Incremental Lambda Lifting:
An Exercise in Almost-Denotational Semantics

(ICFP'97 Paper Code: 7553.203)

Gregory T. Sullivan* Mitchell Wand*
College of Computer Science
Northeastern University
360 Huntington Avenue, 161CN
Boston, MA 02115, USA
voice: (617) 373-2462, fax: (617) 373-5121
{gregs,wand}@ccs.neu.edu
http://www.ccs.neu.edu/home/{gregs,wand}

November 17, 1996

Abstract

We prove the correctness of incremental lambda-lifting, an optimization that attempts to reduce the closure allocation overhead of higher-order programs by changing the scope of nested procedures. This optimization is invalid in the standard denotational semantics of Scheme, because it changes the storage behavior of the program. Our method consists of giving Scheme a denotational semantics in an operationally-based term model in which interaction is the basic observable. Lambda lifting is then shown to preserve meaning in the model.

1 Introduction

Compilers for higher-order languages typically perform elaborate program transformations in order to improve performance. Such transformations often change the storage behavior of the program. In [WS97], we introduced a method for proving the correctness of such transformations. The method consists of giving the programs a denotational semantics in an operationally-based term model in which interaction is the basic observable, and showing that the transformation is meaning-preserving. This allows us to consider correctness for programs that interact with their environment without terminating, and also for transformations that change the internal store behavior of the program. In [WS97], we illustrated the method by giving the first proof of the correctness of assignment elimination [KKR$^+$86]. In this paper, we use the method to prove the correctness of incremental lambda lifting as performed by the Twobit Scheme compiler [CH94].

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

Consider the Scheme lambda-expression

$$
\begin{array}{l}
(\mathsf{lambda}\ (x)\ (\mathsf{letrec} \\
\qquad\qquad\qquad ((f_1\ (\mathsf{lambda}\ (y_1)\ e_1)) \\
\qquad\qquad\qquad\ (f_2\ (\mathsf{lambda}\ (y_2)\ e_2))) \\
\qquad\qquad e))
\end{array}
$$

This abstraction is represented by a *closure* consisting of the abstraction's constituent Scheme expressions and any bindings of the abstraction's free variables. Because Scheme is a higher-order language, the closure can be passed as an argument, bound to variables, and stored in the heap. Each time the closure is used – applied to a value for the argument x – closures for the nested procedures f_1 and f_2 must also be created. The environment components of the closures for f_1 and f_2 may reference the outer closure's parameter x, which is in the scope of the nested procedures' bodies.

The *incremental lambda lifting* transformation attempts to reduce the closure allocation overhead by hoisting the nested procedures out of the $\mathsf{lambda}(x)$ abstraction – rewriting the above abstraction to:

$$
\begin{array}{l}
(\mathsf{letrec} \\
\qquad ((f_1\ (\mathsf{lambda}\ (x\ y_1)\ e_1')) \\
\qquad\ (f_2\ (\mathsf{lambda}\ (x\ y_2)\ e_2'))) \\
\quad (\mathsf{lambda}\ (x)\ e'))
\end{array}
$$

The process of moving the nested procedures out of their original scope has a number of consequences. If f_1 and f_2 reference the x parameter of the original outer abstraction, then x must be added as to the argument lists of f_1 and f_2. Also, once x has been added to the argument lists of f_1 and f_2, all call sites for those procedures within e, e_1, and e_2 must be modified to pass the value for x. After lambda-lifting, the closures for f_1 and f_2 are allocated *before* the closure for $\mathsf{lambda}\,(x)$ is created. Thus, even if the closure for $\mathsf{lambda}\,(x)$ is copied and used many times, the overhead of creating closures for f_1 and f_2 is incurred only once.

Proving the correctness of the incremental lambda lifting transformation is challenging in many respects:

1. The transformation itself is complex, involving changes to the scope, signature and bodies of possibly many related procedures.

2. The conditions under which the transformation is allowed are subtle. For example, we cannot lift procedures which may escape the scope we are considering (e.g. if f_1 were returned from e in the above example). Also, we do not want to add extra parameters to the lifted procedures, but we must add all referenced variables and all variables which must be passed to other lifted procedures.

3. The lambda lifting transformation is invalid in the standard denotational semantics of Scheme [CR91], as it can significantly change the storage behavior of programs.

The approach we take is to give Scheme a denotational semantics in an *event-based model*. An event-based model is a term model embedded in a process calculus. Observable behavior is taken from the process calculus; equality of terms in the model is taken to be contextual equivalence as measured by this observable behavior. Equivalence in our semantics does not necessarily imply equivalence in the official denotational semantics of Scheme. In particular, we intend to allow equivalences where the standard semantics would distinguish terms because of differences in non-observable store behavior. Terms which are equivalent in the standard denotational semantics for Scheme, however, must be equivalent under the semantics we use as well.

Following a review of related work in Section 2, we introduce in Section 3 the incremental lambda lifting transformation. We use a dialect of Scheme called Scheme$_{cell}$ as the source language for the transformation. Whereas Scheme variables always denote locations, the Scheme$_{cell}$ dialect distinguishes *value-bound* variables from *cell-bound* (i.e. heap-allocated) variables. Using Scheme$_{cell}$ as the source language lets us ignore any complications from side effects (set!) to local variables which could otherwise invalidate the lambda lifting transformation. A meaning-preserving translation from Scheme to Scheme$_{cell}$ (assignment elimination) was presented and shown correct in [WS97].

Sections 4 and 5 present the semantic framework we will use. In Section 4, we present the syntax and operational semantics of our metalanguage. This metalanguage is an extended PCF with recursive types, a store, and a CPS I/O system. Reduction in the side-effect-free subset of the language is a relation between metalanguage terms. Store and I/O operations, however, take place in term-store pairs we call *states*. The semantics of Scheme$_{cell}$ is specified by a syntax-directed translation into this metalanguage. In Section 5, we define two metalanguage terms to be *contextually equivalent* (denoted $\cong_{\forall C}$) if they have the same I/O behavior when inserted into every possible program context. By identifying input/output as the observable actions of the system, we avoid considering the store as part of the "final answer" of a computation. Indeed, we avoid the notion of a "final answer" at all, so that we can study equivalence of terms that have infinite behavior. Contextual equivalence is the natural notion of equivalence given an operational semantics; however, contextual equivalence is unwieldy and difficult to prove directly. We define two metalanguage terms to be *extensionally equivalent* (denoted $\cong_{ext}$) if they have the have the same I/O behavior in a limited set of contexts we call *extensional contexts*. For example, terms of function type are extensionally equivalent if they produce equivalent results when applied to the same argument. Our extensional contexts are similar to Gordon's *experiments* [Gor95]. From [WS97], we have a context lemma which proves that our metalanguage is extensional – that is, two metalanguage terms are contextually equivalent iff they are extensionally equivalent.

