Lisp and Symbolic Computation, 10, 189–199 (1998)

The Theory of Fexprs is Trivial *

MITCHELL WAND wand@ccs.neu.edu
College of Computer Science, Northeastern University, 360 Huntington Avenue, 161CN, Boston, MA 02115, USA

Abstract. We provide a very simple model of a reflective facility based on the pure λ-calculus, and we show that its theory of contextual equivalence is trivial: two terms in the language are contextually equivalent iff they are α-congruent.

1. Introduction

The thesis of much of programming language semantics is that the fundamental question about a programming language is its notion of *contextual equivalence*: which pairs of phrases (M, N) have the property that M and N may be freely substituted for each other in any program context, without changing the behavior of the resulting program [10, 14]. This is a key notion because, for example, any source-to-source optimization in a compiler (except possibly for a whole-program transformation) should produce a term contextually equivalent to the original.

In this note, we provide a very simple model of a reflective facility based on the pure λ-calculus, and we show that its theory of contextual equivalence is trivial: two terms in the language are contextually equivalent iff they are α-congruent.

Our language contains a primitive reflective capability through a **fexpr** construct. The **fexpr** construct creates a procedure which, when called, reifies its unevaluated actual parameter as a λ-term in normal form using the Mogensen-Scott coding [9]. The behavior of the language is characterized using a simple reduction semantics.

To characterize the contextual equivalence, we observe that α-congruence is a bisimulation of the reduction relation, and then we construct, for any two non-congruent terms, a context that will distinguish them.

Since all "fully reflective" languages (e.g. 3-Lisp [15], Brown [18]) have even higher degrees of introspection than our simple language, we conclude that these languages similarly have trivial equational theories.

Our operational techniques contrast with the usual treatment of reflective languages, which have traditionally been studied either using the metaphor of an infinite tower of interpreters [15, 4], or via a denotational semantics using meta-continuations or the like [18, 3]. These studies have largely been descriptive and have not offered any theorems. An exception is [5], which also uses operational techniques and which includes a Church-Rosser theorem. [7] uses operational techniques somewhat similar to ours.

---

* Work supported by the National Science Foundation under grants numbered CCR-9304144, CCR-9404646, and CCR-9629801.

Our result extends an old observation of Albert Meyer [8] that **eval** and **quote** render contextual equivalence trivial; we show that the same phenomenon arises in a simple, well-defined system without any assumptions about the data in the language.

The remainder of the paper is organized as follows: Section 2 describes our mini-language and its operational semantics. Section 3 presents the main technical results. Section 4 discusses some of the ways in which the results might be extended to systems with environments, including traditional Lisp fexprs. Section 5 discusses the implications of our results for the existence of denotational models of reflective languages. Finally, Section 6 suggests some conclusions.

2. The Language

Our goal is to give the simplest possible system in which the reflection of expressions makes sense. Thus our system has no types, no lists or other data structures, and is given by a λ-calculus-like substitution semantics rather than using environments. Some of the issues raised by these additional features are addressed in Section 4.

The language is given by the following grammar:

$$M, N ::= x \mid \lambda x.M \mid (M\ N) \mid (\mathbf{fexpr}\ M) \mid (\mathbf{eval}\ M) \quad \text{terms}$$

Except as otherwise noted, "terms" will refer to the terms of this language. We will omit parentheses whenever convenient. α-congruence is defined in the obvious way.

Our formulation is based on the principle that there should be only one binding operator in the language. Therefore, we write $(\mathbf{fexpr}\,(\lambda x.M))$ in place of the more familiar `(fexpr (`x`)` M`)`.

