Yale University
Department of Computer Science

Call by Name, Assignment, and the Lambda Calculus[1]

Martin Odersky Dan Rabin Paul Hudak

Research Report YALEU/DCS/RR-929
October, 1992

This work was supported in part by DARPA grant number N00014-91-J-4043. The second author was supported during the final preparation of this paper by an IBM Graduate Fellowship.

[1]This article will appear in the proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 11–13, 1993.

Abstract

We define an extension of the call-by-name lambda calculus with additional constructs and reduction rules that represent mutable variables and assignments. The extended calculus has neither a concept of an explicit store nor a concept of evaluation order; nevertheless, we show that programs in the calculus can be implemented using a single-threaded store. We also show that the new calculus has the Church-Rosser property and that it is a conservative extension of classical lambda calculus with respect to operational equivalence; that is, all algebraic laws of the functional subset are preserved.

1 Introduction

Are assignments harmful? Common wisdom in the functional programming community has it that they are: seemingly, they destroy referential transparency, they require a determinate evaluation order, and they weaken otherwise powerful type systems such as ML's. Consequently, programming languages with a strong functional orientation often forbid or at least discourage the use of assignments.

On the other hand, assignments are useful. With them, one can implement mutable, implicit, distributed state—a powerful abstraction, even if it is easily misused. The traditional alternative offered by functional programming is to make state explicit. The resulting "plumbing" problems can be ameliorated by hiding the state parameter using monads [20] or by using continuation-passing style [10]. Wadler, for example, uses the monad technique in [22] to present "pure" functional programming as an alternative to "impure" programming with assignments. Monads are indeed successful in eliminating explicit mention of state arguments, but they still require a centralized definition of state.

We show here that one need not choose between purity and convenience. We develop a framework that combines the worlds of functions and state in a way that can naturally express advanced imperative constructs without destroying the algebraic properties of the functional subset. The combinations are referentially transparent: names can be freely exchanged with their definitions. More generally, we show that every meaningful operational equivalence of the functional subset carries over to the augmented language.

Since we would like to abstract away from the issues of a particular programming language, we will concentrate in this paper on a calculus for reasoning about functions and assignments. The calculus is notable in that it has neither a concept of an explicit store nor a concept of evaluation order. Instead, expanding on an idea of Boehm [2], we represent "state" by the collection of assignment statements in a term. A Church-Rosser property guarantees that every reduction sequence to normal form yields the same result. Following Plotkin [15] and Felleisen [3], we derive from the reduction rules both a theory and an evaluator and study the relationship between them.

The main contributions of this paper are:

- We define (in Section 2) syntax and reduction rules of λ_{var}, a calculus for functions and state.
- We show (in Section 3) that λ_{var} is Church-Rosser and that it admits a deterministic evaluation function which acts as a semi-decision procedure for equations between terms and answers.
- Even though the syntax of λ_{var} is storeless, we show (in Section 4) that λ_{var}-programs can still be efficiently implemented using a single-threaded store.
- We show (in Section 5) a strong conservative extension theorem: every operational equivalence between terms in classical applied λ-calculus also holds in λ_{var} (provided the domain of basic constants and constructors is sufficiently rich). This is to our knowledge the first time such a result has been established for an imperative extension of the λ-calculus.

These properties make λ_{var} suitable as a basis for the design of wide-spectrum languages which combine functional and imperative elements. On the functional side, we generally assume call-by-name, but call-by-value can also be expressed, since strictness can be defined by a δ-rule. On the imperative side, first class variables and procedures can be used as building blocks for mutable objects (Section 6 presents an example making use of these constructs). We do not impose any particular restrictions on either functions or side-effecting procedures, except for requiring that their difference is made explicit.

Building on λ_{var} is attractive because it gives us an equational semantics that makes reasoning about programs quite straightforward. In contrast, the traditional store-based denotational or operational semantics of imperative languages impose a much heavier burden on program derivations and proofs: at every step, one has to consider the global layout of the store, including a map from names to locations and a map from locations to values. Other semantic approaches, such as Hoare

x	$\in$	*Vars*	immutable variables
v	$\in$	*Tags*	mutable variables (tags)
f	$\in$	*FConsts*	primitive functions
c^n	$\in$	*Constrs*	constructors of arity n $(n \geq 0)$
M	$\in$	Λ_{var}	terms

$$\begin{array}{rcl} M & ::= & f \mid c^n \mid x \mid x.M \mid M_1\ M_2 \\ & \mid & v \mid \mathbf{var}\ v.M \mid M? \mid M_1 =: M_2 \mid M_1 \triangleright x.M_2 \\ & \mid & \mathbf{return}\ M \mid \mathbf{pure}\ M \end{array}$$

Figure 1: Syntax of λ_{var}

logic or weakest predicate transformers might accommodate simpler reasoning methods, but they are not easily extended to structure sharing or higher-order functions.

2 Term Syntax and Reduction Rules of λ_{var}

The term-forming productions of λ_{var} fall into three groups, each presented on one line in Figure 1. The first group consists of clauses defining λ-calculus with primitive function symbols and data constructors. We refer to this basic calculus as the *applied* λ-calculus. The second group adds the constructs for modeling assignment; the third introduces constructs for mediating between the world of assignments and the world of functions.

Basic applied λ-terms. We denote functional abstraction $(x.M)$ without the customary leading λ; this modification makes some of our reduction rules more legible. The presence of primitive function symbols f and fixed-arity constructors c^n shows the applied nature of the calculus. Basic constants are included as constructors of arity 0. We assume that every calculus we consider has at least the unit value () as basic constant.

Store tags and primitive state transformers. The scope of a mutable variable v is delimited by the construct **var** $v.M$. Mutable variables, also called *tags*, are syntactically distinct from the immutable variables introduced by abstractions $x.M$. We denote tags by the letters u, v, w, and immutable variables by x, y, z.

Tag readers $M?$ and assignments $M_1 =: M_2$ are the primitive state transformers. If M computes a tag, $M?$ is the state transformer that produces the value associated with that tag without altering the store. Dually, if M_2 computes a tag, $M_1 =: M_2$ is the state transformer that sets that tag to M_1 and produces an ignorable value.

Composition of state transformers. State transformers are composed into sequences using the monad-bind expression $M_1 \triangleright x.M_2$. This construct connects a state transformer M_1 with a functional abstraction $x.M_2$. It denotes the state transformer that passes the value produced by M_1 to $x.M_2$ in the state resulting from the computation of M_1. We take $(\triangleright)$ to be right-associative and often employ the following abbreviation:

$$N\ ;\ M \quad \stackrel{\text{def}}{=} \quad N \triangleright x.M \qquad (x \notin fv\ M).$$

Coercion of state transformers. The λ_{var}-expression **return** M allows a pure expression M to be used as a state transformer; the expression **pure** M permits (under certain conditions) the coercion of a state transformer to a pure expression.

Correspondence with programming languages. Figure 2 relates terms of Λ_{var} with constructs of traditional imperative programming languages. We use Modula as a representative of such a language.

The λ_{var}-calculus deviates from common imperative programming languages in its notation for assignments, which goes from left to right, and in its variable-readers, which are explicit state transformers rather than expressions. These notational conventions make tag-matching in the reduction rules easier to follow. In particular, because of the re-orientation of assignments, information

λ_{var}	Modula	
$v? \triangleright x.M$	$[v/x]M$	variable lookup (implicit in Modula)
$N \triangleright x.M$	$N(x)\ ;\ M$	procedure call, x is result parameter
$\mathbf{var}\ v.M$	$VAR\ v : T\ ;\ M$	variable definition
$M =: v$	$v := M$	assignment
$N\ ;\ M$	$N\ ;\ M$	sequential composition
$\mathbf{return}\ M$	$RETURN\ M$	return statement
$\mathbf{pure}\ M$	M	effect masking, implicit in Modula

Figure 2: Correspondence between λ_{var} and Modula

and computation in a state transformer flows uniformly from left to right. In each case, the conventional notation can be obtained by syntactic sugaring, if desired. We would expect that such sugaring is introduced for any programming languages based on λ_{var}.

Notational conventions for reduction. We use *bv* M (*fv* M) to denote the bound (free) variables and tags in a term M. A term is *closed* if *fv* $M = \emptyset$. Closed terms are also called *programs*. We use $M \equiv N$ for syntactic equality of terms (modulo α-renaming) and reserve $M = N$ for convertibility. If R is a notion of reduction, we use $M \underset{R}{\rightarrow} N$ to express that M reduces in one R reduction step to N, and $M \underset{R}{\twoheadrightarrow} N$ to express that M reduces in zero or more R-steps to N. The subscript is dropped if the notion of reduction is clear from the context. A *value* V is a λ-abstraction, a primitive function, or a (possibly applied) constructor. An *observable value* (or *answer*) A is an element of some nonempty subset of the basic constants[1].

$$\begin{array}{lcl} V & ::= & x.M \mid f \mid c^n\ M_1 \ldots M_m \quad (0 \leq m \leq n) \\ A & \subseteq & c^0 \end{array}$$

A *context* C is a term with a hole $[\,]$ in it. A *state prefix* S is a special context that is of one of the forms

$$S \quad ::= \quad [\,] \quad | \quad \mathbf{var}\ v.S \quad | \quad M =: v\ ;\ S$$

and that satisfies in addition the requirement that $wr\ S \subseteq bv\ S$. The set of variables *written* in S, $wr\ S$, is defined as follows:

$$\begin{array}{lcl} wr\ [\,] & = & \emptyset \\ wr\ (\mathbf{var}\ v.S) & = & wr\ S \\ wr\ (M =: v\ ;\ S) & = & \{v\} \cup wr\ S. \end{array}$$

[1] Other observations such as convergence to an arbitrary value can be encoded using suitable δ-rules.