In Section 6, we give define the semantics of Scheme$_{cell}$ expressions via a syntax-directed CPS translation $[\![-]\!]_{cell}$ into the metalanguage.

Given the incremental lambda-lifting transformation LL on Scheme$_{cell}$ expressions, and the translation $[\![-]\!]_{cell}$ from Scheme$_{cell}$ expressions into the metalanguage, Section 7 sketches the proof of correctness for incremental lambda lifting. The proof obligation is to show that for all Scheme$_{cell}$ expressions e,

$$[\![e]\!]_{cell} \cong_{\forall C} [\![LL(e)]\!]_{cell}$$

We first invoke the context lemma so that we need only consider extensional contexts. We then construct a relation between metalanguage states which includes the translations of Scheme$_{cell}$ expressions before and after lambda lifting. The construction of this relation is the technically most difficult part of the proof. We then use a coinductive argument to show that the relation we have constructed is a *bisimulation up to bisimulation* [Mil90], which is easily shown to satisfy the requirements of extensional equivalence. A more detailed presentation of the proof of correctness can be found in [Sul96].

We conclude, in Section 8, with consideration of open problems and future work.

2 Related Work

Operationally-based term models were introduced in [TW96]. Here we extend [TW96] by adding types and a heap, and by considering applications to program transformations. The formulation of input-output both here and in [TW96] is based on Gordon's CPS input-output system [Gor94].

Observational or contextual equivalence for programming languages was introduced by [Plo77], adapting the work of Morris [Mor68] for the λ-calculus. Plotkin's formulation, like almost all other work for sequential languages, dealt with conversion to a constant. Context lemmas for proving contextual equivalence were introduced by Milner [Mil77] in a typed functional language. Gordon [Gor95] extended the result to include recursive types with potentially infinite behavior.

The work most closely related to ours is that of Mason and Talcott [MT91] and Felleisen [Fel87]. They considered contextual equivalence for a number of untyped call-by-value languages using operational techniques, and proved context lemmas for these languages. Our approach differs from theirs in that we propose a single metalanguage into which a variety of source languages can be translated; neither Mason and Talcott nor Felleisen consider issues of translation. Our Context Lemma (Theorem 2) also extends their results by considering input-output effects rather than final answers and by validating full β-conversion rather than their β-value.

Bisimulation was introduced by [Par81]. Abramsky [Abr90] used bisimulation to define a λ-model, but his observables were very different from ours. The concept of a relation preserved under computation is probably much older. It was used for program equivalence in [MT91, WO92, ORW95]; [Sta94] uses a similar notion extended to become a logical relation.

Incremental lambda-lifting in the Twobit Scheme compiler is presented in [CH94]. An overview of lambda lifting in a purely functional setting is given in [PJ87]. Lambda lifting differs from *closure conversion*, such as in [WS94, MMH96]; whereas closure conversion attempts to optimize the representation of closures, lambda lifting attempts change the closure allocation behavior of a program or even to remove the need for some closures entirely. Incremental lambda lifting is also distinct from the *super-combinators* of [Hug82] in that procedures are not necessarily lifted to the top level. The process of incremental lambda lifting can be performed repeatedly in order to bring a deeply-nested procedure to the top level, as would be done with traditional lambda lifting. However, being able to lift only one level at a time gives the compiler added flexibility.

3 Incremental Lambda Lifting

Incremental lambda lifting transforms terms in the Scheme$_{cell}$ dialect of Scheme. The syntax of Scheme$_{cell}$ is given in Figure 1; note that there is no `set!` operator. All mutable variables are stored in heap-allocated cells, and operations on cells are explicit in the Scheme$_{cell}$ language. Side-effects to variables added to the parameter lists of lifted procedures would confound the lambda-lifting transformation in Scheme. This is because the side-effects to the once-free-now-bound variables would, as specified in the Scheme semantics, be modifying a different cell in the lambda-lifted body than in the original. The correctness of a translation from Scheme to Scheme$_{cell}$ was shown in [WS97].

Scheme$_{cell}$ has a simple type-checking system to verify that value-bound and cell-bound variables are used consistently. We omit the rules, but judgements of the form $\Gamma \vdash_{cell} e{:}ok$ are read as "With type assignment Γ mapping the free variables of Scheme$_{cell}$ term e to $\{Val, ref(Val)\}$, e is well-typed."

$$
\begin{array}{rcl}
e & ::= & x \mid n \mid (\mathsf{lambda}\ (x \ldots)\ e) \mid (e\ e \ldots) \\
 & \mid & (\mathsf{let}\ ((x\ rhs) \ldots)\ e) \\
 & \mid & (\mathsf{letrec}\ ((x\ e) \ldots)\ e) \\
 & \mid & (\mathsf{cell\text{-}ref}\ x) \mid (\mathsf{cell\text{-}set!}\ x\ e) \\
rhs & ::= & x \mid (\mathsf{make\text{-}cell}\ e)
\end{array}
$$

Figure 1: Syntax of Scheme$_{cell}$

Given e a Scheme$_{cell}$ expression and $\mathbb{X} = \langle X_1, \ldots, X_n \rangle$ a list of lists of variables

if $e =$	then $LL(\mathbb{X}, e) =$
$(\mathsf{lambda}\ (x_1 \ldots x_{m_0})\ (\mathsf{letrec}$ $((f_1\ (\mathsf{lambda}\ (y_1 \ldots y_{m_1})\ e_1)) \ldots$ $(f_n\ (\mathsf{lambda}\ (y_1 \ldots y_{m_n})\ e_n)))$ $e_0))$	$(\mathsf{letrec}$ $((f_1\ (\mathsf{lambda}\ ([x_i]_{\in X_1} y_1 \ldots y_{m_1})$ $LL_{body}(\langle f_1, \ldots, f_n \rangle, \mathbb{X}, e_1))) \ldots$ $(f_n\ (\mathsf{lambda}\ ([x_i]_{\in X_n} y_1 \ldots y_{m_n})$ $LL_{body}(\langle f_1, \ldots, f_n \rangle, \mathbb{X}, e_n))))$ $(\mathsf{lambda}\ (x_1 \ldots x_{m_0})$ $LL_{body}(\langle f_1, \ldots, f_n \rangle, \mathbb{X}, e_0')))$

Figure 2: The Lambda Lifting Transformation

Figure 2 summarizes the lambda lifting transformation. If e is of the form on the left in Figure 2, and each element X_k in the list of lists $\mathbb{X}$ defines which variables need to be added to the k^{th} local procedure, then $LL(\mathbb{X}, e)$ is the term on the right. The transformation handles any number of mutually recursive nested functions. Each nested function f_k has some number (possibly zero) of the identifiers from $\{x_1, \ldots, x_{m_0}\}$ added to its parameter list, indicated by the "splicing" notation $(f_k\ (\mathsf{lambda}\ ([x_i]_{\in X_k} y_1 \ldots y_{m_k})))$.