Although this language has no data structures, we can simulate data structures using one or another of the standard encodings of the pure λ-calculus. For concreteness, we encode expressions using a Mogensen-Scott encoding [9, 16], here denoted $\lceil - \rceil$, which represents each term as a pure λ-term in normal form that behaves like a **case** statement:[1]

$$\begin{array}{ll}
\lceil x \rceil & \equiv \lambda abcde.ax \\
\lceil MN \rceil & \equiv \lambda abcde.b\lceil M\rceil\lceil N\rceil \\
\lceil \lambda x.M \rceil & \equiv \lambda abcde.c(\lambda x.\lceil M\rceil) \\
\lceil (\mathbf{fexpr}\ M) \rceil & \equiv \lambda abcde.d\lceil M\rceil \\
\lceil (\mathbf{eval}\ M) \rceil & \equiv \lambda abcde.e\lceil M\rceil
\end{array}$$

where a, b, c, d, and e are fresh variables in each case.

This coding has the following properties:

LEMMA 1 *1. For any term M, $\lceil M\rceil$ is a term of the pure λ-calculus in normal form.*

2. $fv(\lceil M\rceil) = fv(M)$.

3. $\ulcorner M \urcorner =_\beta \ulcorner N \urcorner$ *iff* $M \equiv_\alpha N$.

4. For any term M it is decidable whether or not there is a term N such that $M \equiv_\alpha \ulcorner N \urcorner$; if so, N is unique up to α-congruence and can be found effectively from M.

Proof: All easy. ■

Now we can formulate a reduction semantics for the closed terms of this language, using Felleisen's notion of reduction contexts:

$$\begin{array}{lll} R ::= [\,] \mid (R\ M) \mid ((\lambda x.M)\ R) \mid (\mathbf{fexpr}\ R) \mid (\mathbf{eval}\ R) & \text{reduction contexts} \\ V ::= \lambda x.M \mid (\mathbf{fexpr}\ V) & \text{values} \\ I\ ::= ((\lambda x.M)\ V) \mid ((\mathbf{fexpr}\ V)\ M) \mid (\mathbf{eval}\ V) & \text{redices (instructions)} \end{array}$$

The reduction contexts tell us where in a term a reduction may occur: we can reduce at the top level, or in the operator position of an application. If the operator has been reduced to an abstraction, then we can reduce the operand. Both **fexpr** and **eval** evaluate their arguments. This uniquely specifies the location of the redex:

LEMMA 2 *Every closed term M is either a value or is uniquely decomposable in the form $R[\,I\,]$, where R is an reduction context and I is a redex.*

We now state the reduction rules:

$$\begin{array}{lll} R[\,((\lambda x.M)\ V)\,] & \to R[\,M[V/x]\,] & \beta\text{-value} \\ R[\,((\mathbf{fexpr}\ V)\ M)\,] & \to R[\,(V\ \ulcorner M \urcorner)\,] & \text{reification} \\ R[\,(\mathbf{eval}\ \ulcorner M \urcorner)\,] & \to R[\,M\,] & \text{reflection} \end{array}$$

An attempt to apply **eval** to something which is not a coded term will stick; this is decidable by Lemma 1.

The key rule here is the one for reification: when the operator is an abstraction wrapped in a **fexpr**, then the abstraction is applied to the encoding of the *unevaluated* operand. This is why the an M appears in the rule instead of a V'; correspondingly, the rules for reduction contexts do not allow reduction of a term in this position. We consider a number of variants of this system in Section 4.

The converse of reification is reflection, represented here by the **eval** special form. The term $\lambda x.\,\mathbf{eval}\,x$ serves as an evaluator for this language, since $(\lambda x.\,\mathbf{eval}\,x)\ulcorner M \urcorner \to M$ by definition. The terms $(\mathbf{eval}\ M)$ do not play a role in our construction, but are needed to make a plausible model of reflection.[2]

Since this system is an extension of the call-by-value lambda-calculus, we can use many of the familiar coding techniques. We code booleans by letting T denote $\lambda xy.x$ and F denote $\lambda xy.y$; then we can code the conditional as application, as usual. The Church numerals $\ulcorner n \urcorner$ can serve as a numeral system; in particular there is a term $Eqnum$ such that $Eqnum\ulcorner n \urcorner \ulcorner n \urcorner \to^* T$ but $Eqnum\ulcorner n \urcorner \ulcorner m \urcorner \to^* F$ for $n \neq m$. $\Omega = (\lambda x.xx)(\lambda x.xx)$ is a non-terminating term in this system.