Following Barendregt [1], we take terms that differ only in the names of bound variables to be equal. Hence all terms we write are representatives of equivalence classes of α-convertible terms. We follow the "hygiene" rule that bound and free variables in a representative are distinct, and we use the same conventions for tags.

Figure 3 gives the reduction rules of λ_{var}.

Rule (β) is the usual β-rule of applied λ-calculus. It is the only rule whose reduction involves substitution.

Rule (δ) expresses rewriting of applied basic functions. To abstract from particular constants and their rewrite rules, we only require the existence of a partial function δ from primitive functions[2] f and values to terms. We restrict δ not to "look inside" the structure of its argument term, except when the term is a fully applied constructor at top-level. That is, we postulate that for every primitive function f there exist terms N_f and N_{f,c^n} ($c^n \in Constrs$) such that for all values V for which $\delta(f, V)$ is defined:

$$\delta(f,V) \quad = \quad \begin{cases} N_{c^n}\ M_1 \ldots M_n & \text{if } V \equiv c^n\ M_1 \ldots M_n \\ N\ V & \text{otherwise.} \end{cases}$$

State transformers obey two of the three laws of a Kleisli monad: ($\triangleright$) is associative and **return** is a left unit. The third law, stating that **return** is a right unit, fails. A counter-example is

$$1 \triangleright (x.\mathbf{return}\ x) \quad \neq \quad 1.$$

Note, however, that this example would be typically regarded as a type error in a statically typed language, since the number 1 is not a state transformer. In fact, every reasonable type system should establish the third monad law as an operational equivalence for well-typed terms.

[2] Primitive functions of more than one argument are obtained by currying.

$$
\begin{array}{llcll}
\beta & (x.M)\ N & \to & [N/x]\ M & \\
\delta & f\ V & \to & \delta(f,V) & (\delta(f,V) \text{ defined}) \\
\\
\triangleright\triangleright & (M_1 \triangleright x.M_2) \triangleright y.M_3 & \to & M_1 \triangleright x.(M_2 \triangleright y.M_3) & \\
r\triangleright & (\mathbf{return}\ N) \triangleright x.M & \to & (x.M)\ N & \\
v\triangleright & (\mathbf{var}\ v.M) \triangleright x.N & \to & \mathbf{var}\ v.(M \triangleright x.N) & \\
=:\triangleright & (M_1 =: M_2) \triangleright x.M_3 & \to & M_1 =: M_2\ ;\ (x.M_3)\ () & (x \in fv\ M_3) \\
\\
f & N =: v\ ;\ v? \triangleright x\ .\ M & \to & N =: v\ ;\ (x.M)\ N & \\
b_1 & N =: v\ ;\ w? \triangleright x\ .\ M & \to & w? \triangleright x\ .\ N =: v\ ;\ M & (v \neq w) \\
b_2 & \mathbf{var}\ v\ .\ w? \triangleright x\ .\ M & \to & w? \triangleright x\ .\ \mathbf{var}\ v\ .\ M & (v \neq w) \\
\\
p_c & \mathbf{pure}\ (S[\mathbf{return}\ c^n\ M_1 \ldots M_k]) & \to & c^n\ (\mathbf{pure}\ (S[\mathbf{return}\ M_1])) \ldots (\mathbf{pure}\ (S[\mathbf{return}\ M_k])) & (k \leq n) \\
p_\lambda & \mathbf{pure}\ (S[\mathbf{return}\ x.M]) & \to & x\ .\ \mathbf{pure}\ (S[\mathbf{return}\ M]) & \\
p_f & \mathbf{pure}\ (S[\mathbf{return}\ f]) & \to & f &
\end{array}
$$

Figure 3: Reduction rules for λ_{var}.

Rule ($v\triangleright$) extends the scope of a tag over a ($\triangleright$) to the right. Variable capture is prevented by the hygiene condition (bound and free variables are always different). Rule ($=:\triangleright$) passes (), the result value of an assignment, to the term that follows the assignment.

Rules (f), (b_1), and (b_2) deal with assignments. The fusion rule (f) reduces a pair of an assignment and a dereference with the same tag. The bubble rules (b_1) and (b_2) allow variable-readers to "bubble" to the left past assignments and introductions involving other tags. Note that bubble and fusion reductions are defined only on tags v whereas the corresponding productions $M_v?$ and $M_1 =: M_v$ in the context-free syntax (Figure 1) admit arbitrary terms in place of M_v. This is a consequence of tags being first class, for even if M_v is not a tag it might still be reducible to one.

The final three rules implement "effect masking", by which local state manipulation can be isolated for use in a purely functional context. These three rules can be applied only if the argument to **pure** is of form $S[\mathbf{return}\ V]$ where V is a value and S is a state prefix. The context-condition ($wr\ S \subseteq bv\ S$) for state prefixes S ensures that evaluation of the argument to **pure** neither affects nor observes global storage. Effect masking "pushes state inwards", and thus exposes the outermost structure of the result of the **pure** expression. In the special cases where the result is a basic constant or primitive function the state disappears altogether.[3]

[3] Initially, we studied a calculus that had only one effect masking rule: A context **pure** ($S[\mathbf{return}$ [ ]]) can be dropped if global storage is unaffected and none of the variables bound in S appear in the term in the hole. This approach looks simpler at first glance, but it is not clear how a standard evaluation function for the resulting calculus can be constructed.

Example 2.1 (Counters) To illustrate the syntax and reduction semantics of λ_{var}, we construct a function to generate counter objects. The generated counters encapsulate an accumulator *cnt*. They export a function that takes an increment (*inc*) and yields the state transformer that adds *inc* to the "current" value of *cnt* while returning *cnt*'s "old" value. This is expressed in λ_{var} as follows (with layout indicating grouping):

$$
\begin{array}{l}
mkcounter\ = \\
\qquad initial\ .\ \mathbf{var}\ cnt\ . \\
\qquad\qquad initial =: cnt\ ; \\
\qquad\qquad \mathbf{return}\ inc\ .\ \ cnt? \triangleright \\
\qquad\qquad\qquad c\ .\ \ c + inc =: cnt\ ; \\
\qquad\qquad\qquad\quad \mathbf{return}\ c
\end{array}
$$

The reduction rules define a reduction relation between terms in the usual way: we take $\to$ to be the smallest relation on $\Lambda_{var} \times \Lambda_{var}$ that contains the rules in Figure 3 and that, for any context C, is closed under the impliciation

$$M \to N \ \Rightarrow\ C[M] \to C[N].$$

The sample reduction given in Figure 4 illustrates the use of *mkcounter* in a program that defines a counter