The LL_{body} function transforms calls of the lifted local procedure f_k by adding the necessary arguments:

$$LL_{body}(\langle f_1, \ldots, f_n \rangle, \mathbb{X}, (f_k\ e_1 \ldots e_{m_k})) = (f_k\ [x_i]_{\in X_k}\ LL_{body}(F, \mathbb{X}, e_1) \ldots LL_{body}(F, \mathbb{X}, e_{m_k}))$$

The rest of the definition of $\overset{LL_{body}}{\rightsquigarrow}$ is a straightforward recursive descent. Note that the variables $x_i \in X_k$ must be in scope at all transformed call sites $(f_k\ [x_i]_{\in X_k} \ldots)$.

We use the following notation to denote the different sequences of identifiers, where i ranges from 1 to the number of local procedures, n:

$$
\begin{array}{lcll}
X_0 & = & \langle x_1, \ldots, x_{m_0} \rangle & \text{arguments to outer lambda} \\
F & = & \langle f_1, \ldots, f_n \rangle & \text{local procedures} \\
Y_i & = & \langle y_1, \ldots, y_{m_i} \rangle,\ 1 \leq i \leq n & \text{local procedure arguments} \\
X_i & = & \multicolumn{2}{l}{\{x_j \mid x_j \in X_0 \text{ must be added to the argument list of } f_i\}} \\
\mathbb{X} & = & \langle X_1, \ldots, X_n \rangle & \text{list of lists of added arguments} \\
\mathbb{F} & = & \langle (f_1, Y_1), \ldots, (f_n, Y_n) \rangle & \text{local procedure signatures}
\end{array}
$$

We define a Scheme$_{cell}$ term e to be *lambda-liftable with respect to* a list of lists of variables $\mathbb{X}$ iff

1. e is of the form of the left-hand side of Figure 2
2. The variables x_i, y_j, f_k are disjoint, for $1 \leq i \leq m_0$, $1 \leq j \leq max(m_k)$, $1 \leq k \leq n$. Also, all binding constructs within e_0 and the e_k bind identifiers distinct from the x_i, y_j, and f_k.
3. All occurrences of each f_k in $e_0, e_1, \ldots, e_n$ are in operator position, for $1 \leq k \leq n$. This guarantees that no f_k "escapes", and thus that all call sites are known.
4. The list of lists of variables, $\mathbb{X}$, has n elements, $X_1, \ldots, X_n$. Letting k range from 1 to n, and treating each X_k as a set, we require that for each k,
   (a) $X_k \subseteq \{x_1, \ldots, x_{m_0}\}$
   (b) $X_k \supseteq (fv(f_k) \cap X_0) \cup (\bigcup\{X_j \mid f_k \textit{ calls } f_j\})$

We require that, for a given f_k, the arguments to be added are in a solution to a set of constraints, given in Item 4 of the above list. Before lifting f_k outside the scope of lambda $(x_1 \ldots x_{m_0})$, any of the x_i referenced in e_k must be added to f_k's parameter list. More subtly, if f_k calls f_j, i.e. if an application $(f_j\ e_1 \ldots e_{m_j})$ is a subterm of e_k, then the new parameters which must be sent to the lifted f_j must be added (if not already there) to the incoming parameters of f_k. Adding these newly required parameters to f_k may have an effect on the parameters added to any f_i's which, in turn, call f_k – and so on. Eventually a fixed point will be reached. The list of arguments to the outer lambda, $\langle x_1, \ldots, x_{m_0} \rangle$ is always a solution to the set of constraints, though not necessarily the smallest solution.

4 The Metalanguage

In order to prove the correctness of the lambda lifting transformation, we will translate the before-and-after Scheme$_{cell}$ terms into the metalanguage presented in [WS97]. We then show that the translations are equivalent in the operational term model of the metalanguage.

The metalanguage is based on PCF [Plo77], extended with recursive, sum, and product types and with operators for a global store and I/O. Figure 3 presents the syntax and operational semantics of the metalanguage. In [WS97], we show how the recursive types allow the metalanguage to serve as a model for a variety of source languages, including Scheme and Idealized Algol.

The base types include integers *int*, booleans *bool*, processes *pr* and references $ref(\tau)$. Types are possibly infinite trees which respect the grammar given for τ. The typing rules are standard and omitted. A typing judgement is given as $\Gamma \vdash M{:}\tau$ and read as "term M has type τ assuming Γ assigns types to each free variable in M." Location constants l^τ are annotated with the type of

Syntax:

$$\begin{aligned} M &::= x \mid n \mid c \mid l^{\tau} \mid \lambda x{:}\tau.M \mid MM \\ \tau &::= o \mid \tau{\to}\tau \mid \tau{+}\tau \mid \tau{\times}\tau \\ o &::= \mathit{unit} \mid \mathit{bool} \mid \mathit{int} \mid \mathit{ref}(\tau) \mid \mathit{pr} \end{aligned}$$

