Yale University
Department of Computer Science

The Unexpurgated *Call by Name, Assignment, and the Lambda Calculus*

Revised Report[1]

Martin Odersky and Dan Rabin

Research Report YALEU/DCS/RR-930
May, 1993

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 preparation of this paper.

[1] A shorter version of this report appears in the proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 11–13, 1993, and is also available as Yale Research Report YALEU/DCS/RR-929.

The Unexpurgated *Call by Name, Assignment, and the Lambda Calculus*

*Revised Report**

Martin Odersky and Dan Rabin

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 [22, 29] or by using continuation-passing style [13]. Wadler, for example, uses the monad technique in [31] 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 [23] and Felleisen [5], we derive from the reduction rules both a theory and an evaluator, and we 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 6) 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

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 preparation of this paper.

*A shorter version of this report appears in the proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 11–13, 1993, and is also available as Yale Research Report YALEU/DCS/RR-929.

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

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

Figure 1: Syntax of λ_{var}

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. We do not impose any particular restrictions on either functions or side-effecting procedures, except for requiring that the difference between them be 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 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, as shown 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. To this kernel language the second group adds mutable variables v and a block construct **pure** M that encapsulates an imperative computation M. The third group adds the constructs for modeling imperative computations; these constructs form a separate syntactic category of *procedures P*.

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 procedures. 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 procedures. If M computes a tag, $M?$ is the procedure 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 procedure that sets that tag to M_1 and produces an ignorable value.

Composition of procedures. Procedures are composed into sequences using the expression $M_1 \triangleright x.M_2$, which is a syntactic embodiment of the monadic-semantics operation that Wadler [31] calls 'bind'. This

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

construct connects a procedure[1] M_1 with a functional abstraction $x.M_2$. It denotes the procedure 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:

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

Coercion of procedures. The λ_{var}-expression **return** M allows a pure expression M to be used as a procedure; the expression **pure** M permits (under certain conditions) the coercion of a procedure 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 procedures 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 and computation in a procedure 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) identifiers M. Likewise, *bt* M and *ft* M denote the bound and free tags in M. A term is *closed* if $fv\ M = ft\ M = \emptyset$. 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 $M \xrightarrow[R]{} N$ to express that M reduces in one R reduction step to N. $M \twoheadrightarrow_R N$ is used 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. The letter Δ stands for a redex. A superscripted arrow $M \xrightarrow{\Delta} N$ expresses that M reduces to N by contracting redex Δ in M.

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[2].

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

A *context* C is a term with a hole $[\,]$ in it. $(\cdot)$ expresses context composition, i.e.

$$(C_1 \cdot C_2)\ M \equiv C_1[C_2[M]].$$

A *pre-state prefix* is a context R that has one of the forms

$$R \ ::= \ [\,] \mid \mathbf{var}\ v.R \mid M =: v\ ;\ R.$$

The set of variables written in a pre-state prefix R, *wr* R, is the smallest set that satisfies

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

[1] Actually, any term. We expect that any term so used will *reduce* to a procedure.

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

A *state prefix* S is a pre-state prefix that satisfies the requirement that $wr\ S \subseteq bt\ S$.

We let letters L, M, N range over terms, while P, Q range over procedures, and V, W range over values.

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. Thus, for example, $M =: v\ ;\ \mathbf{var}\ v.M$ is not a legal term since it contains a bound and a free variable, both with the same name. To satisfy the hygiene condition, we need to α-rename the bound variable v in this term.

Reduction rules. Figure 3 gives the reduction rules of λ_{var}. A reduction relation between terms is the defined in the usual way: we take $\rightarrow$ 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 implication

$$M \rightarrow N \;\Rightarrow\; C[M] \rightarrow C[N].$$

Rule (β) is the usual β-rule of applied λ-calculus. 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[3] 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:

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

Procedures can be thought of as axiomatizing the laws of the Kleisli category associated with a monad [18, 31]. Two of the three laws of this monad are embodied in reduction rules: the rule $(\rhd\rhd)$ states that $(\rhd)$ is associative, and the rule $(v\rhd)$ states **return** is a left unit. The third law, stating that **return** is also a right unit, is an operational equivalence (Proposition 5.4 (7)).