$$
\begin{array}{ll}
\underline{\mathit{mkcounter}\ 0} \triangleright \mathit{ctr}\ .\ \mathit{ctr}\ 1\ ;\ \mathit{ctr}\ 0 & \xrightarrow[\beta]{} \\
\underline{(\mathbf{var}\ \mathit{cnt}\ .\ 0 =: \mathit{cnt}\ ;\mathbf{return}\ \mathit{CTR}) \triangleright \mathit{ctr}\ .\ \mathit{ctr}\ 1\ ;\ \mathit{ctr}\ 0} & \xrightarrow[v\triangleright]{} \\
\mathbf{var}\ \mathit{cnt}\ .\ \underline{(0 =: \mathit{cnt}\ ;\mathbf{return}\ \mathit{CTR}) \triangleright \mathit{ctr}\ .\ \mathit{ctr}\ 1\ ;\ \mathit{ctr}\ 0} & \xrightarrow[\triangleright\triangleright]{} \\
\mathbf{var}\ \mathit{cnt}\ .\ 0 =: \mathit{cnt}\ ;\underline{\mathbf{return}\ \mathit{CTR} \triangleright \mathit{ctr}\ .\ \mathit{ctr}\ 1\ ;\ \mathit{ctr}\ 0} & \xrightarrow[r\triangleright]{} \\
\mathbf{var}\ \mathit{cnt}\ .\ 0 =: \mathit{cnt}\ ;\ \underline{(\mathit{ctr}\ .\ \mathit{ctr}\ 1\ ;\ \mathit{ctr}\ 0)\ \mathit{CTR}} & \xrightarrow[\beta]{} \\
\mathbf{var}\ \mathit{cnt}\ .\ 0 =: \mathit{cnt}\ ;\ \underline{\mathit{CTR}\ 1\ ;\ \mathit{CTR}\ 0} & \equiv \\
\mathbf{var}\ \mathit{cnt}\ .\ 0 =: \mathit{cnt}\ ;\ \underline{(\mathit{inc}\ .\ \mathit{cnt}? \triangleright c\ .\ c + \mathit{inc} =: \mathit{cnt}\ ;\mathbf{return}\ c)\ 1\ ;\ \mathit{CTR}\ 0} & \xrightarrow[\beta]{} \\
\mathbf{var}\ \mathit{cnt}\ .\ \underline{0 =: \mathit{cnt}\ ;\ \mathit{cnt}? \triangleright c\ .\ c + 1 =: \mathit{cnt}\ ;\mathbf{return}\ c\ ;\ \mathit{CTR}\ 0} & \xrightarrow[f]{} \\
\mathbf{var}\ \mathit{cnt}\ .\ 0 =: \mathit{cnt}\ ;\ \underline{(c\ .\ c + 1 =: \mathit{cnt}\ ;\mathbf{return}\ c)\ 0\ ;\ \mathit{CTR}\ 0} & \xrightarrow[\beta]{} \\
\mathbf{var}\ \mathit{cnt}\ .\ 0 =: \mathit{cnt}\ ;\ 0 + 1 =: \mathit{cnt}\ ;\ \underline{\mathbf{return}\ 0\ ;\ \mathit{CTR}\ 0} & \xrightarrow[r\triangleright]{} \\
\mathbf{var}\ \mathit{cnt}\ .\ 0 =: \mathit{cnt}\ ;\ 0 + 1 =: \mathit{cnt}\ ;\ \underline{\mathit{CTR}\ 0} & \twoheadrightarrow \\
\mathbf{var}\ \mathit{cnt}\ .\ 0 =: \mathit{cnt}\ ;\ 0 + 1 =: \mathit{cnt}\ ;\ 0 + 1 + 0 =: \mathit{cnt}\ ;\mathbf{return}\ 1 &
\end{array}
$$

Figure 4: A sample reduction.

ctr, increments it, and then inspects the final value. We use the abbreviation $\mathit{CTR} \equiv \mathit{inc}\ .\ \mathit{cnt}? \triangleright c\ .\ c + \mathit{inc} =: \mathit{cnt}\ ;\mathbf{return}\ c$. For each step in the reduction, the redex for the next reduction is underlined. Other reduction sequences are possible as well, but they all yield the same normal form, since λ_{var} is Church-Rosser (Section 3).

3 Fundamental Theorems

In this section, we establish that our calculus has the fundamental properties that make it suitable as a basis for reasoning about programs. We first show that reduction is confluent; we then derive from the reduction relation a theory λ_{var} for equational reasoning about Λ_{var} terms. We also derive from the reduction relation an evaluation function that takes programs to answers. We conclude by showing that the evaluation function is a semi-decision procedure for equations between programs and answers. Due to space limitations, most proofs are sketched or omitted; full proofs can be found in [14].

In the sequel, let $\underset{!}{\rightarrow}$ be the union of all reductions in Figure 3 except (β) and (δ).

Proposition 3.1 $\underset{!}{\rightarrow}$ is strongly normalizing: every sequence of $\underset{!}{\rightarrow}$-reductions terminates.

Proof: A standard termination measure argument. ■

Proposition 3.2 $\underset{!}{\rightarrow}$ is Church-Rosser: if $M \underset{!}{\twoheadrightarrow} M_1$ and $M \underset{!}{\twoheadrightarrow} M_2$ then there exists M_3 such that $M_1 \underset{!}{\twoheadrightarrow} M_3$ and $M_2 \underset{!}{\twoheadrightarrow} M_3$.

Proof: A case analysis on the relative positions of redexes coupled with a case analysis on reduction rules shows that $\underset{!}{\rightarrow}$ is weakly Church-Rosser. The proposition then follows by Newman's lemma ([1], Proposition 3.1.25) and Proposition 3.1. ■

This leads us to the confluence result of the full reduction relation:

Theorem 3.3 $\rightarrow$ is Church-Rosser.

Proof: The purely functional reduction relation $\xrightarrow[\beta\delta]{}$ is easily shown to be Church-Rosser (using Mitschke's theorem ([1], Theorem 15.3.3), for instance). By Proposition 3.2, $\underset{!}{\rightarrow}$ is Church-Rosser. A straightforward case analysis on the relative positions of redexes establishes that $\underset{!}{\rightarrow}$ commutes with $\xrightarrow[\beta\delta]{}$. By the lemma of Hindley and Rosen ([1], Proposition 3.3.5), the combined notion of reduction $\rightarrow$ is Church-Rosser. ■

Reduction gives rise in the standard way to an equational theory. As usual, we define equality (=) to be the smallest equivalence relation that contains reduction.

Definition. The theory λ_{var} has as formulas equations $M = N$ between terms $M, N \in \Lambda_{var}$. Equality (=) is the smallest equivalence relation between terms that contains reduction ($\rightarrow$).

We now define a computable procedure, or *evaluation function*, that maps a program to an answer if it reduces to one. We define our evaluation function via a context machine. At every step, a context machine separates its argument term into a *head redex* that occupies a uniquely-determined *evaluation context* and then performs a reduction on the redex. Evaluation stops once the argument is an answer.

Evaluation contexts for λ_{var} are defined as follows:

$$\begin{array}{rcl} E & ::= & [\,] \mid E\,M \mid f\,E \\ & \mid & \mathbf{var}\ v.E \mid E? \mid M =: E \\ & \mid & E \triangleright x.M \mid M =: v \triangleright x.E \\ & \mid & \mathbf{pure}\ E \mid \mathbf{pure}\ S[\mathbf{return}\ E] \end{array}$$

The productions on the first line generate evaluation contexts in classical λ-calculus with constants; the other productions deal with the evaluation of state transformers.

Given a λ_{var}-term, a step of the evaluation function starts at the root of the term. If it is a redex, it is reduced; otherwise, the term's abstract syntax tree is matched against the E-productions, and the subterm occupying the position of the E is recursively checked. If no redex is found, evaluation stops; otherwise the process is repeated.

Definition. A redex Δ is a *left redex* of a Λ_{var} term M if $M \equiv E[\Delta]$, for some evaluation context E. A left redex Δ of M is the *head redex* of M if, for all left redexes Δ' of M, $\Delta' \subseteq \Delta$.

Definition. The evaluation function $eval_{var}$ on Λ_{var} programs is defined as follows:

$$\begin{array}{rcll} eval\ E[M] & = & eval\ E[N] & \text{if } M \text{ is head redex} \\ & & & \text{in E[M] and } M \to N, \\ eval\ A & = & A. & \end{array}$$

What is the relation between λ_{var} and $eval_{var}$? We can show (by adapting a proof of the Curry-Feys standardization theorem in [1], Section 11.4) that $eval_{var}$ is a semi-decision procedure for equations in λ_{var} of the form $M = A$ where M is a program and A is an answer (a constant c^0):

Theorem 3.4 (Correspondence) For every closed term $M \in \Lambda_{var}$ and answer A,

$$\lambda_{var} \vdash M = A \Leftrightarrow eval_{var}\ M \equiv A.$$

4 Simulation by a Single-Threaded Store

We now show that assignments in λ_{var} can be implemented using a single sequentially-accessed store. In order to do this, we define a translation from λ_{var} into another calculus, λ_σ, that represents stores explicitly. This calculus has reduction rules that closely resemble the usual meanings of store-operations in imperative models of computation; furthermore, we can define an evaluation function on the language Λ_σ that evaluates sequences of such operations in the expected temporal order. We establish that the evaluation functions for λ_σ and λ_{var} agree on those terms that are present in both languages. This simulation result shows both that λ_{var} possesses a reasonable implementation as a programming language and also that λ_{var} indeed reasons about assignment as claimed.

To form the new term language Λ_σ, we make stores explicit by extending the defining grammar of Λ_{var} (Figure 1) with the additional production $M ::= \sigma \cdot M$. Here, $\sigma = \{v_1 : M_1, ..., v_n : M_n\}$ is a *state*, represented by a set of pairs $v : M$ of tags v and terms M. $dom\ \sigma = \{v_1, ..., v_n\}$ is called the domain of σ. Tags in $dom\ \sigma$ are considered to be bound by σ.

Reduction rules for states are derived from the reduction rules of λ_{var}, with the following modifications: We keep (β) and (δ) reduction as well as the flattening rules ($\triangleright\triangleright$), ($r\triangleright$), ($v\triangleright$), ($=:\triangleright$). We replace the remaining bubble, fusion, and effect masking rules by rules that construct, access, update, and destroy states, as shown in Figure 5. The new basic constant **undef** is used to flag an unitialized variable. The rules in Figure 5 define a reduction relation λ_σ between terms in Λ_σ. This relation can be shown to be confluent:

Theorem 4.1 $\to_\sigma$ is Church-Rosser.

Note that, even though a state σ can be duplicated in rule σ_{pc}, the resulting states are all read-only. Therefore it suffices to copy a pointer to the state instead of the state itself: state in λ_σ is *single-threaded* [17].