Store Operations ($\to_{sto}$):

$$\begin{array}{ll} M \to_{pcf} M' \Rightarrow \langle\!\langle M \,;\, \Sigma \rangle\!\rangle \to_{sto} \langle\!\langle M' \,;\, \Sigma \rangle\!\rangle & \\ \langle\!\langle \mathsf{new}^{\tau}\, MN \,;\, \Sigma \rangle\!\rangle \to_{sto} \langle\!\langle N l^{\tau} \,;\, \Sigma[l^{\tau} \mapsto M] \rangle\!\rangle & \text{if } l^{\tau} \notin \mathit{dom}(\Sigma) \cup \mathit{locs}(\mathit{rng}(\Sigma), M, N) \\ \langle\!\langle \mathsf{deref}^{\tau}\, l^{\tau} M \,;\, \Sigma \rangle\!\rangle \to_{sto} \langle\!\langle MN \,;\, \Sigma \rangle\!\rangle & \text{if } \Sigma(l^{\tau}) = N \\ \langle\!\langle \mathsf{update}^{\tau}\, l^{\tau} MN \,;\, \Sigma \rangle\!\rangle \to_{sto} \langle\!\langle N \,;\, \Sigma[l^{\tau} \mapsto M] \rangle\!\rangle & \text{if } l^{\tau} \in \mathit{dom}(\Sigma) \end{array}$$

I/O Operations ($\xrightarrow{a}_{io}$):

$$\frac{\langle\!\langle M \,;\, \Sigma \rangle\!\rangle \to^{*}_{sto} \langle\!\langle \mathsf{write}\, n\, M' \,;\, \Sigma' \rangle\!\rangle}{\langle\!\langle M \,;\, \Sigma \rangle\!\rangle \xrightarrow{!n}_{io} \langle\!\langle M' \,;\, \Sigma' \rangle\!\rangle} \qquad \frac{\langle\!\langle M \,;\, \Sigma \rangle\!\rangle \to^{*}_{sto} \langle\!\langle \mathsf{read}\, M' \,;\, \Sigma' \rangle\!\rangle}{\langle\!\langle M \,;\, \Sigma \rangle\!\rangle \xrightarrow{?n}_{io} \langle\!\langle M'\, n \,;\, \Sigma' \rangle\!\rangle}$$

$$\frac{\langle\!\langle M \,;\, \Sigma \rangle\!\rangle \to^{*}_{sto} \langle\!\langle \mathsf{stop} \,;\, \Sigma' \rangle\!\rangle}{\langle\!\langle M \,;\, \Sigma \rangle\!\rangle \xrightarrow{\surd}_{io} \langle\!\langle \mathsf{stopped} \,;\, \Sigma' \rangle\!\rangle}$$

Figure 3: Syntax and Semantics of the Metalanguage

term they point to, and are given type $\mathit{ref}(\tau)$. PCF reduction, $\to_{pcf}$, is the standard call-by-name relation between terms given in [Plo77] and is omitted.

Store and I/O operations take place in a term-store pair called a *state*. The store component of a state is a finite map from locations to terms. The first rule for $\to_{sto}$ says that any PCF-reduction can take place within a state context. The new operator allocates a new location in the store, initializes the fresh location to its first argument, and invokes its second argument with the new location. The deref operator invokes its second argument with the value in the store at the location given by its first argument. The update operator destructively updates the location given by its first argument to be the term given by its second argument and then invokes its third argument. Note that $\to_{sto}$ transitions may take place only if a store operator is in the leftmost-outermost position. The reflexive, transitive closure of $\to_{sto}$ is denoted $\to^{*}_{sto}$.

The I/O operators are similarly restricted to firing only when they are in the leftmost-outermost position of the term component of a state. The actions emitted by I/O transitions lead us to a natural notion of state equivalence, namely *strong bisimulation*. The definition of bisimulation ($\approx_{bisim}$) is standard [Mil90] and is omitted. Roughly speaking, two states are bisimilar if they can always take identical labeled transitions. $\approx_{bisim}$ can be formulated as the greatest fixed point of an operator $[-]_{bisim}$ on relations.

5 Contextual and Extensional Equivalence

Two terms are *contextually equivalent* ($\cong_{\forall C}$) if they give the same observable behavior in all state contexts. That is,

Definition 1 (contextual equivalence, $\cong_{\forall C}$)
If $\Gamma \vdash M, N{:}\tau$, then $\Gamma \vdash M \cong_{\forall C} N{:}\tau$ iff for all closing contexts $C[-_\tau]$ of type pr that respect Γ, and for all stores Σ,

$$\langle\!\langle C[M]\ ;\ \Sigma\rangle\!\rangle \approx_{bisim} \langle\!\langle C[N]\ ;\ \Sigma\rangle\!\rangle$$

A context $C[-]$ respects a type assignment Γ if $C[-]$ binds the free variables of any term inserted in the context hole with binders of the correct type.

For terms M and N containing no locations, Definition 1 is equivalent to one in which Σ is fixed to be empty; this includes all of our applications. The meaning of a metalanguage term M is defined to be the $\cong_{\forall C}$-equivalence class of M. Contextual equivalence is, by the form of its definition, a *congruence* – that is, it is preserved by all term constructors of the language. Furthermore, it is easy to show (using the confluence property for PCF), that if $M =_{pcf} N$, then $M \cong_{\forall C} N$. Hence the structure obtained by taking equivalence classes of open terms under $\cong_{\forall C}$ is a model of typed lambda-calculus. This is our operationally-based term model. Because this is a model, ordinary reasoning using β-conversion is valid in the model.

The quantification over all contexts makes $\cong_{\forall C}$ difficult to establish directly. We must consider not only contexts in which the terms are executed, but also contexts that pass the terms around, put many copies of them in the store, etc.

Our major tool for easing this proof burden is a "context lemma" which states that two terms are contextually equivalent if and only if they are equivalent in a smaller set of contexts we call *extensional contexts*. At function types, extensional contexts are head contexts, as in Milner's context lemma for PCF [Mil77] – that is, extensionally equivalent functions take the same argument to extensionally equivalent results. Extensional contexts for the other type constructors are natural: for product types, both projections must be extensionally equivalent, and for sum types, summands exposed by the `case` operator must be equivalent. We omit the formal definition of extensional equivalence, denoted $\cong_{ext}$, in this abstract. We then have, from [WS97]:

Theorem 2 (Context Lemma)

$$\Gamma \vdash M \cong_{ext} N{:}\tau \Leftrightarrow \Gamma \vdash M \cong_{\forall C} N{:}\tau$$

6 Translation from Scheme$_{cell}$ to the Metalanguage

Figure 4 defines the semantics of Scheme$_{cell}$ terms via a syntax-directed CPS translation, $[\![-]\!]_{cell}$, into the metalanguage. The translation is a standard CPS translation, as in [Plo75, Cli84, App92], and closely follows the official denotational semantics of Scheme [CR91].