Rule $(v\rhd)$ extends the scope of a tag over a $(\rhd)$ to the right — this axiomatizes the intuition that allocation is of indefinite time-extent. Variable capture is prevented by the hygiene condition (bound and free variables are always different). Rule $(=:\rhd)$ 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 ($wr\ S \subseteq bt\ S$) 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.[4]

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 procedure 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):

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

The sample reduction given in Figure 4 illustrates the use of *mkcounter* in a program that defines a counter *ctr*, increments it, and then inspects the final value. We use the abbreviation $CTR \equiv inc\ .\ cnt? \rhd c\ .\ c +$

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

[4] Initially, we studied a calculus that had only one effect masking rule: A context **pure** (S[**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.

β	$(x.M)\ N$	$\rightarrow$	$[N/x]\ M$	
δ	$f\ V$	$\rightarrow$	$\delta(f, V)$	($\delta(f, V)$ defined)
$\triangleright\triangleright$	$(M \triangleright x.N) \triangleright y.P$	$\rightarrow$	$M \triangleright x.(N \triangleright y.P)$	
$r\triangleright$	$(\mathbf{return}\ M) \triangleright x.P$	$\rightarrow$	$(x.P)\ M$	
$v\triangleright$	$(\mathbf{var}\ v.M) \triangleright x.P$	$\rightarrow$	$\mathbf{var}\ v.(M \triangleright x.P)$	
$=:\triangleright$	$(M =: N) \triangleright x.P$	$\rightarrow$	$M =: N\ ;\ [()/x]\ P$	($x \in fv\ P$)
f	$M =: v\ ;\ v? \triangleright x.P$	$\rightarrow$	$M =: v\ ;\ (x.P)\ M$	
b_1	$M =: v\ ;\ w? \triangleright x.P$	$\rightarrow$	$w? \triangleright x\ .\ M =: v\ ;\ P$	($v \neq w$)
b_2	$\mathbf{var}\ v\ .\ w? \triangleright x.P$	$\rightarrow$	$w? \triangleright x\ .\ \mathbf{var}\ v.P$	($v \neq w$)
p_c	$\mathbf{pure}\ (S[\mathbf{return}\ c^n\ M_1 \ldots M_k])$	$\rightarrow$	$c^n\ (\mathbf{pure}\ (S[\mathbf{return}\ M_1])) \ldots (\mathbf{pure}\ (S[\mathbf{return}\ M_k]))$	($k \leq n$)
p_λ	$\mathbf{pure}\ (S[\mathbf{return}\ x.M])$	$\rightarrow$	$x\ .\ \mathbf{pure}\ (S[\mathbf{return}\ M])$	
p_f	$\mathbf{pure}\ (S[\mathbf{return}\ f])$	$\rightarrow$	f	

Figure 3: Reduction rules for λ_{var}.

inc =: *cnt* ; **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 (Theorem 3.10)

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. Our main goals here are to prove that λ_{var} is Church-Rosser, which establishes that it is meaningful to regard terms as denoting answers, and to show that λ_{var} possesses a standard evaluation order, which makes it reasonable to regard the calculus as a programming language.

Our technical development here parallels the development in [1], Chapters 3 and 11. In Section 3.1 we prove a result combining the finiteness of developments for $\xrightarrow[\beta\delta]{}$ with strong normalization for $\xrightarrow[!]{}$. This result will be used to prove the Church-Rosser property in Section 3.2 and to establish the standard evaluation order in Section 3.4. In Section 3.3 we introduce the conversion theory for λ_{var}, the first of our formal theories of program equivalence.

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

3.1 Finite Developments

In this subsection, we prove that λ_{var} has the property of *finite developments*: if we mark some of a term's β- and δ-redexes, and agree to reduce only marked redexes, all such reduction sequences (which may include !-reductions) terminate. We will use this technical result both in the proof that λ_{var} is Church-Rosser and in the proof that it possesses a standard evaluation order.