3. Results

We begin by defining our notions of contextual equivalence. We define contexts as follows:

$$C, C' ::= [\,] \mid x \mid \lambda x.C \mid (C\,C') \mid (\textbf{fexpr}\ C) \mid (\textbf{eval}\ C) \quad \text{contexts}$$

Definition 1. M *contextually approximates* N ($M \lesssim N$) iff for every context C, if $C[M]$ reduces to a value then $C[N]$ reduces to a value.

Definition 2. M and N are *contextually equivalent* ($M \cong N$) iff for every context C, $C[M]$ reduces to a value iff $C[N]$ reduces to a value.

Reduction contexts $R[\,]$ define those contexts in which a program phrase is "ready to run," but contexts C include *all* contexts, including those that might save multiple copies of a procedure, pass it around in the program, run it multiple times, etc. Contexts also bind the free variables of the term in the hole $[\,]$: If C denotes the context $\lambda x.\lambda y.[\,]$, then $C[x]$ denotes the term $\lambda x.\lambda y.x$. Contextually equivalent program fragments are "plug-compatible": they may be interchanged freely in *any* context. Thus contextual equivalence is the minimal goal for local source-to-source compiler transformations: an optimized expression should behave the same way as the original, no matter how it is used in the larger program.

LEMMA 3 *If $M \equiv_\alpha N$, then M is a value iff N is a value.*

Proof: Immediate. ■

LEMMA 4 *If $M \equiv_\alpha N$ and $M \to M'$ and $N \to N'$, then $M' \equiv_\alpha N'$.*

Proof: Immediate. ■

LEMMA 5 *If $M \equiv_\alpha N$, then $M \lesssim N$.*

Proof: If $M \equiv_\alpha N$, then $C[M] \equiv_\alpha C[N]$. If $C[M]$ reduces to a value, then so does $C[N]$, by the preceding two lemmas. ■

We next construct, for any set X of variables and any term M such that $\text{fv}(M) \subseteq X$, a context that will distinguish M from any other term N with $\text{fv}(N) \subseteq X$.

We first fix some notation. Let *Vars* denote the set of variables, and let σ range over numeral-valued substitutions, that is, finite partial maps from *Vars* to Church numerals.

LEMMA 6 *Let $X \subseteq Vars$, let M be any term with $fv(M) \subseteq X$, and σ be any injective numeral-valued substitution. Then there exists a term $Eq_{M,\sigma}$ such that*

$$Eq_{M,\sigma}(\lceil M \rceil \sigma) \to^* T, \quad \textit{but}$$
$$Eq_{M,\sigma}(\lceil N \rceil \sigma) \to^* F$$

$$
\begin{array}{lll}
Eq_{x,\sigma} & \equiv \lambda m.m & \\
 & & (\lambda v.\,Eqnum\ v\ (\sigma_x)) \\
 & & (\lambda mn.F) \\
 & & (\lambda f.F) \\
 & & (\lambda m.F) \\
 & & (\lambda m.F)
\end{array}
$$

$$
\begin{array}{lll}
Eq_{(M_1 M_2),\sigma} & \equiv \lambda m.m & \\
 & & (\lambda v.F) \\
 & & (\lambda m_1 m_2.(Eq_{M_1,\sigma} m_1)(Eq_{M_2,\sigma} m_2)F) \\
 & & (\lambda f.F) \\
 & & (\lambda m.F) \\
 & & (\lambda m.F)
\end{array}
$$