The type *Val* of expressed values is a recursive sum:

$Val = N + F$	expressed values
$N = int$	integers
$F = Val^* \to (Val \to pr) \to pr$	procedures

$$[\![n]\!]_{cell} = \lambda\kappa{:}\, Val{\rightarrow}pr.\, in_N\; n\; \kappa$$

$$[\![x]\!]_{cell} = \lambda\kappa{:}\, Val{\rightarrow}pr.\kappa\; x$$

$$\begin{array}{l} [\![(\mathsf{lambda}\ (x_1 \ldots x_n)\ e)]\!]_{cell} = \lambda\kappa{:}\, Val{\rightarrow}pr. \\ \qquad in_F\,(\lambda v^*{:}\, Val^*.\lambda\kappa'{:}\, Val{\rightarrow}pr. \\ \qquad\quad [\![e]\!]_{cell}[(listref_1\; v^*)/x_1, \ldots, (listref_n\; v^*)/x_n]\; \kappa')\; \kappa \end{array}$$

$$\begin{array}{l} [\![(e_0\; e_1 \ldots e_n)]\!]_{cell} = \lambda\kappa{:}\, Val{\rightarrow}pr.[\![e_0]\!]_{cell}(\lambda f{:}\, Val. \\ \qquad [\![e_1]\!]_{cell}(\lambda v_1{:}\, Val. \cdots \\ \qquad\quad [\![e_n]\!]_{cell}(\lambda v_n{:}\, Val. \\ \qquad\qquad out_F\; f\; (\lambda f'{:}\, Val^*{\rightarrow}(Val{\rightarrow}pr){\rightarrow}pr. \\ \qquad\qquad\quad f'\; \langle v1, \ldots, v_n\rangle\; \kappa)) \ldots)) \end{array}$$

$$[\![(\mathsf{make\text{-}cell}\; e)]\!]_{cell} = \lambda\kappa{:}ref(Val){\rightarrow}pr.[\![e]\!]_{cell}(\lambda v{:}\, Val.\, \mathsf{new}\; v\; \kappa)$$

$$[\![(\mathsf{cell\text{-}ref}\; x)]\!]_{cell} = \lambda\kappa{:}\, Val{\rightarrow}pr.\, \mathsf{deref}\; x\; \kappa$$

$$[\![(\mathsf{cell\text{-}set!}\ x\ e)]\!]_{cell} = \lambda\kappa{:}\, Val{\rightarrow}pr.[\![e]\!]_{cell}(\lambda v{:}\, Val.\, \mathsf{update}\; x\; v\; (\kappa\; \Omega)))$$

$$\begin{array}{l} [\![(\mathsf{letrec}\ ((x_1\; e_1) \ldots (x_n\; e_n))\ e)]\!]_{cell} = \lambda\kappa{:}\, Val{\rightarrow}pr. \\ \qquad \mathsf{new}\,\Omega_{Val}(\lambda x_1{:}ref(Val). \cdots \mathsf{new}\,\Omega_{Val}(\lambda x_n{:}ref(Val). \\ \qquad\quad [\![e_1]\!]_{cell}(\lambda v_1{:}\, Val.\, \mathsf{update}\; x_1 v_1 \cdots \\ \qquad\qquad [\![e_n]\!]_{cell}(\lambda v_n{:}\, Val.\, \mathsf{update}\; x_n v_n\; ([\![e]\!]_{cell}\kappa) \cdots) \end{array}$$

Figure 4: CPS translation from Scheme$_{cell}$ to metalanguage

For each summand S of the Val type, we use CPS injection and extraction combinators, $in_S : S{\rightarrow}(Val{\rightarrow}pr){\rightarrow}pr$ and $out_S : Val{\rightarrow}(S{\rightarrow}pr){\rightarrow}pr$. We also use $\langle -, \ldots, - \rangle$ and $listref_i$ for the evident list operations. All these can be easily defined in the metalanguage.

Identifiers denote either values of type Val or cells of type $ref(Val)$.

A Scheme$_{cell}$ lambda-expression returns a closure which expects a list of values v^* and a continuation κ.

The rule for letrec follows the official semantics of Scheme [CR91], which initializes the variables to undefined values while building up the scope (var/loc pairs) and then destructively updates the variables to their correct values.

It is easy to show that if we apply the translation $[\![-]\!]_{cell}$ to well-typed Scheme$_{cell}$ terms, the resultant metalanguage terms will be well-typed:

Lemma 3 *For a Scheme$_{cell}$ term e, and a type assumption Γ mapping variables to $\{Val, ref(Val)\}$,*

$$\Gamma \vdash_{cell} e{:}ok \Rightarrow \Gamma \vdash [\![e]\!]_{cell}{:}(Val{\rightarrow}pr){\rightarrow}pr$$

□

7 Correctness of Incremental Lambda Lifting

Our goal is to show that the incremental lambda lifting transformation preserves the meaning of Scheme$_{cell}$ terms; that is:

Theorem 4 (Correctness of Incremental Lambda Lifting) *For any Scheme$_{cell}$ expression e and list of lists of variables $\mathbb{X}$ such that e is lambda-liftable with respect to $\mathbb{X}$, and any Γ mapping the free variables of e to $ref(Val)$, if $\Gamma \vdash_{cell} e{:}ok$, then*

$$\Gamma \vdash [\![e]\!]_{cell} \cong_{\forall C} [\![LL(\mathbb{X}, e)]\!]_{cell}{:}(Val \to pr) \to pr$$

PROOF:

1. First we check that the lambda lifting transformation preserves types; that is,

$$\Gamma \vdash_{cell} e{:}ok \Rightarrow \Gamma \vdash [\![LL(\mathbb{X}, e)]\!]_{cell}{:}(Val \to pr) \to pr$$

   The proof is by structural induction on $\vdash_{cell}$. We need criterion (4b) from the list of criteria for lambda-liftability in order to be assured that the identifiers in X_k added to calls of the lifted procedure f_k will be in scope at the modified call sites.

2. By appeal to the Context Lemma (Theorem 2) and the definition of $\cong_{ext}$, it suffices to show that for all closing substitutions σ mapping free variables to closed terms of type $ref(Val)$, all closed terms K of type $Val \to pr$, and all stores Σ,

$$\langle\!\langle ([\![e]\!]_{cell}\sigma)\ K\ ;\ \Sigma \rangle\!\rangle \approx_{bisim} \langle\!\langle ([\![LL(\mathbb{X}, e)]\!]_{cell}\sigma)\ K\ ;\ \Sigma \rangle\!\rangle$$

3. To show these states are bisimilar, we construct a relation S_{LL} between metalanguage states that characterizes the syntactic differences between (non-transformed) Scheme$_{cell}$ programs and those programs after incremental lambda lifting, as they reduce under $\overset{\alpha}{\to}_{io}$. As the original program and its lambda-lifted version run, there are several different ways in which corresponding subterms can differ:

   (a) The initial translations, before any evaluation takes place, will differ as follows.