The evaluation contexts in λ_σ are given by the grammar:

$$\begin{array}{rcl} E & ::= & [\,] \mid E\,M \mid f\,E \\ & \mid & E? \mid M =: E \\ & \mid & E \triangleright x.M \mid \\ & \mid & \mathbf{pure}\ E \mid \mathbf{pure}\ S[\mathbf{return}\ E] \\ & \mid & \sigma \cdot E \end{array}$$

Based on this definition of evaluation context, we define the notion of head redex and the standard evaluation

$$
\begin{array}{llcll}
\sigma_{var} & \sigma \ \cdot \ \mathbf{var}\ v.M & \to & \sigma \cup \{v : \mathbf{undef}\} \ \cdot \ M & \\
\sigma_{=:} & \sigma \cup \{v : N'\} \ \cdot \ N =: v \ ; M & \to & \sigma \cup \{v : N\} \ \cdot \ M & \\
\sigma_{?} & \sigma \cup \{v : N\} \ \cdot \ v? \triangleright x.M & \to & \sigma \cup \{v : N\} \ \cdot \ (x.M)\ N & (N \not\equiv \mathbf{undef}). \\
 & & & & \\
\sigma_{pure} & \mathbf{pure}\ M & \to & \emptyset \ \cdot \ M & \\
\sigma_{p_c} & \sigma \cdot \mathbf{return}\, [c^n\ M_1 \ldots M_k] & \to & c^n\ (\sigma \cdot \mathbf{return}\, [M_1]) \ldots (\sigma \cdot \mathbf{return}\, [M_k]) & (k \le n) \\
\sigma_{p_\lambda} & \sigma \cdot \mathbf{return}\, [x.M] & \to & x \,.\, \sigma \cdot \mathbf{return}\, [M] & \\
\sigma_{p_f} & \sigma \cdot \mathbf{return}\, [f] & \to & f &
\end{array}
$$

Figure 5: Modified reduction rules for λ_σ.

function $eval_\sigma$ for programs in Λ_σ as was done for Λ_{var} in Section 3. $eval_\sigma$ closely corresponds to usual notions of store-based computations with store access and update as single reduction steps. Analogously to the situation in λ_{var}, $eval_\sigma$ is a semi-decision procedure for equations between terms and answers in Λ_σ.

Theorem 4.2 (Correspondence) For every closed term $M \in \Lambda_\sigma$ and answer A,

$$\lambda_\sigma \ \vdash \ M = A \ \Leftrightarrow \ eval_\sigma\ M \equiv A.$$

Since $\Lambda_{var} \subset \Lambda_\sigma$, it makes sense to apply $eval_\sigma$ to a term in Λ_{var}. Moreover, both evaluation functions are equivalent if we consider only observable results:

Theorem 4.3 (Simulation) For every closed term M in Λ_{var} and answer A,

$$\lambda_{var} \ \vdash \ M = A \ \Leftrightarrow \ \lambda_\sigma \ \vdash \ M = A.$$

Proof: There is a close correspondence between states in λ_σ and state prefixes in λ_{var}. Every state prefix S corresponds to a state σ_S, defined by

$$
\begin{array}{l}
dom\ \sigma_S = bv\ S \\
S \equiv S'[N =: v \ ; C[\,]], v \notin wr\ C \ \Rightarrow \ (v : N) \in \sigma_S \\
v \in bv\ S, v \notin wr\ S \ \Rightarrow \ (v : \mathbf{undef}) \in \sigma_S
\end{array}
$$

Define $\mathcal{S}[\![\mathbf{pure}\ (S[M])]\!] = \sigma_S \cdot M$ and extend $\mathcal{S}$ canonically to all of Λ_{var}. S is surjective but not injective: every non-empty state corresponds to an infinite number of state prefixes. We define a right inverse $\mathcal{S}^{-1}$ of $\mathcal{S}$ by picking for each state σ one of the state prefixes that corresponds to σ. Assume that tag identifiers are totally ordered, and that the identifiers $v_1, \ldots, v_n$ in a state $\sigma = \{v_1 : M_1, \ldots, v_n : M_n\}$ form an ascending sequence. Define

$$
\begin{array}{rcl}
\mathcal{S}^{-1}[\![\sigma \cdot M]\!] & \equiv & \mathbf{pure}\,(\mathbf{var}\ v_1. \ \ldots \ \mathbf{var}\ v_n. \\
 & & \quad M_1 =: v_1 \ ; \ \ldots \ ; M_n =: v_n \ ; M).
\end{array}
$$

and extend $\mathcal{S}^{-1}$ canonically to a mapping from Λ_σ to Λ_{var}. It is straightforward to verify that

(i) $M \underset{\sigma}{\twoheadrightarrow} \mathcal{S}[\![M]\!]$,

(ii) $\lambda_\sigma \ \vdash \ \mathcal{S}[\![\mathcal{S}^{-1}[\![\sigma]\!]]\!] = \sigma$,

(iii) $\lambda_{var} \models \mathcal{S}^{-1}[\![\mathcal{S}[\![M]\!]]\!] \cong M$ (operational equivalence $\cong$ is defined in the next section).

Using these laws, one shows by a case analysis over the respective notions of reduction in λ_{var} and λ_σ that

(iv) If $M \to N$ by contracting a head redex Δ in M, and $N \twoheadrightarrow A$ then $\lambda_\sigma \ \vdash \ \mathcal{S}[\![M]\!] = \mathcal{S}[\![N]\!]$,

(v) If $M \underset{\sigma}{\to} N$ then $\lambda_{var} \ \vdash \ \mathcal{S}^{-1}[\![M]\!] \cong \mathcal{S}^{-1}[\![N]\!]$.

The theorem then follows from laws $(i - v)$ by an induction on the length of the reduction sequence from M to A. ■

5 Operational Equivalence

Operational equivalence is intended to reflect the notion of interchangeability of program fragments. It equates strictly more terms than does convertibility. We will define operational equivalence for arbitrary extensions of the λ-calculus.

Definition. An equational theory λ_* over terms in Λ_* is an *extension* of λ (wrt conversion), if $\Lambda \subseteq \Lambda_*$, and, for any terms M, N in Λ,

$$\lambda \ \vdash \ M = N \ \Rightarrow \ \lambda_* \ \vdash \ M = N$$

An extension is *conservative* if the implication in the last statement can be strengthened to an equivalence.

Definition. Let λ_* be some extension of the λ-calculus. Two terms N and M are *operationally equivalent* in λ_*, written $\lambda_* \models N \cong M$, if for all contexts C in Λ_* such that $C[M]$ and $C[N]$ are closed, and for all answers A,

$$\lambda_* \vdash C[N] = A \quad \Leftrightarrow \quad \lambda_* \vdash C[M] = A.$$

Lemma 5.1 For any terms M, N, and context C,

$$\lambda_* \vdash M \cong N \Leftrightarrow \lambda_* \vdash C[M] \cong C[N].$$

Proposition 5.2 The following are operational equivalences in λ_{var}:

$$\begin{array}{lrcl}
(1) & v?\triangleright x\ .\ w?\triangleright y\ .\ M & \cong & w?\triangleright y\ .\ v?\triangleright x\ .\ M \\
(2) & N =: v\ ;\ N' =: w\ ;\ M & \cong & N' =: w\ ;\ N =: v\ ;\ M \\
 & & & (v \neq w) \\
(3) & \mathbf{var}\ v\ .\ N =: w\ ;\ M & \cong & N =: w\ ;\mathbf{var}\ v\ .\ M \\
 & & & (v \neq w, v \notin fv\ N) \\
(4) & \mathbf{var}\ v\ .\ \mathbf{var}\ w\ .\ M & \cong & \mathbf{var}\ w\ .\ \mathbf{var}\ v\ .\ M \\
(5) & N =: v\ ;\ N' =: v\ ;\ M & \cong & N' =: v\ ;\ M \\
(6) & S[S'[M]] & \cong & S'[M] \quad (S' \not\equiv [\,], \\
 & & & bv\ S \cap fv\ S'[M] = \emptyset)
\end{array}$$

Proof: One uses the correspondence and simulation results of Sections 3 and 4, together with an induction on the definition of $eval_\sigma$. ■

Equation (1) says that variable lookups commute. Equations (2), (3) and (4) say that assignments and variable introductions commute with themselves and with each other. Equation (5) says that if a variable is written twice in a row, the second assigned value is the one that counts.

Equation (6) implements "garbage collection": it says that a state prefix S of an expression $S[S'[M]]$ can be dropped if every variable written in S is unused in $S'[M]$. The reason for the second state prefix S' is to prevent false operational equivalences involving nonsense terms, as in **var** $v.1 \not\cong 1$. Note that, using the "bubble" conversion laws and the commutative laws (2), (3) and (4), garbage can always be moved to a state prefix.