$$
\begin{array}{lll}
Eq_{\lambda x.M,\sigma} & \equiv \lambda m.m & \\
 & & (\lambda v.F) \\
 & & (\lambda mn.F) \\
 & & (\lambda f.\,Eq_{M,\sigma[x\mapsto \mathbf{c}]}(f\mathbf{c})) \\
 & & (\lambda m.F) \\
 & & (\lambda m.F)
\end{array}
$$

$$
\begin{array}{lll}
Eq_{\mathbf{fexpr}\,M,\sigma} & \equiv \lambda m.m & \\
 & & (\lambda v.F) \\
 & & (\lambda mn.F) \\
 & & (\lambda f.F) \\
 & & (\lambda m.\,Eq_{M,\sigma} m) \\
 & & (\lambda m.F)
\end{array}
$$

$$
\begin{array}{lll}
Eq_{\mathbf{eval}\,M,\sigma} & \equiv \lambda m.m & \\
 & & (\lambda v.F) \\
 & & (\lambda mn.F) \\
 & & (\lambda f.F) \\
 & & (\lambda m.F) \\
 & & (\lambda m.\,Eq_{M,\sigma} m)
\end{array}
$$

Figure 1. Definition of $Eq_{M,\sigma}$

for any N *with* $fv(N) \subseteq X$ *and* $M \not\equiv_\alpha N$.

Proof: We construct $Eq_{M,\sigma}$ by induction on M. The definition is shown in Figure 1. There σ_x denotes the Church numeral that is the image of x under σ. In the case for $\lambda x.M$, $\mathbf{c}$ denotes a Church numeral not in the range of σ.

To show that this term behaves as specified, we proceed by induction on M, and then by cases on N.

Consider first $Eq_{x,\sigma}(\lceil N \rceil \sigma)$. If N is anything other than a variable, $Eq_{x,\sigma}(\lceil N \rceil \sigma) \to^* F$. If N is the variable x, then

$$Eq_{x,\sigma}(\lceil N \rceil \sigma) \to^* (Eqnum\, \sigma_x \sigma_x) \to^* T$$

If N is some other variable y, then

$$Eq_{x,\sigma}(\lceil N \rceil \sigma) \to^* (Eqnum\, \sigma_y \sigma_x) \to^* F$$

since σ is assumed to be injective.

For the $\lambda x.M$ case, we assume without loss of generality that the bound variable of the argument is x. Then we calculate:

$$\begin{array}{l}
Eq_{\lambda x.M,\sigma}(\lceil \lambda x.N \rceil \sigma) \\
\equiv_\alpha Eq_{\lambda x.M,\sigma}((\lambda abcde.c(\lambda x.\lceil N \rceil))\sigma) \\
\equiv_\alpha Eq_{\lambda x.M,\sigma}(\lambda abcde.c(\lambda x.(\lceil N \rceil \sigma))) \\
\to^* (\lambda abcde.c(\lambda x.(\lceil N \rceil \sigma)))(\ldots)(\ldots)(\lambda f.\, Eq_{M,\sigma[x\mapsto \mathbf{c}]}(f\ \mathbf{c}))(\ldots)(\ldots) \\
\to^* (\lambda f.\, Eq_{M,\sigma[x\mapsto \mathbf{c}]}(f\ \mathbf{c}))(\lambda x.(\lceil N \rceil \sigma)) \\
\to^* Eq_{M,\sigma[x\mapsto \mathbf{c}]}((\lambda x.(\lceil N \rceil \sigma))\mathbf{c}) \\
\to^* Eq_{M,\sigma[x\mapsto \mathbf{c}]}((\lceil N \rceil \sigma)[\mathbf{c}/x]) \\
\equiv_\alpha Eq_{M,\sigma[x\mapsto \mathbf{c}]}(\lceil N \rceil (\sigma[\mathbf{c}/x]))
\end{array}$$

which reduces to T or F depending on whether M and N are α-congruent; the use of the induction hypothesis depends on the observation that if σ is injective, and $\mathbf{c}$ is a numeral not in the range of σ, then $\sigma[\mathbf{c}/x]$ is injective also.