   On the left-hand (non-lambda-lifted) side, the continuation term K will be immediately invoked with a closure for the Scheme$_{cell}$ term

$$(\mathsf{lambda}\ (X_0)\ (\mathsf{letrec}\ (\ldots (f_k\ (\mathsf{lambda}\ (Y_k)\ e_k)) \ldots))\ e_0)$$

   The closure sent to K expects values for each identifier in X_0, and once invoked with such arguments, will do allocation for each of the local procedure closures. We represent the closure sent to K using a macro[1] $\mathcal{L}$, which is parameterized by the identifiers used during lambda lifting and by the bodies of the local procedures. The term component of the state will look like:

$$(K\ \mathcal{L}(X_0, \mathbb{F},\ [\![e_0]\!]_{cell}, \ldots, [\![e_n]\!]_{cell}))$$

   and the store is unchanged.

[1]The final paper will contain the explicit definitions of $\mathcal{L}$ and other macros.

On the right-hand (lambda-lifted) side, much more happens before a closure is sent to K. The closures for the local procedures $f_1, \ldots, f_n$ are allocated and initialized, and then a closure for the translation of

$$(\mathsf{lambda}\ (X_0)\ LL_{body}(\langle f_1, \ldots, f_n\rangle, \mathbb{X}, e_0))$$

is sent to K. We represent the closure sent to K on the right-hand side using a macro $\mathcal{R}(X_0, P)$ which takes only the X_0 identifier list and the body of the $\mathsf{lambda}\ (X_0)$ abstraction. So we will have a state with a term component of:

$$(K\ \mathcal{R}(X_0,\ [\![LL_{body}(F, \mathbb{X}, e_0)]\!]_{cell}))[l_i/f_i]_{i=1}^{n}$$

The store will have newly allocated locations $l_1, \ldots, l_n$ initialized to closures, for which we use a macro $\mathcal{C}'(X_k, Y_k, P)$. For $1 \leq k \leq n$ we have:

$$\Sigma(l_k) = \mathcal{C}'(X_k, Y_k,\ [\![LL_{body}(F, \mathbb{X}, e_k)]\!]_{cell})[l_i/f_i]_{i=1}^{n}$$

(b) We next consider what happens when the closure sent to K above is actually used. This corresponds to sending the closure a list of arguments, say $\langle M_1, \ldots, M_{m_0}\rangle$, which are bound to the identifiers $x_1, \ldots, x_{m_0}$.

The left-hand side allocates n fresh locations, storing new instances of the closures for its local procedures at these locations. Each of these newly-created closures inherits the bindings of values to the $x_1, \ldots, x_{m_0}$ identifiers. After creating the new local procedure closures, the body, e_0, is executed, at which point we will have a state with a term component something like:

$$[\![e_0]\!]_{cell}[l_k/f_k]_{k=1}^{n}[M_j/x_j]_{j=1}^{m_0}$$

and with a store that looks like:

$$\Sigma[l_k \mapsto (\mathcal{C}(Y_k, M_{f_k})[l_i/f_i]_{i=1}^{n}[M_j/x_j]_{j=1}^{m_0})]_{k=1}^{n}$$

where the $\mathcal{C}$ macro represents a closure for the translation of a local procedure.

On the right-hand, lambda-lifted, side, much less happens when the closure for $\mathsf{lambda}(X_0)$ is applied. Evaluation of the lambda-lifted body, $LL_{body}(\langle f_1, \ldots, f_n\rangle, \mathbb{X}, e_0)$ proceeds immediately with the substitution $[M_j/x_j]_{j=1}^{m_0}$, as the closures were initialized in the previous step, so we will have a state with a term component such as

$$[\![LL_{body}(\langle f_1, \ldots, f_n\rangle, \mathbb{X}, e_0)]\!]_{cell}[l_i/f_i]_{i=1}^{n}[M_j/x_j]_{j=1}^{m_0}$$

and the store component will not have changed significantly.

(c) Finally, let us consider what happens at runtime when a local procedure f_k is applied. On the left-hand side, we represent an application using the $\mathcal{A}$ macro, which takes the procedure identifier, each argument, and a continuation. So we will have something like:

$$\mathcal{A}(f_k, [\![e_1]\!]_{cell}, \ldots, [\![e_{m_k}]\!]_{cell}, M_{cont})$$

On the left-hand side, the application will also take the arguments added by the lambda lifting transformation, and we use the $\mathcal{A}'$ macro, which is the same as $\mathcal{A}$ but with an added parameter, so we have:

$$\mathcal{A}'(f_k, X_k, [\![e'_1]\!]_{cell}, \ldots, [\![e'_{m_k}]\!]_{cell}, N_{cont})$$

Given $\mathbb{X} = \langle X_1, \ldots, X_n \rangle$ and $\mathbb{F} = \langle (f_1, Y_1), \ldots, (f_n, Y_n) \rangle$, $\overset{LL_{body}}{\leadsto}$ is the least relation closed under the following rules:

$$\Gamma[x{:}\tau], \mathbb{X}, \mathbb{F} \vdash x \overset{LL_{body}}{\leadsto} x{:}\tau, \quad x \notin dom(\mathbb{F}) \quad (\textsc{cong-var})$$

$$\Gamma, \mathbb{X}, \mathbb{F} \vdash l^\tau \overset{LL_{body}}{\leadsto} l^\tau{:}\mathit{ref}(\tau) \quad (\textsc{cong-loc})$$

$$\frac{\Gamma, \mathbb{X}, \mathbb{F} \vdash M \overset{LL_{body}}{\leadsto} N{:}\tau' \to \tau \quad \Gamma \vdash M' \overset{LL_{body}}{\leadsto} N'{:}\tau'}{\Gamma, \mathbb{X}, \mathbb{F} \vdash (M\ M') \overset{LL_{body}}{\leadsto} (N\ N'){:}\tau} \quad (\textsc{cong-app})$$

$$\frac{\begin{array}{c}\Gamma, \mathbb{X}, \mathbb{F} \vdash M_i \overset{LL_{body}}{\leadsto} N_i{:}(\mathit{Val} \to \mathit{pr}) \to \mathit{pr}, \quad 1 \le i \le m_k \\ \Gamma, \mathbb{X}, \mathbb{F} \vdash M_{cont} \overset{LL_{body}}{\leadsto} N_{cont}{:}\mathit{Val} \to \mathit{pr} \\ M = \mathcal{A}(f_k, M_1, \ldots, M_{m_k}, M_{cont}) \\ N = \mathcal{A}'(f_k, X_k, N_1, \ldots, N_{m_k}, N_{cont})\end{array}}{\Gamma, \mathbb{X}, \mathbb{F} \vdash M \overset{LL_{body}}{\leadsto} N{:}\mathit{pr}} \quad (\textsc{app-}f_k)$$