Relationship between λ_{var} and classical λ-calculus. Clearly, convertibility in λ implies convertibility in λ_{var}, since (β) and (δ) are reduction rules in λ_{var}. However, this goes only part of the way. For instance, the equation $tail \circ cons\ x \cong id$ between list processing functions is not an equality in the sense of $\beta\delta$-convertibility, but it *is* an operational equivalence. Other operational equivalences are those that identify some diverging terms or terms that involve fixpoints. Since equivalences like these are routinely used when reasoning about programs, we would like them to be preserved in λ_{var}. We establish now the result that λ_{var} indeed preserves the operational equivalences of λ, and, furthermore, that λ_{var} does not introduce any new operational equivalences between Λ-terms. The only provision on this result is that the underlying set of constructors and basic function symbols needs to be "sufficiently rich" (meaning that we can always find enough constructors that are not used in the reduction of some given program).

Definition. An (extension of) applied λ calculus λ_* has a *sufficiently rich* set of constants if

(a) The constructor alphabet includes for every arity n an infinite number of constructors that do not form part of any of the terms N_{f,c^n}, N_f used to define the δ function.

(b) For every type constructor c^n one can define in λ_* a projection function $proj\_c^n$ such that

$$\begin{array}{lcl}
proj_c^n\ (c^n\ M_1 \ldots M_n)\ P\ Q & = & P\ M_1 \ldots M_n \\
proj_c^n\ V\ P\ Q & = & Q\ V \\
\quad \text{for any other value } V. & &
\end{array}$$

(c) One can define in λ_* a function projector $proj\_f$ such that

$$\begin{array}{lcl}
proj_f(c^n\ M_1 \ldots M_n)\ P\ Q & = & Q\ (c^n\ M_1 \ldots M_n) \\
\quad \text{for any data value } c^n\ M_1 \ldots M_n & & \\
proj_f\ V\ P\ Q & = & P\ V \\
\quad \text{for any non-data value } V \text{ (i.e. for any function).} & &
\end{array}$$

Clearly, these projection functions can be defined by suitable δ-rules. The functions $proj\_c^n$ represent a stripped down version of pattern matching on data types, as it is found in many functional programming languages. Function $proj\_f$ can be thought to be a dynamic type test, similar to `procedure?` in Scheme.

Theorem 5.3 (Conservative Extension) Assume that λ and λ_{var} have the same, sufficiently rich set of constants. Then for any two terms $M, N \in \Lambda$,

$$\lambda \models M \cong N \Leftrightarrow \lambda_{var} \models M \cong N.$$

Proof: The proof is based on finding a *syntactic embedding* $\mathcal{F}$ from the store-based calculus Λ_σ to terms in Λ.

Definition. Let λ_* be an extension of λ. Let $\mathcal{R}$ be an unspecified domain of *environments*. A mapping $\mathcal{E} : \Lambda_* \to \mathcal{R} \to \Lambda$ is a *syntactic embedding* from λ_* to λ if $\mathcal{E}$ is compositional[4], i.e.

$$\begin{array}{l}\forall C \in \Lambda_*[\,]\ \forall \rho \in \mathcal{R}\ \exists \rho' \in \mathcal{R}\ \forall M \in \Lambda_* \\ \qquad \lambda\ \vdash\ \mathcal{E}[\![C[M]]\!]\rho = (\mathcal{E}[\![C]\!]\rho)[\mathcal{E}[\![M]\!]\rho'],\end{array}$$

$\mathcal{E}$ is the identity on Λ programs, i.e. for all closed $M \in \Lambda$, $\rho \in \mathcal{R}$,

$$\lambda\ \vdash\ \mathcal{E}[\![M]\!]\rho = M,$$

and $\mathcal{E}$ is semantics preserving, i.e.

$$\lambda_*\ \vdash\ M = A \Leftrightarrow \lambda\ \vdash\ \mathcal{E}[\![M]\!]\rho = A.$$

For technical reasons, we use a variant of λ_σ, in which states are represented as sequences of bindings $v : M$, rather than as sets of such bindings. The reduction rules in Figure 5 carry over, except that the first three rules are now defined on sequences rather than sets:

$$\begin{array}{l}\sigma\ \cdot\ \mathbf{var}\ v.M \quad \to \\ \qquad \sigma +\!\!+ [v : \mathbf{undef}]\ \cdot\ M \\ \sigma +\!\!+ [v : N'] +\!\!+ \sigma'\ \cdot\ N =: v\ ;\ M \quad \to \\ \qquad \sigma +\!\!+ [v : N] +\!\!+ \sigma'\ \cdot\ M \\ \sigma +\!\!+ [v : N] +\!\!+ \sigma'\ \cdot\ v? \triangleright x.M \quad \to \\ \qquad \sigma +\!\!+ [v : N] +\!\!+ \sigma'\ \cdot\ (x.M)\ N \quad (N \not\equiv \mathbf{undef}).\end{array}$$

where $+\!\!+$ is the append operator on lists. Clearly, Theorem 4.3 holds for the new just as for the original λ_σ calculus. Assuming for the moment that we have found a syntactic embedding $\mathcal{F}$ from the new λ_σ to λ, we can then prove Theorem 5.3 as follows:

"$\Rightarrow$": Assume that $\lambda \models M \cong N$ and let A be an answer. Then, for all λ-contexts C_λ such that $C_\lambda[M]$ and $C_\lambda[N]$ are closed:

$$\lambda\ \vdash\ C_\lambda[M] = A \Leftrightarrow \lambda\ \vdash\ C_\lambda[N] = A.$$

Assume first that both M and N are closed. Let C be an arbitrary closed λ_{var}-context and let ρ be in the environment domain of $\mathcal{F}$. Since $\mathcal{F}$ is compositional, there exists an environment ρ' with

$$\mathcal{F}[\![C[M]]\!]\rho = (\mathcal{F}[\![C]\!]\rho)[\mathcal{F}[\![M]\!]\rho'].$$

[4] We assume that syntactic embeddings are extended canonically to contexts, e.g. $\mathcal{E}[\![[\,]]\!] = [\,]$.

Furthermore,

$$\begin{array}{ll} & \lambda_{var}\ \vdash\ C[M] = A \\ \Leftrightarrow & \text{(Theorem 4.3)} \\ & \lambda_\sigma\ \vdash\ C[M] = A \\ \Leftrightarrow & (\mathcal{F} \text{ is semantics preserving}) \\ & \lambda\ \vdash\ \mathcal{F}[\![C[M]]\!]\rho = A \\ \Leftrightarrow & (\mathcal{F} \text{ is compositional}) \\ & \lambda\ \vdash\ (\mathcal{F}[\![C]\!]\rho)[\mathcal{F}[\![M]\!]\rho'] = A \\ \Leftrightarrow & (\mathcal{F} \text{ is the identity on } \Lambda \text{ programs}) \\ & \lambda\ \vdash\ (\mathcal{F}[\![C]\!]\rho)[M] = A \\ \Leftrightarrow & (\text{premise: } \lambda \models M \cong N) \\ & \lambda\ \vdash\ (\mathcal{F}[\![C]\!]\rho)[N] = A \\ \Leftrightarrow & \text{(reverse the argument)} \\ & \lambda_{var}\ \vdash\ C[N] = A.\end{array}$$

Now let M and N be arbitary Λ terms, with $fv\ M \cup fv\ N = \{x_1, ..., x_n\}$. Then,

$$\begin{array}{ll} & \lambda\ \vdash\ M \cong N \\ \Rightarrow & \text{(Lemma 5.1)} \\ & \lambda\ \vdash\ x_1.\ \ldots\ x_n.M \cong x_1.\ \ldots\ x_n.N \\ \Rightarrow & \text{(first part of proof)} \\ & \lambda_{var}\ \vdash\ x_1.\ \ldots\ x_n.M \cong x_1.\ \ldots\ x_n.N \\ \Rightarrow & \text{(Lemma 5.1)} \\ & \lambda_{var}\ \vdash\ M \cong N.\end{array}$$

"$\Leftarrow$". Assume $\lambda_{var} \models M \cong N$. Then we have

$$\lambda_{var}\ \vdash\ C[M] = A \Leftrightarrow \lambda_{var}\ \vdash\ C[N] = A$$

for all contexts C in Λ_{var} such that $C[M]$ and $C[N]$ are closed and therefore also for all such contexts C in Λ. Since terms $M \in \Lambda$ have only β and δ redexes, and since Λ is closed under $\beta\delta$ reduction, this implies $\lambda \models M \cong N$. ∎

The remainder of this section is devoted to the definition of the syntactic embedding $\mathcal{F}$ from λ_σ to λ. This construction is actually of a broader importance than just as a technique for the proof of conservative extension, for it also gives us a way to construct models for λ_{var}, by composing any denotational semantics of applied λ calculus with $\mathcal{F}$. We assume from now on that λ_{var} has a sufficiently rich set of constants.