The remaining cases are similar. In the application case, the line

$$(Eq_{M_1,\sigma} m_1)(Eq_{M_2,\sigma} m_2)F$$

encodes the conjunction of $(Eq_{M_1,\sigma} m_1)$ and $(Eq_{M_2,\sigma} m_2)$, relying on the coding of truth and falsity as $\lambda xy.x$ and $\lambda xy.y$. ■

LEMMA 7 *If $M \not\equiv_\alpha N$, then $M \not\lesssim N$.*

Proof: Let $\{x_1, \ldots, x_n\}$ be a set of variables including the free variables of M and N. Let C be the context

$$(\mathbf{fexpr}\ (\lambda x.\, Eq_{\lambda x_1 \ldots x_n.M,\emptyset}\ x\ (\lambda x.\ T)\ (\lambda x.\Omega)\ (\lambda z.z)))(\lambda x_1 \ldots x_n.[\ ])$$

To see how this context does the job, we consider the reduction of $C[P]$ for an arbitrary P with $\mathrm{fv}(P) \subseteq \{x_1 \ldots x_n\}$:

$$\begin{array}{l}
(\mathbf{fexpr}\ (\lambda x.\, Eq_{\lambda x_1 \ldots x_n.M,\emptyset}\ x\ (\lambda x.\ T)\ (\lambda x.\Omega)\ (\lambda z.z))(\lambda x_1 \ldots x_n.P) \\
\to Eq_{\lambda x_1 \ldots x_n.M,\emptyset}\ \lceil (\lambda x_1 \ldots x_n.P) \rceil\ (\lambda x.\ T)\ (\lambda x.\Omega)(\lambda z.z) \\
\to \begin{cases} (\lambda x.\ T)(\lambda z.z) & \text{if } \lambda x_1 \ldots x_n.P \equiv_\alpha \lambda x_1 \ldots x_n.M \\ (\lambda x.\Omega)(\lambda z.z) & \text{otherwise} \end{cases}
\end{array}$$

which either reduces to T or loops, depending on whether or not $\lambda x_1 \ldots x_n.P \equiv_\alpha \lambda x_1 \ldots x_n.M$. Thus $C[M]$ converges, but $C[N]$ loops. So $M \not\lesssim N$. ■

THEOREM 1 $M \lesssim N$ *iff* $M \equiv_\alpha N$.

Proof: Immediate from Lemmas 5 and 7. ∎

COROLLARY 1 $M \cong N$ *iff* $M \equiv_\alpha N$.

Proof: By symmetry of $\equiv_\alpha$. ∎

This corollary is the main result: two terms are contextually equivalent iff they are α-congruent.

4. Environments

Our model differs from Lisp fexprs in that we use a substitution model of evaluation in place of the standard Lisp or Scheme environment model. Adding environments makes the model more complicated in several ways.

First, for an environment model, the simplest arrangement is to use a big-step semantics, with judgements of the form $(M, \rho) \Downarrow (V, \rho')$, where ρ and ρ' denote environments mapping variables to closures, defined by the grammar

$$\rho ::= \emptyset \mid \rho[x \mapsto (V, \rho)]$$

In this system, the **fexpr** rule would be

$$\frac{\begin{array}{l}(M, \rho) \Downarrow ((\mathbf{fexpr}\ (\lambda x.P)), \rho') \\ (P[\lceil N \rceil / x], \rho') \Downarrow (V, \rho'')\end{array}}{((M\ N), \rho) \Downarrow (V, \rho'')}$$