Figure 5: Definition of $\overset{LL_{body}}{\leadsto}$ on Terms

The left-hand side will retrieve its most recently created f_k closure and send it a list of m_k parameters. When evaluation reaches the translated body of the procedure, e_k, references to $x_j \in X_0$ are resolved by the substitution in effect when the closure for f_k was created.

The right-hand side will retrieve the closure for f_k created in Step 3a. Because the closure was stored before the current substitution for $x_j \in X_0$ was in effect, the values for each such x_j required by f_k must be sent at application time, which happens as a result of the lambda-lifting transformation.

4. Once the closures for lambda (X_0) have been evaluated, the differences between the left and right-hand sides correspond to calls to the lifted local procedures. The relation $\overset{LL_{body}}{\leadsto}$ between metalanguage terms, defined in Figure 5, encompasses the differences found within translations of procedure bodies. The rule $\textsc{app-}f_k$ is the most interesting – it says that two applications (covered by the $\mathcal{A}$ and $\mathcal{A}'$ macros) are $\overset{LL_{body}}{\leadsto}$-related if the right-hand side adds the correct arguments, X_k, at the call site.

5. The candidate bisimulation relation S_{LL} is defined in Figure 6. The "interesting" differences between the translations of Scheme$_{cell}$ terms and their lambda-lifted counterparts are abstracted out using fresh variables u_i and v_i as placeholders, for i ranging from 1 to the total number of different corresponding subterms.

   The u_i variables map to the **u**nevaluated closures for lambda (X_0) discussed in Step 3a above. On the right-hand side, we have n closures already initialized in the store, at locations $l_1, \ldots, l_n$.

   The v_i variables map to the e**v**aluated closures discussed in Step 3b above. The left-hand side has n freshly allocated locations bound to its local procedure identifiers $f_1, \ldots, f_n$. Both sides have an active substitution for the $x_j \in X_0$ arguments. Note that the closures newly stored in Σ (the left-hand store) have substitutions for the $x_j \in X_0$ as part of the closure.

Definition 5 (S_{LL})
Given $\mathbb{X} = \langle x_1, \ldots, x_n \rangle$ *and* $\mathbb{F} = \langle (f_1, Y_1), \ldots, (f_n, Y_n) \rangle$,

$$S_{LL} \stackrel{def}{=} \{ (\langle\!\langle M \;;\; \Sigma \rangle\!\rangle \sigma, \langle\!\langle M \;;\; \Sigma' \rangle\!\rangle \sigma') \mid$$

$$\text{preamble} \left\{ \begin{array}{l} \exists u, v \in \mathit{Var}^* \text{ s.t.} \\ \mathit{fv}(M, \mathit{rng}(\Sigma), \mathit{rng}(\Sigma')) \subseteq u \cup v \\ \sigma = \sigma_u \cup \sigma_v \\ \sigma' = \sigma'_u \cup \sigma'_v \\ \exists \Sigma_0 \text{ s.t. } \mathit{dom}(\Sigma_0) \subseteq \mathit{dom}(\Sigma), \text{ and } \mathit{dom}(\Sigma_0) \subseteq \mathit{dom}(\Sigma') \\ (\Gamma \vdash M{:}\mathit{pr}) \Rightarrow (\Gamma(u_i) = \mathit{Val},\ \Gamma(v_i) = \mathit{pr}) \end{array} \right.$$

$$\begin{array}{c}\text{unevaluated} \\ \mathsf{lambda}\ (X) \\ \text{closures}\end{array} \left\{ \begin{array}{l} \sigma_u = [\, u_i \mapsto (\mathcal{L}(X_0, \mathbb{F}, M_0^i, \ldots, M_n^i)) \,]_{i=1}^{\#u} \\ \sigma'_u = [u_i \mapsto (\mathcal{R}(X_0, N_0^i)[l_k/f_k]_{k=1}^{n}) \,]_{i=1}^{\#u} \\ \Sigma' = \Sigma_0[l_k \mapsto (\mathcal{C}'(X_k, Y_k, N_k)[l_i/f_i]_{i=1}^{n}) \,]_{k=1}^{n} \\ \left. \begin{array}{l} \Gamma, \mathbb{X}, \mathbb{F} \vdash M_0^i \stackrel{LL_{body}}{\leadsto} N_0^i{:}(\mathit{Val} \to \mathit{pr}) \to \mathit{pr} \\ \Gamma, \mathbb{X}, \mathbb{F} \vdash M_k^i \stackrel{LL_{body}}{\leadsto} N_k{:}(\mathit{Val} \to \mathit{pr}) \to \mathit{pr} \end{array} \right\} \begin{array}{c} 1 \leq i \leq \#u \\ 1 \leq k \leq n \end{array} \end{array} \right.$$

$$\begin{array}{c}\text{evaluated} \\ \mathsf{lambda}\ (X) \\ \text{closures}\end{array} \left\{ \begin{array}{l} \sigma_v = [v_i \mapsto M_i[l_k^i/f_k]_{k=1}^{n}[M_{x_j}^i/x_j]_{j=1}^{m_0} \,]_{i=1}^{\#v} \\ \sigma'_v = [v_i \mapsto N_i[N_{x_j}^i/x_j]_{j=1}^{m_0}\ [l_i/f_i]_{i=1}^{n} \,]_{i=1}^{\#v} \\ \Sigma = \Sigma_0[l_k^i \mapsto (\mathcal{C}(Y_k, M_k^i)[l_k^i/f_k]_{k=1}^{n}[M_{x_j}^i/x_j]_{j=1}^{m_0}) \,]_{i=1,k=1}^{\#v,\, n} \\ \Gamma, \mathbb{X}, \mathbb{F} \vdash M_i \stackrel{LL_{body}}{\leadsto} N_i{:}\mathit{pr}, \text{ for } 1 \leq i \leq \#v \\ \Gamma, \mathbb{X}, \mathbb{F} \vdash M_{x_j}^i \stackrel{LL_{body}}{\leadsto} N_{x_j}^i{:}\mathit{Val}, \text{ for } 1 \leq i \leq \#v, 1 \leq j \leq m_0 \end{array} \right.$$

$$\}$$

Figure 6: The S_{LL} Candidate Simulation

All procedure bodies are related by $\stackrel{LL_{body}}{\leadsto}$.