$\mathcal{F}$ is defined in Figure 6. It takes as environment a stack of symbol tables. Each symbol table contains bindings for mutable and immutable variables local to some **pure** scope. (A **pure** scope extends over a subterm with outermost constructor **pure**, but excludes any nested **pure**-terms). Symbol tables are represented as sets of

$$
\begin{array}{lcl}
\mathcal{F}[\![f]\!]\ ts & = & f \\
\mathcal{F}[\![c^n]\!]\ ts & = & c^n \\
\mathcal{F}[\![x]\!]\ (t\!:\!ts) & = & \textbf{if}\ \exists M.\{x \mapsto M\} \subseteq t\ \textbf{then}\ M \\
 & & \textbf{else}\ outer\ (\mathcal{F}[\![x]\!]\ ts) \\
\mathcal{F}[\![x.M]\!]\ (t\!:\!ts) & = & y.\mathcal{F}[\![M]\!]((\{x \mapsto y\} \cup t)\!:\!ts) \\
 & & \textbf{where}\ y \notin fv(t\!:\!ts) \\
\mathcal{F}[\![M_1\ M_2]\!]\ ts & = & (\mathcal{F}[\![M_1]\!]\ ts)\ (\mathcal{F}[\![M_2]\!]\ ts) \\
\mathcal{F}[\![v]\!](t\!:\!ts) & = & \textbf{if}\ \exists M.\{v \mapsto M\} \subseteq t\ \textbf{then}\ M \\
 & & \textbf{else}\ outer\ (\mathcal{F}[\![v]\!]\ ts) \\
\mathcal{F}[\![\textbf{var}\ v\ M]\!]\ (t\!:\!ts) & = & Var \\
 & & \quad (y.\mathcal{F}[\![M]\!]((\{v \mapsto y\} \cup t)\!:\!ts)) \\
 & & \textbf{where}\ y \notin fv(t\!:\!ts) \\
\mathcal{F}[\![M?]\!]\ ts & = & Deref\ (\mathcal{F}[\![M]\!]\ ts) \\
\mathcal{F}[\![M_1 =: M_2]\!]\ ts & = & Assign\ (\mathcal{F}[\![M_1]\!]\ ts)\ (\mathcal{F}[\![M_2]\!]\ ts) \\
\mathcal{F}[\![\textbf{return}\ M]\!]\ ts & = & Return\ (\mathcal{F}[\![M]\!]\ ts) \\
\mathcal{F}[\![M_1 \rhd x.M_2]\!]\ ts & = & bind\ (\mathcal{F}[\![M_1]\!]\ ts)\ (\mathcal{F}[\![x.M_2]\!]\ ts) \\
\mathcal{F}[\![\textbf{pure}\ M]\!]\ ts & = & exec\ \epsilon\ (\mathcal{F}[\![M]\!]\ ts) \\
\mathcal{F}[\![\sigma \cdot M]\!]\ ts & = & exec\ s\ (\mathcal{F}[\![M]\!]\ (t\!:\!ts))
\end{array}
$$

$$
\begin{array}{l}
\quad \textbf{where} \\
\quad [v_1\!:\!M_1, ..., v_n\!:\!M_n] \ = \ \sigma \\
\quad s \ = \ [\mathcal{F}[\![M_1]\!]\ (t\!:\!ts), ..., \mathcal{F}[\![M_n]\!]\ (t\!:\!ts)] \\
\quad t \ = \ \{v_1 \mapsto Tag\ 0, ..., v_n \mapsto Tag\ (n-1)\}
\end{array}
$$

Figure 6: Syntactic embedding $\mathcal{F}$

bindings $x \mapsto M$ and $v \mapsto M$. The stack is implemented as a list, using ϵ for the empty list and (:) as constructor symbol.

The translation scheme mentions constructors *Var*, *Deref*, *Assign*, *Bind*, *Return*, *Tag* in Figure 6, as well as *In*, *Out*, *Undef*, which are defined later. We call these constructors *$\mathcal{F}$-internal*, and assume that they do not occur in the terms $\mathcal{F}$ is applied to. This can always be achieved by a suitable renaming since λ_{var} is sufficiently rich.

$\mathcal{F}$ maps state transformers in Λ_{var} to data structures in Λ that are then passed to one of two "interpreter" functions *bind* or *exec*. To define these functions and others used in the definition of $\mathcal{F}$, we use a functional notation similar to Haskell, rather than a formulation in terms of projection functions in order to aid legibility. Functional abstractions are still expressed as $x.M$ instead of Haskell's $\backslash x \rightarrow M$.

$$
\begin{array}{lcl}
bind\ (Bind\ x\ f)\ g & = & Bind\ x\ (y.bind\ (f\ y)\ g) \\
bind\ (Return\ x)\ g & = & g\ x \\
bind\ (Var\ f)\ g & = & Var\ (y.bind\ (f\ y)\ g)
\end{array}
$$

Intuitively, *bind* simulates λ_σ reductions ($\rhd\rhd$), ($r\rhd$), and ($v\rhd$). The remaining non-functional λ_σ reductions, which all reference state, are simulated by function *exec*.

$$
\begin{array}{l}
exec\ s\ (Var\ f) \ = \\
\qquad exec\ (s +\!\!+ [Undef])\ (f\ (Tag\ (length\ s))) \\
exec\ s\ (Bind\ (Assign\ x\ (Tag\ i))\ g) \ = \\
\qquad exec\ (take\ i\ s +\!\!+ [x] +\!\!+ (drop\ (i+1)\ s))\ (g\ ()) \\
exec\ s\ (Bind\ (Deref\ (Tag\ i))\ g) \mid s!!i \neq Undef \ = \\
\qquad exec\ s\ (g\ (s!!i)) \\
exec\ s\ (Return\ (c^n\ x_1\ ...\ x_n)) \ = \\
\qquad c^n\ (exec\ s\ (Return\ x_1))\ ...\ (exec\ s\ (Return\ x_n)) \\
exec\ s\ (Return\ f) \mid f \text{ not a data value} \\
\qquad x\ .\ exec\ s\ (Return\ (f\ x))
\end{array}
$$

In the second-to-last clause c^n ranges over all data constructors *except* those that are $\mathcal{F}$-internal. In the last clause f ranges over all non-data values (i.e. values that do not consist of a fully applied constructor at top-level). The syntax of values ensures that non-data values are always functions.

The translation scheme represents states as lists of terms, and tags as values $Tag\ i$ where i acts as an index into the "state" list[5]. This scheme poses one rather difficult problem: λ_σ uses globally unique tag names, but the representation of a tag as an index is unique only among all tags bound in the same state prefix. However, it is mandatory to be able to distinguish between tags bound in a given state prefix and tags that are free in it. Otherwise, global variable accesses and updates in a **pure** go undetected. There is no hope of finding a syntactic embedding $\mathcal{F}$ that assigns globally unique names to tags; every such mapping would have to pass a name supply *between* pure terms. This would violate the condition that $\mathcal{F}$ maps purely functional λ-terms to themselves, and hence $\mathcal{F}$ would not be a syntactic embedding.

We overcome this problem by introducing the mutually recursive functions *outer* and *inner*. Function *outer* marks occurrences of (mutable and immutable) variables in **pure** scopes other than the one in which the variables are defined. The number of *outer* operators applied to such variables equals the difference in nesting level of the **pure** scope that defines the variable and

[5]This part of the embedding is similar to the presentation of monadic state transformers in [22]

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{QEntry}\ a & = & \mathit{Cons}\ a\ (\mathit{Var}\ (\mathit{QEntry}\ a)) \\
\mathbf{type}\ \mathit{Queue}\ a & = & \{\ \mathit{put} \ :\ a \rightarrow \mathit{Proc}\ (), \\
 & & \quad \mathit{get} \ :\ \mathit{Proc}\ a, \\
 & & \quad \mathit{isempty} \ :\ \mathit{Proc}\ \mathit{Bool}\ \}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{mkqueue} & : & \mathit{Proc}\ (\mathit{Queue}\ a) \\
\mathit{mkqueue} & = & \mathbf{var}\ v\ . \\
 & & \mathbf{var}\ \mathit{front}\ .\ v =: \mathit{front}\ ; \\
 & & \mathbf{var}\ \mathit{rear}\ .\ v =: \mathit{rear}\ ; \\
 & & \mathbf{return}\ \{\ \mathit{put}\ x \ = \ \mathit{rear}? \triangleright y\ .\ \mathbf{var}\ w\ .\ \mathit{Cons}\ x\ w =: y\ ;\ w =: \mathit{rear}, \\
 & & \qquad \mathit{get} \ = \ \mathit{front}? \triangleright y\ .\ y? \triangleright \mathit{Cons}\ x\ z\ .\ z =: \mathit{front}\ ;\ \mathbf{return}\ x, \\
 & & \qquad \mathit{isempty} \ = \ \mathit{front}? \triangleright y\ .\ \mathit{rear}? \triangleright z\ .\ \mathbf{return}\ y \doteq z\ \ \}
\end{array}
$$

Figure 7: A queue implementation

$$
\begin{array}{rcl}
\mathit{mkqueue} \triangleright q\ .\ q{\downarrow}\mathit{put}\ x\ ;\ q{\downarrow}\mathit{get} \triangleright M & \cong & \mathit{mkqueue} \triangleright q\ .\ M\ x \\
q{\downarrow}\mathit{put}\ x\ ;\ q{\downarrow}\mathit{put}\ y\ ;\ q{\downarrow}\mathit{get} & \cong & q{\downarrow}\mathit{put}\ x\ ;\ q{\downarrow}\mathit{get} \triangleright z\ .\ q{\downarrow}\mathit{put}\ y\ ;\mathbf{return}\ z \\
\\
\mathit{mkqueue} \triangleright q\ .\ q{\downarrow}\mathit{isempty} \triangleright M & \cong & \mathit{mkqueue} \triangleright q\ .\ M\ \mathit{True} \\
q{\downarrow}\mathit{put}\ x\ ;\ q{\downarrow}\mathit{isempty} & \cong & q{\downarrow}\mathit{put}\ x\ ;\mathbf{return}\ \mathit{False}
\end{array}
$$