However, in an environment model, our reification strategy is inappropriate. Consider, in the example above, the situation in which N has a free variable y, which is bound by ρ, but $\lambda x.P$ is closed, so ρ' is empty. In our representation, $\lceil N \rceil$ will contain y free, so the subgoal $(P[\lceil N \rceil / x], \rho') \Downarrow (V, \rho'')$ will stick if it refers to y.[3] It is therefore more appropriate to use first-order abstract syntax instead of higher-order abstract syntax. To obtain a first-order coding, we change the variable and abstraction lines of the coding scheme to:

$$\begin{array}{ll}\lceil x \rceil & \equiv \lambda abcde.a\lfloor x \rfloor \\ \lceil \lambda x.M \rceil & \equiv \lambda abcde.c\lfloor x \rfloor \lceil M \rceil\end{array}$$

where $\lfloor x \rfloor$ is some fixed coding of variables into Church numerals. In this representation, $\lceil M \rceil$ is always closed. Furthermore, the identity of variables is detectable in this scheme, so our results would be stronger: two terms are contextually equivalent iff they are identical, not just α-congruent.

Since every coded term is closed, a closed term $\mathbf{eval}\, M$ might reduce to an open term. Hence **eval** should take two arguments: the representation of a term and the representation of an environment (encoded following the grammar given above). The rule for **eval** might be

$$\frac{\begin{array}{l}(M,\rho) \Downarrow (\lceil P \rceil, \rho_1) \\ (N,\rho) \Downarrow (\lceil \rho' \rceil, \rho_2) \\ (P,\rho') \Downarrow (V,\rho'')\end{array}}{((\textbf{eval}\ M\ N),\rho) \Downarrow (V,\rho'')}$$

Here ρ_1 and ρ_2 are irrelevant since $\lceil P \rceil$ and $\lceil \rho' \rceil$ are always closed. As noted above, it is decidable whether a term is an encoding of a term, or of an environment.

Representations $\lceil \rho \rceil$ could be obtained either by explicit construction or by reification using a 2-argument **fexpr** like the one in [6], with a rule like:

$$\frac{\begin{array}{l}(M,\rho) \Downarrow ((\textbf{fexpr}\ V),\rho') \\ (((V\lceil N \rceil)\lceil \rho \rceil),\rho') \Downarrow (V,\rho'')\end{array}}{((M\ N),\rho) \Downarrow (V,\rho'')}$$

5. Denotational Semantics

As far back as 1980, Muchnick and Pleban reported their inability to formulate a denotational definition for "modern LISP" including fexprs [11]. Danvy and Malmkjær [3] report a similar difficulty for the reflective language Blond. Our results sharpen these observations.

In general, a denotational semantics is a map $\mathcal{A}[\![-]\!]$ from program phrases to some set of meanings. We say two phrases M and N are equal in $\mathcal{A}$, $M =_{\mathcal{A}} N$, iff $\mathcal{A}[\![M]\!] = \mathcal{A}[\![N]\!]$.

The requirement that $\mathcal{A}[\![-]\!]$ be "denotational" or "compositional" may be expressed by the requirement that $=_{\mathcal{A}}$ be a congruence, that is, that $M =_{\mathcal{A}} N$ implies $C[\,M\,] =_{\mathcal{A}} C[\,N\,]$ for any context C.

The usual requirement is that the denotational semantics be *adequate* for an operational semantics: that for any closed program M, $M \rightarrow^* c$ iff $M =_{\mathcal{A}} c$, for some class of constants c. In our situation, we have no obvious class of constants c, so we can formulate an even weaker notion of adequacy:

Definition 3. A semantics $\mathcal{A}[\![-]\!]$ is *weakly adequate* iff for any closed terms M and N, if $M =_{\mathcal{A}} N$ and M reduces to a value, then N reduces to a value.