6. With an eye towards the coinductive definition of bisimulation, we show that:

$$S_{LL} \subseteq [\approx_{bisim} \circ S_{LL} \circ \approx_{bisim}]_{bisim}$$

This requires an induction on the number of steps to an input-output operation, which is done using a case analysis on the form of M; the interesting cases are when M is headed by a u_i or v_i variable.

7. $S_{LL} \subseteq \approx_{bisim}$ follows by a straightforward application of "bisimulation up to" reasoning, as in [Mil90].

8. Finally, we show that the S_{LL} relation encompasses the translations of all pre- and post-lambda-lifted programs, but this is easy, as the S_{LL} was defined with exactly such programs in mind.

□

8 Conclusion

Given a source-to-source transformation which we wish to show preserves meaning, we do the following:

1. Define a translation from the source language to the metalanguage. The translation must be sound with respect to the semantics of the source language.

2. Define a "candidate bisimulation relation" which characterizes all of the ways in which corresponding metalanguage subterms might differ as the translations of pre- and post-transformed terms execute in a state context.

3. Prove that the relation defined above is contained in $\approx_{bisim}$. These proofs involve reducing each possible form of the translated terms in extensional contexts of the appropriate type and verifying that as the states reduce they stay within the candidate relation. While translation from the source language to our metalanguage is denotational in style, the proofs of bisimulation tend to involve lots of operational reasoning – thus the moniker "almost denotational" for our approach.

The relations constructed as a basis for a proof of correctness are distillations of the essence of the effects of the transformation being considered. As such, the proof method as a whole serves to clarify and debug the formalization of any complex transformation. We have found that our methods are able to handle relatively complex source-to-source transformations such as incremental lambda lifting and language-to-language transformations such as assignment elimination, even in the presence of a global store and imperative features.

We plan on extending this research to handle more complicated type systems, such as those found in object-oriented languages. We also plan to investigate using our method to justify transformations which rely on flow analysis. Also, the process of inventing candidate bisimulation relations and verifying their correctness is both tricky and time consuming, and we would like to pursue possible tools to facilitate the process.

References

[Abr90] Samson Abramsky. The lazy lambda calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, Cambridge, 1992.

[CH94] William D. Clinger and Lars Thomas Hansen. Lambda, the ultimate label; or a simple optimizing compiler for scheme. In *Proc. 1994 ACM Symposium on Lisp and Functional Programming*, pages 128–139, 1994.

[Cli84] William Clinger. The scheme 311 compiler: An exercise in denotational semantics. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364, August 1984.

[CR91] William Clinger and Jonathan A. Rees. The revised^4 report on the algorithmic language scheme. *ACM LISP Pointers*, 4(3), 1991.

[Fel87] Matthias Felleisen. *The Calculi of Lambda-v-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Indiana University, 1987.

[Gor94] Andrew D. Gordon. *Functional Programming and Input/Output.* Cambridge University Press, Cambridge, 1994.

[Gor95] Andrew D. Gordon. Bisimilarity as a theory of functional programming. In *Proceedings of 11th Conference on Mathematical Foundations of Programming Semantics*, 1995.

[Hug82] R. J. M. Hughes. Super-combinators: A new implementation method for applicative languages. In *Proc. 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, 1982.

[KKR$^+$86] David A. Kranz, Richard Kelsey, Jonathan A. Rees, Paul Hudak, James Philbin, and Norman I. Adams. Orbit: An optimizing compiler for scheme. In *Proceedings SIGPLAN '86 Symposium on Compiler Construction*, 1986. *SIGPLAN Notices 21*(7), July, 1986, 219-223.

[Mil77] Robin Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[Mil90] Robin Milner. Operational and algebraic semantics of concurrent processes. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 1201–1242. MIT Press/Elsevier, 1990.

[MMH96] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Conf. Rec. 23rd ACM Symposium on Principles of Programming Languages*, pages 271–283, 1996.

[Mor68] James H. Morris, Jr. *Lambda Calculus Models of Programming Languages.* PhD thesis, MIT, Cambridge, MA, 1968.

[MT91] Ian A. Mason and Carolyn L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

[ORW95] Dino P. Oliva, John D. Ramsdell, and Mitchell Wand. The VLISP verified prescheme compiler. *Lisp and Symbolic Computation*, 8(1/2):111–182, 1995.

[Par81] David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science: 5th GI-Conference, Karlsruhe*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, Berlin, Heidelberg, and New York, March 1981. Springer-Verlag.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International, 1987.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

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

[Sta94] Ian Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, December 1994. Also published as Technical Report 363, University of Cambridge Computer Laboratory.

[Sul96] Gregory T. Sullivan. *Operationally-Based Models of Higher-Order Imperative Programming Languages*. PhD thesis, Northeastern University, December 1996.

[TW96] Jerzy Tiuryn and Mitchell Wand. Untyped lambda-calculus with input-output. In H. Kirchner, editor, *Trees in Algebra and Programming: CAAP'96, Proc. 21st International Colloquium*, volume 1059 of *Lecture Notes in Computer Science*, pages 317–329, Berlin, Heidelberg, and New York, April 1996. Springer-Verlag.

[WO92] Mitchell Wand and Dino P. Oliva. Proving the correctness of storage representations. In *Proc. 1992 ACM Symposium on Lisp and Functional Programming*, pages 151–160, 1992.

[WS94] Mitchell Wand and Paul Steckler. Selective, lightweight closure conversion. In *Conf. Rec. 21st ACM Symposium on Principles of Programming Languages*, pages 435–445, 1994. Revised version submitted for publication.

[WS97] Mitchell Wand and Gregory T. Sullivan. Denotational semantics using an operationally-based term model. In *Conf. Rec. 24th ACM Symposium on Principles of Programming Languages*, page ??, 1997.