Figure 8: Axioms for an imperative queue abstract data type

the **pure** scope in which it is used. Function *inner* cancels out the effect of *outer*. The definition of these two functions is as follows:

$$
\begin{array}{lcl}
\mathit{outer}\ (\mathit{Tag}\ M) & = & \mathit{Out}\ (\mathit{Tag}\ M) \\
\mathit{outer}\ (\mathit{Out}\ M) & = & \mathit{Out}\ (\mathit{Out}\ M) \\
\mathit{outer}\ (\mathit{In}\ M) & = & M \\
\\
\mathit{inner}\ (\mathit{Tag}\ M) & = & \mathit{In}\ (M) \\
\mathit{inner}\ (\mathit{Out}\ M) & = & M \\
\mathit{inner}\ (\mathit{In}\ M) & = & \mathit{In}\ (\mathit{In}\ M)
\end{array}
$$

For every other data value $c^n\ M_1\ ...\ M_n$, including values formed from $\mathcal{F}$-internal constructors:

$$
\begin{array}{lcl}
\mathit{outer}(c^n\ M_1\ ...\ M_n) & = & c^n\ (\mathit{outer}\ M_1)\ ...(\mathit{outer}\ M_n) \\
\mathit{inner}(c^n\ M_1\ ...\ M_n) & = & c^n\ (\mathit{inner}\ M_1)\ ...(\mathit{inner}\ M_n)
\end{array}
$$

For every non-data value f:

$$
\begin{array}{lcl}
\mathit{outer}\ f & = & x.\mathit{outer}\ (f\ (\mathit{inner}\ x)) \\
\mathit{inner}\ f & = & x.\mathit{inner}\ (f\ (\mathit{outer}\ x))
\end{array}
$$

Proposition 5.4 $\mathcal{F}$ is a syntactic embedding.

Proof: It is straightforward to verify that $\mathcal{F}$ is compositional and that it maps Λ-programs to themselves. That $\mathcal{F}$ also preserves semantics is shown using a technique similar to the proof of Theorem 4.3. ■

Proposition 5.4 gives us a way to treat Λ_{var} programs as syntactic sugar for functional programs. In the terminology of [4], λ *can express* λ_{var}. One might ask why one should bother with λ_{var} at all, if all its terms can be mapped via $\mathcal{F}$ to functional values. We believe that the main reason for studying λ_{var} independently lies in its simplicity, compared to the translated image under $\mathcal{F}$. In the next section, we give an example showing how the laws of λ_{var} can help reasoning about imperative programs that previously required very complex proofs.

6 Example: Queue ADT

Figure 7 presents an imperative implementation of an abstract data type "Queue". A queue is represented as a record whose fields are closures implementing the operations *put* (i.e. append to end), *get* (remove from front) and *isempty*.

Internally, a queue is implemented by two references to a linked list of entries. Each entry has a data field and a link field. The link field is a mutable variable pointing to the next entry in the list. The last link field in the list is always uninitialized. *front* always refers to a variable that in turn either refers to the first entry in the queue, or is uninitialized, if the queue is empty. *rear* always refers to the last link field of the queue.

For conciseness we augment the basic calculus with pattern matching and records. Field selection is expressed by infix ($\downarrow$), of higher precendence than function application. Also, even though λ_{var} is untyped, we still write type declarations and function signatures in order to help understanding. *Var* α designates the type of mutable variables that contain values of type α. *Proc* α designates the type of state transformers that return results of type α.

One feature of λ_{var} not discussed so far concerns variable identity: In the last line of the example, $y \doteq z$ is intended to be true iff y and z designate the same tag. ($\doteq$) cannot be defined via (δ) since tags are not values. We define ($\doteq$) instead by adding reduction rules $v \doteq v \rightarrow \mathbf{true}$ and $v \doteq w \rightarrow \mathbf{false}$ if $v \neq w$. It is straightforward to show that this addition does not invalidate any of the results presented in earlier sections.

The implementation in Figure 7 satisfies the axioms for queues shown in Figure 8. This can be shown using λ_{var}'s conversion rules and the operational equivalences of Proposition 5.2. For the second axiom, a structural induction on terms is needed. As an example, we show in Figure 9 the proof that our implementation satisfies the first queue axiom. Even though this proof is far from short, all its steps are simple and amenable to machine-assisted proof-checking. Also, some of the proof's size is due to the detailed level of presentation. By contrast, the traditional approach to verifying programs with pointers treats pointer-threaded structures as graphs. This requires complex arguments when isomorphism between graphs needs to be shown.

7 Related Work

Hoare *et. al.* [9] present a normalizing set of equations for an imperative language with assignment, conditional and nondeterministic choice. Functional abstraction is not considered. Field [7] extends the deterministic part of their theory with shared variables. Boehm [2] gives an equational semantics for a first-order Algol-like language. In his setting, expressions have both values and effects, which are defined by different fragments of his calculus.

Felleisen, Friedman, and Hieb [5, 6] have developed a succession of calculi for reasoning about Scheme programs. Since their target programming language is call-by-value, they have based their work on the λ_V-calculus of Plotkin[15] instead of the pure λ-calculus. It is inherent in their goal of reasoning about Scheme that their theories are not a conservative extension with respect to operational equivalence of either the classical λ-calculus or of λ_V. Mason and Talcott [11, 12] have also developed equational calculi with motivations similar to those of Felleisen *et. al.* and with comparable results.

Our work was influenced in part by the Imperative Lambda Calculus (ILC) of Swarup, Reddy and Ireland [18]. Like λ_{var}, ILC assumes call-by-name and models assignment by rewriting variable uses to approach and merge with their definitions. Unlike λ_{var}, ILC is defined in terms of a three-level type system of values, references and observers. This somewhat restricts expressiveness on the imperative side: references to objects that encapsulate state cannot be expressed, and all procedures have to be formulated in continuation-passing style. Also, unlike λ_{var}, ILC is strongly normalizing, and, as a consequence, not Turing-equivalent (e.g. recursion is prohibited).

A programming language with motivation similar to that of λ_{var} is Forsythe [16]. The language distinguishes between mutable and immutable variables, and also between value expressions and commands; however, it does so by means of a refined type system that is based on intersection types. Forsythe essentially uses a two-phase semantics, in which a term is first expanded to some potentially infinite program which is then executed in a second phase. Some common programming idioms such as procedure variables do not fit in this framework and therefore cannot be expressed.

8 Conclusions and Future Work

We have extended the applied λ-calculus with assignment. We have shown that the resulting calculus is confluent, preserves all operational equivalences of the original calculus, and permits implementation by a conventional, sequentially updated, store. We hope that λ_{var} will prove useful as a framework for extending lazy functional programming languages with imperative constructs.

An important step to that goal will be the study of type systems for λ_{var}. We have intentionally kept the present treatment untyped in order that many of our results may be applied immediately to versions of λ_{var} with ar-