It is a familiar result that contextual equivalence is the largest weakly adequate congruence. This leads to the following corollary:

COROLLARY 2 *If $\mathcal{A}[\![-]\!]$ is any weakly adequate denotational semantics, then $M =_{\mathcal{A}} N$ implies $M \equiv_\alpha N$.*

Proof: If $M =_{\mathcal{A}} N$ and C is any context, then $C[\,M\,] =_{\mathcal{A}} C[\,N\,]$ by compositionality. Hence, by weak adequacy, $C[\,M\,]$ reduces to a value iff $C[\,N\,]$ does so. Therefore $M =_{\mathcal{A}} N$ implies $M \cong N$, which implies $M \equiv_\alpha N$. ■

Hence any weakly adequate denotational semantics must (up to isomorphism) map each term to itself, so no useful abstraction is possible.

A similar result can be obtained for models with an order relation: Define $M \sqsubseteq_{\mathcal{A}} N$ iff $\mathcal{A}[\![M]\!] \sqsubseteq \mathcal{A}[\![N]\!]$. Then the weakest adequacy condition on the semantics is that $M \sqsubseteq_{\mathcal{A}} N \Rightarrow M \lesssim N$. Hence for our language we get $M \sqsubseteq_{\mathcal{A}} N \Rightarrow M \equiv_{\alpha} N$, so no non-trivial orders are possible.

6. Observations and Conclusions

We have presented a very simple model of a language with a simple reflective facility, and showed that this facility makes contextual equivalence trivial. Our language is the simplest model of a reflective facility that we know of: it has no types, no data, and no environments, and yet the reflection is powerful enough to make the resulting theory trivial. In this section we will discuss some of the consequences of this result.

More conventional reflective languages, such as those of [15, 18, 3, 12], reify expressions as list structures, so the construction of distinguishing contexts is even easier, and the same results apply to these languages. Hence a variety of quite different reflective languages have the same equational theory. This suggests that the emphasis on equational theories in classical semantics may be somewhat misplaced: we care not only when two terms have the same behavior, but we may also care just what that behavior is!

To get non-trivial theories, one needs to find weaker reflection principles. For example, reifying continuations via `call/cc` or the equivalent (in the absence of expression reification) leads to a highly non-trivial theory. For example, [7] reifies contexts using operational techniques similar to ours, though he does not specify representations. It would be interesting to see if a similar program could be carried out in order to introduce reflection into a simple object calculus like that of Abadi and Cardelli [1], or to study contextual equivalence in the presence of meta-object protocols.

Types add complexity to the system, but they do not change the result, so long as expressions are reifiable in a form that allows them to be tested for equality.

We must be careful to understand the consequences of our result for the implementation of reflective languages. For example, the result means that there are no valid source-to-source optimizations for our reflective language. It does *not* mean there are no optimizations at all: one could still translate a term into some other optimizable formalism, so long as the translation allowed one to retrieve the original source code for use by **fexpr**. It also does not preclude optimizations based on whole-program analysis: a flow analysis, for example, might give a more refined approximation of the contexts in which a given expression might be executed, allowing for the possibility that an expression might be equivalent to some other expression in all of the contexts in which it is actually executed, even though the two expressions might be distinguished by some unreachable context.

Our result is more widely applicable to conventional languages than it might appear. Many languages have some facility (or misfeature!) that enables one to simulate a **fexpr**. Debuggers or programming environments can distinguish essentially arbitrary programs. If such debuggers are controllable from inside the language, then **fexpr**-like behavior can result, making correct compiler optimizations much more difficult at the very least. This kind of behavior, usually considered unimportant in the programming-language community, could be a central issue if one is considering, say, computer security.

The result reminds us that any consideration of the correctness of program transformations must be preceded by a careful delineation of just which portions of the program's behavior are to be preserved and which parts may be allowed to vary. In complex programming systems, this separation may not be easy to make.

Acknowledgments

We thank the anonymous referees for useful comments.

