightarrow \tau_1 \leq \sigma_2 \rightarrow \tau_2}$$

P3

$$\frac{\Phi \wedge x{:}\ \tau \ \vdash\ e{:}\ \sigma \quad x \notin FV(\Phi,\ \tau,\ \sigma)}{\Phi\ \vdash\ \texttt{proc}(\lambda x.e){:}\ \tau \rightarrow \sigma}$$

P4

$$\frac{\Phi\ \vdash\ e{:}\ \tau \quad \Phi\ \vdash\ w{:}\ \tau \rightarrow \sigma}{\Phi\ \vdash\ \texttt{apply\_proc}(w,\ e){:}\ \sigma}$$

Figure 1: Fundamental inference rules for procedures.

S1 $$\frac{\texttt{type}(\tau_1),\ \ldots,\ \texttt{type}(\tau_n) \qquad x_1,\ \ldots,\ x_n \notin FV(\tau_1,\ \ldots,\ \tau_n)}{T \wedge x_1{:}\ \tau_1 \wedge \ldots \wedge x_n{:}\ \tau_n \ \vdash\ x_i{:}\ \tau_i}$$

S2 $$\frac{}{\texttt{type}(\alpha)}$$

S3 $$\frac{\texttt{type}(\sigma \cap \tau) \qquad \gamma \leq \sigma \qquad \gamma \leq \tau}{\gamma \leq \sigma \cap \tau}$$

S4 $$\frac{\gamma \leq \sigma \cap \tau}{\gamma \leq \sigma \qquad \gamma \leq \tau}$$

S5 $$\frac{\Phi \ \vdash\ e{:}\ \sigma \qquad \sigma \leq \tau}{\Phi \ \vdash\ e{:}\ \tau}$$

S6 $$\frac{\texttt{type}(\sigma)}{\sigma \leq \sigma}$$

S7 $$\frac{\texttt{type}(\sigma),\ \texttt{type}(\tau)}{\texttt{type}(\sigma \cap \tau),\ \texttt{type}(\bot),\ \texttt{type}(\top),\ \texttt{type}(\forall\alpha.\sigma)}$$

S8 $$\frac{\texttt{type}(\sigma)}{\sigma \leq \top,\ \bot \leq \sigma}$$

S9 $$\frac{\Phi \ \vdash\ e{:}\ \tau,\ \ x \notin FV(\Phi)}{\Phi \ \vdash\ e{:}\ \forall\alpha.\tau}$$

S10 $$\frac{\Phi \ \vdash\ e{:}\ \forall\alpha.\tau \qquad \texttt{type}(\sigma)}{\Phi \ \vdash\ e{:}\ \tau[\sigma/\alpha]}$$

Figure 2: Optional inference rules. In rule S1 we require that $x_1,\ \ldots,\ x_n$ be distinct. This is done by making S1 a schema for an infinite set of SEF rules each of which involves specific object-level variables. In rule S2 α ranges over variables, i.e., it is a variable meta-variable. Rule S7 represents four different rules — any subset of these four rules may be included in a normal rule set.

O1

$$\frac{\begin{array}{l}\texttt{type}(\sigma_{1,1}),\ \ldots,\ \texttt{type}(\sigma_{1,k_1}),\ \texttt{type}(\tau_1)\\ \vdots\\ \texttt{type}(\sigma_{m,1}),\ \ldots,\ \texttt{type}(\sigma_{m,k_m}),\ \texttt{type}(\tau_m)\end{array}}{\texttt{type}(\ f_1().\sigma_{1,1}\times\ldots\times\sigma_{1,k_1}\rightharpoonup\tau_1\ \ \cap\ \ \ldots\ \ \cap\ \ f_m().\sigma_{m,1}\times\ldots\times\sigma_{m,k_m}\rightharpoonup\tau_m\ )}$$

O2

$$\frac{\begin{array}{l}\sigma_1\le\gamma_1,\ \ldots,\ \sigma_n\le\gamma_n\\ \tau\le\omega\end{array}}{f().\gamma_1\times\ldots\times\gamma_n\rightharpoonup\tau\le f().\sigma_1\times\ldots\times\sigma_n\rightharpoonup\omega}$$

O3

$$\frac{\begin{array}{l}T\ \wedge\ y_1{:}\,\gamma_1\ \wedge\ldots\wedge\ y_n{:}\,\gamma_n\ \wedge\ldots\wedge\ x_{1,1}{:}\,\sigma_{1,1}\ \wedge\ldots\wedge\ x_{1,k_1}{:}\,\sigma_{1,k_1}\ \vdash\ e_1{:}\,\tau_1\\ \vdots\\ T\ \wedge\ y_1{:}\,\gamma_1\ \wedge\ldots\wedge\ y_n{:}\,\gamma_n\ \wedge\ldots\wedge\ x_{m,1}{:}\,\sigma_{m,1}\ \wedge\ldots\wedge\ x_{m,k_1}{:}\,\sigma_{m,k_1}\ \vdash\ e_m{:}\,\tau_m\\ y_1,\ \ldots,\ y_n,\ x_{1,1},\ \ldots,\ x_{m,k_m}\notin FV(\gamma_1,\ \ldots,\gamma_n,\ \sigma_{1,1},\ldots,\ \sigma_{m,k_m},\ \tau_1,\ \ldots,\ \tau_m)\end{array}}{c(\gamma_1,\ \ldots,\ \gamma_n){::}\ f_1().(\sigma_{1,1}\times\ldots\times\sigma_{1,n_1})\rightharpoonup\tau_1\ \ \cap\ \ \ldots\ \ \cap\ \ f_m().\sigma_{m,1}\ \times\ldots\times\sigma_{m,n_m}\rightharpoonup\tau_m}$$

O3′

$$\frac{\begin{array}{l}\Phi\ \vdash\ e_1{:}\,\sigma_1,\ \ldots,\ \Phi\ \vdash\ e_n{:}\,\sigma_n\\ c(\sigma_1,\ \ldots,\ \sigma_n){::}\ \tau\end{array}}{\Phi\ \vdash\ c(e_1,\ \ldots,\ e_n){:}\,\tau}$$

O4

$$\frac{\begin{array}{l}\Phi\ \vdash\ e_1{:}\,\sigma_1,\ \ldots,\ \Phi\ \vdash\ e_k{:}\,\sigma_k\\ \Phi\ \vdash\ u{:}\ f().\sigma_1\times\ldots\times\sigma_k\rightharpoonup\tau\end{array}}{\Phi\ \vdash\ f(u,\ e_1,\ \ldots,\ e_k){:}\,\tau}$$

Figure 3: Fundamental inference rules for objects. In the rules O1 and O3 we require that $f_1,\ \ldots,\ f_n$ be distinct message names and $m \geq 1$. The notation $f_i()$ abbreviates $f_i(\texttt{dummy},\ \ldots,\ \texttt{dummy})$ — a term used to represent f_i itself. In rule O3 we must have that the method definitions for object constructor c are the rewrite rules $f(c(y_1,\ \ldots,\ y_n),\ x_{1,1},\ \ldots,\ x_{1,k_1}) \to e_1,\ \ldots,$ $f(c(y_1,\ \ldots,\ y_n),\ x_{m,1},\ \ldots,\ x_{m,k_m}) \to e_m$. Note that object constructors and message names are not typable — no variable can have value $f()$ or $c()$.