$$
\begin{array}{ll}
 & \mathit{mkqueue} \triangleright q \;.\; q{\downarrow}\mathit{put}\ x \;;\; q{\downarrow}\mathit{get} \triangleright M \\
= & (\text{expand } \mathit{mkqueue}) \\
 & \mathbf{var}\ v \;.\; \mathbf{var}\ \mathit{front} \;.\; v =: \mathit{front} \;;\; \mathbf{var}\ \mathit{rear} \;.\; v =: \mathit{rear} \;;\; \mathbf{return}\ Q \triangleright q \;. \\
 & q{\downarrow}\mathit{put}\ x \;;\; q{\downarrow}\mathit{get} \triangleright M \\
 & \mathbf{where}\ Q \equiv \{\mathit{put} = ..., \mathit{get} = ..., \mathit{isempty} = ...\},\ \text{as in Figure 7} \\
= & (r\triangleright \text{ on } \mathbf{return}\ Q\text{, followed by } \beta) \\
 & \mathbf{var}\ v \;.\; \mathbf{var}\ \mathit{front} \;.\; v =: \mathit{front} \;;\; \mathbf{var}\ \mathit{rear} \;.\; v =: \mathit{rear} \;; \\
 & Q{\downarrow}\mathit{put}\ x \;;\; Q{\downarrow}\mathit{get} \triangleright [Q/q]\ M \\
= & (\text{expand } Q{\downarrow}\mathit{put},\ Q{\downarrow}\mathit{get}) \\
 & \mathbf{var}\ v \;.\; \mathbf{var}\ \mathit{front} \;.\; v =: \mathit{front} \;;\; \mathbf{var}\ \mathit{rear} \;.\; v =: \mathit{rear} \;; \\
 & \mathit{rear}? \triangleright y \;.\; \mathbf{var}\ w \;.\; \mathit{Cons}\ x\ w =: y \;;\; w =: \mathit{rear} \;; \\
 & \mathit{front}? \triangleright y' \;.\; y'? \triangleright \mathit{Cons}\ x'\ z \;.\; z =: \mathit{front} \;;\; \mathbf{return}\ x' \triangleright [Q/q]\ M \\
= & (\text{fuse on } \mathit{rear}) \\
 & \mathbf{var}\ v \;.\; \mathbf{var}\ \mathit{front} \;.\; v =: \mathit{front} \;;\; \mathbf{var}\ \mathit{rear} \;.\; v =: \mathit{rear} \;; \\
 & \mathbf{var}\ w \;.\; \mathit{Cons}\ x\ w =: v \;;\; w =: \mathit{rear} \;; \\
 & \mathit{front}? \triangleright y' \;.\; y'? \triangleright \mathit{Cons}\ x'\ z \;.\; z =: \mathit{front} \;;\; \mathbf{return}\ x' \triangleright [Q/q]\ M \\
= & (\text{bubble and fuse on } \mathit{front}) \\
 & \mathbf{var}\ v \;.\; \mathbf{var}\ \mathit{front} \;.\; v =: \mathit{front} \;;\; \mathbf{var}\ \mathit{rear} \;.\; v =: \mathit{rear} \;; \\
 & \mathbf{var}\ w \;.\; \mathit{Cons}\ x\ w =: v \;;\; w =: \mathit{rear} \;; \\
 & v? \triangleright \mathit{Cons}\ x'\ z \;.\; z =: \mathit{front} \;;\; \mathbf{return}\ x' \triangleright [Q/q]\ M \\
= & (\text{bubble and fuse on } v) \\
 & \mathbf{var}\ v \;.\; \mathbf{var}\ \mathit{front} \;.\; v =: \mathit{front} \;;\; \mathbf{var}\ \mathit{rear} \;.\; v =: \mathit{rear} \;; \\
 & \mathbf{var}\ w \;.\; \mathit{Cons}\ x\ w =: v \;;\; w =: \mathit{rear} \;; \\
 & w =: \mathit{front} \;;\; \mathbf{return}\ x \triangleright [Q/q]\ M \\
\cong & (\text{rearrange, using Proposition 5.2 (2), (3), (4)}) \\
 & \mathbf{var}\ w \;.\; \mathbf{var}\ v \;.\; \mathit{Cons}\ x\ w =: v \;; \\
 & \mathbf{var}\ \mathit{front} \;.\; v =: \mathit{front} \;;\; w =: \mathit{front} \;; \\
 & \mathbf{var}\ \mathit{rear} \;.\; v =: \mathit{rear} \;;\; w =: \mathit{rear} \;; \\
 & \mathbf{return}\ x \triangleright [Q/q]\ M \\
\cong & (\text{Proposition 5.2 (5), twice}) \\
 & \mathbf{var}\ w \;.\; \mathbf{var}\ v \;.\; \mathit{Cons}\ x\ w =: v \;; \\
 & \mathbf{var}\ \mathit{front} \;.\; w =: \mathit{front} \;; \\
 & \mathbf{var}\ \mathit{rear} \;.\; w =: \mathit{rear} \;; \\
 & \mathbf{return}\ x \triangleright [Q/q]\ M \\
\cong & (\text{Proposition 5.2 (6), eliminating } \mathbf{var}\ v \;.\; \mathit{Cons}\ x\ w =: v \;;\; [\,]) \\
 & \mathbf{var}\ w \;.\; \mathbf{var}\ \mathit{front} \;.\; w =: \mathit{front} \;;\; \mathbf{var}\ \mathit{rear} \;.\; w =: \mathit{rear} \;; \\
 & \mathbf{return}\ x \triangleright [Q/q]\ M \\
= & (r\triangleright,\ x \text{ not free in Q}) \\
 & \mathbf{var}\ w \;.\; \mathbf{var}\ \mathit{front} \;.\; w =: \mathit{front} \;;\; \mathbf{var}\ \mathit{rear} \;.\; w =: \mathit{rear} \;;\; [Q/q]\ (M\ x) \\
= & (\beta, r\triangleright \text{ in reverse}) \\
 & \mathbf{var}\ w \;.\; \mathbf{var}\ \mathit{front} \;.\; w =: \mathit{front} \;;\; \mathbf{var}\ \mathit{rear} \;.\; w =: \mathit{rear} \;;\; \mathbf{return}\ Q \triangleright (M\ x) \\
= & (\text{collapse definition of } \mathit{mkqueue}\text{, using that } v \text{ and } w \text{ not free in } Q,\ M) \\
 & \mathit{mkqueue} \triangleright q \;.\; M\ x
\end{array}
$$

Figure 9: Proof of a law on queues.

bitrary descriptive type systems. Had we started out with a typed calculus instead, all our results would hold only for the particular type system used. This would result in a loss in generality, since there are many possible candidates for such a type system. In particular, there are several widely differing approaches to implementing the effect checking required by the *pure* rule (examples are [8, 13, 18, 19, 21]). By keeping λ_{var} untyped we avoid being overly specific.

Also left to future research is the investigation of variants of λ_{var}. A call-by-value variant promises to be a useful tool for reasoning about programs in existing imperative or impurely functional languages. A variant with control-operators could provide an equational theory for a language with call/cc or exceptions.

Acknowledgements

This work was supported in part by DARPA grant number N00014-91-J-4043. The second author was supported by an IBM Graduate Fellowship during the final preparation of this paper.

We thank Manfred Broy, Kung Chen, Matthias Felleisen, John Field, Uday Reddy, Vipin Swarup and Philip Wadler for discussions and comments on earlier versions of this paper.

References

[1] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[2] H.-J. Boehm. Side effects and aliasing can have simple axiomatic descriptions. *ACM Transactions on Programming Languages and Systems*, 7(4):637–655, October 1985.

[3] E. Crank and M. Felleisen. Parameter-passing and the lambda-calculus. In *Proc. 18th ACM Symposium on Principles of Programming Languages*, pages 233–244, January 1991.

[4] M. Felleisen. On the expressive power of programming languages. In N. D. Jones, editor, *ESOP '90, European Symposium on Programming*, Lecture Notes in Computer Science 432, pages 134–151. Springer-Verlag, 1990.

[5] M. Felleisen and D. P. Friedman. A calculus for assignments in higher-order languages. In *Proc. 14th ACM Symposium on Principles of Programming Languages*, pages 314–325, January 1987.

[6] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. Technical Report Rice COMP TR89-100, Rice University, June 1989. To Appear in Theoretical Computer Science.

[7] J. Field. A simple rewriting semantics for realistic imperative programs and its application to program analysis. In *PEPM'92: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 98–107, June 1992. Yale University Research Report YALEU/DCS/RR-909.

[8] J. Guzmán and P. Hudak. Single-threaded polymorphic lambda calculus. In *IEEE Symposium on Logic in Computer Science*, pages 333–343, June 1990.

[9] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, August 1987.

[10] P. Hudak and D. Rabin. Mutable abstract datatypes – or – how to have your state and munge it too. Research Report YALEU/DCS/RR-914, Yale University, Department of Computer Science, July 1992.

[11] I. Mason and C. Talcott. Programming, transforming, and proving with function abstractions and memories. In *Automata, Languages, and Programming: 16th International Colloquium*, Lecture Notes in Computer Science 372, pages 574–588. Springer-Verlag, 1989.

[12] I. Mason and C. Talcott. Equivalence in functional languages with side effects. *Journal of Functional Programming*, 1(3):287–327, July 1991.

[13] M. Odersky. Observers for linear types. In B. Krieg-Brückner, editor, *ESOP '92: 4th European Symposium on Programming, Rennes, France, Proceedings*, Lecture Notes in Computer Science 582, pages 390–407. Springer-Verlag, February 1992.

[14] M. Odersky and D. Rabin. The unexpurgated call-by-name, assignment, and the lambda-calculus. Research Report YALEU/DCS/RR-930, Department of Computer Science, Yale University, New Haven, Connecticut, October 1992.

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

[16] J. C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.

[17] D. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 5(2):299–310, 1985.

[18] V. Swarup, U. S. Reddy, and E. Ireland. Assignments for applicative languages. In J. Hughes, editor, *Proc. 5th ACM Conf. on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 523, pages 192–214. Springer-Verlag, August 1991.

[19] J.-P. Talpin and P. Jouvelot. Type, effect and region reconstruction in polymorphic functional languages. In *Workshop on Static Analysis of Equational, Functional, and Logic Programs*, Bordeaux, Oct. 1991.

[20] P. Wadler. Comprehending monads. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 61–78, June 1990.

[21] P. Wadler. Is there a use for linear logic? In *Proc. Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 255–273, June 1991. SIGPLAN Notices, Volume 26, Number 9.

[22] P. Wadler. The essence of functional programming. In *Proc. 19th ACM Symposium on Principles of Programming Languages*, pages 1–14, January 1992.