Notes

1. The ordinary Church numerals represent each number as a recursion operator (a **foldr**). The idea of using a **case**-like representation is credited to Scott by Wadsworth [16, page 217], citing [2]. Mogensen extended this idea to higher-order constructions in [9]. Our results hold for either Church-style or Scott-style encodings; we have chosen the latter because the constructions are a bit simpler.
2. We would have preferred not to have a second special operator in the language. Unfortunately, the self-evaluating terms in [9, 17] fail for the call-by-value λ-calculus: one has $E\lceil M\rceil \rightarrow_{\beta}^{*} M$ but not $E\lceil M\rceil \rightarrow_{\beta_v}^{*} M$. There are two problems: first, the standard fixpoint operator Y always diverges when applied to an argument under call-by-value reduction. Secondly, the interpretation of abstractions, $\lambda m.\lambda v.e(m\,v)$, requires the reduction of $(m\,v)$ under the λv, which is not consistent with call-by-value reduction. Replacing the fixpoint operator with a CBV fixpoint operator [13] solves the first problem but not the second; $E\lceil M\rceil$ and M are then contextually equivalent in pure CBV λ-calculus. Unfortunately, even contextual equivalence fails for our reflective system. Therefore we were forced to include **eval** as an independent operator. We conjecture that it is not possible to write a self-evaluator E for pure λ-terms such that $E\lceil M\rceil$ reduces under CBV to M.
3. This big-step semantics does not distinguish between sticking and non-termination, so it would be more precise to say that there will be no V and ρ'' such that $(P[\lceil N\rceil/x], \rho') \Downarrow (V, \rho'')$.

References

1. Martin Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, 125(2):78–102, March 1996.
2. Haskell B. Curry, R. Feys, and J. P. Seldin. *Combinatory Logic*, volume 2. North-Holland, 1972.
3. Olivier Danvy and Karoline Malmkjær. Intensions and extensions in a reflective tower. In *Proc. 1988 ACM Symposium on Lisp and Functional Programming*, pages 327–341, 1988.
4. Stanley Jefferson and Daniel P. Friedman. A simple reflective interpreter. *Lisp and Symbolic Computation*, 9(2/3):181–202, May/June 1996.
5. Jacques Malenfant, Christophe Dony, and Pierre Cointe. A semantics of introspection in a reflective prototype-based language. *Lisp and Symbolic Computation*, 9(2/3):153–180, May/June 1996.
6. John McCarthy et al. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, MA, 1965.

7. Anurag Mendhekar. *Theoretical Aspects of Reflection in Programming Languages.* PhD thesis, Indiana University, Bloomington, IN, 1998. to appear.
8. Albert R. Meyer. Thirteen puzzles in programming logic. In D. Bjørner, editor, *Proceedings of the Workshop on Formal Software Development: Combining Specification Methods*, Lecture Notes in Computer Science, Berlin, Heidelberg, and New York, May 1984. Springer-Verlag.
9. Torben Æ. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–364, July 1992.
10. James H. Morris, Jr. *Lambda Calculus Models of Programming Languages.* PhD thesis, MIT, Cambridge, MA, 1968.
11. Steven Muchnick and Uwe Pleban. A semantic comparison of LISP and SCHEME. In *Conference Record of the 1980 LISP Conference*, 1980.
12. Robert Muller. M-LISP: A representation-independent dialect of LISP with reduction semantics. *ACM Transactions on Programming Languages and Systems*, 14(4):589–616, October 1992.
13. Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
14. Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
15. Brian C. Smith. Reflection and semantics in Lisp. In *Conf. Rec. 11th ACM Symposium on Principles of Programming Languages*, pages 23–35, 1984.
16. Christopher P. Wadsworth. Some unusual λ-calculus numeral systems. In J. P. Hindley and J. R. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 215–230. Academic Press, New York and London, 1980.
17. Mitchell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):365–387, July 1993. preliminary version appeared in *Conf. Rec. 20th ACM Symp. on Principles of Prog. Lang.* (1993), 137–143.
18. Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. *Lisp and Symbolic Computation*, 1:11–37, 1988. Reprinted in *Meta-Level Architectures and Reflection* (P. Maes and D. Nardi, eds.) North-Holland, Amsterdam, 1988, pp. 111–134.