First we define the term languages incorporating marked terms and the associated notions of reduction. We define extensions Λ'_{var} and Λ^*_{var} of Λ_{var} as follows:

The terms M in Λ'_{var} are given by extending the productions in Figure 1 with

$$
\begin{array}{lcl}
M & ::= & \ldots \\
 & \mid & ((x.M_1)\ M_2)_0 \\
 & \mid & (f\ M)_0
\end{array}
$$

The notion of reduction on Λ'_{var} is $! \cup \beta_0 \cup \delta_0$, where

$$
\begin{array}{llcl}
\beta_0 & ((x.M)\ N)_0 & \rightarrow & [N/x]\ M \\
\delta_0 & (f\ V)_0 & \rightarrow & \delta(f, V)
\end{array}
$$

We also define a language of weighted terms for use in the proof of finite developments. The terms of Λ^*_{var} are those of Λ'_{var}, plus weighted variables x^n and weighted function constants f^n, where $n \geq 1$. Reduction on Λ^*_{var} is defined as on Λ'_{var}, with the additional rule that in

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

Figure 4: A sample reduction.

rule β_0 substitution on weighted variables is defined by

$$[N/x]\, x^n \ \equiv\ N.$$

We now show that $\xrightarrow[!\beta_0\delta_0]{}$ is strongly normalizing on Λ'_{var}, that is, that all sequences of $!\beta_0\delta_0$-reductions terminate. To this purpose, we define a norm on Λ^*_{var} that is strictly decreasing under reduction: the norm of a term thus bounds the number of reductions in any sequence starting with that terms. The norm is defined in terms of another measure, the *multiplier* $\#M$ of a term M. Essentially, a multiplier is the number of terms connected in a chain by ($\triangleright$) operators. It is defined as follows:

$$
\begin{array}{lcll}
\#(M \triangleright x.N) & = & 1 + \#x.M & \\
\#(x.M) & = & \#M & \\
\#(\mathbf{var}\ v.M) & = & \#M & \\
\#M & = & 1 & \text{for all other terms } M
\end{array}
$$

The norm $||\cdot||$ is then defined as follows:

$$
\begin{array}{lcl}
||f|| & = & 1 \\
||f^n|| & = & n \\
||c^k|| & = & 1 \\
||x|| & = & 1 \\
||x^n|| & = & n \\
||x.M|| & = & 1 + ||M||, \text{ if } x \notin fv\ M \\
||x.M|| & = & 2 + ||M||, \text{ if } x \in fv\ M \\
||M\ N|| & = & ||M|| + ||N|| \\
||v|| & = & 1 \\
||\mathbf{var}\ v.M|| & = & 1 + ||M|| + \#M \\
||M?|| & = & ||M|| \\
||M =: N|| & = & ||M|| + ||N|| \\
||M \triangleright x.N|| & = & (1 + \#N)||M|| + ||x.N|| \\
||\mathbf{return}\ M|| & = & ||M|| \\
||\mathbf{pure}\ M|| & = & (||M|| + 1)^{(||M||+1)} \\
 & = & Eexp(||M|| + 1)
\end{array}
$$

We use the notation $Eexp(n)$ for n^n.

Definition. A term M has a *decreasing weighting* if

$$
\begin{array}{l}
((x.M')\ N)_0 \subseteq M, x^n \subseteq M \\
\quad \text{implies} \quad n \geq ||N|| \\
\text{and} \\
(f^n\ V)_0 \subseteq M, \delta(f, V) \text{ defined} \\
\quad \text{implies} \quad n + ||V|| > ||\delta(f, V)||
\end{array}
$$

Definition. A term $M \in \Lambda'_{var}$ is *completed* to a term in Λ^*_{var} by superscripting some of the bound variables and function symbols in M.

Lemma 3.1 Every term in Λ'_{var} can be completed to a term with a decreasing weighting.

Proof: easy; just work outward. ∎

Lemma 3.2 If M has a decreasing weighting, and $M \xrightarrow[!\beta_0\delta_0]{} N$ then $||M|| > ||N||$.

Proof: By a case analysis according to the form of reduction:

Case β_0

If $x \notin fv\ M$ we have $[N/x]\ M \equiv M$, so

$$\begin{aligned} & ||((x.M)\ N)_0|| \\ = \ & ||x.M|| + ||N|| \\ = \ & 1 + ||M|| + ||N|| \\ > \ & ||M|| \\ = \ & ||[N/x]\ M|| \end{aligned}$$

If $x \in fv\ M$, the fact that $((x.M)\ N)_0$ has a decreasing weighting gives us

$$\begin{aligned} & ||((x.M)\ N)_0|| \\ = \ & ||x.M|| + ||N|| \\ > \ & ||[N/x]\ M|| \end{aligned}$$

The last inequality holds because every instance of N replaces an instance of x having weight n. In fact, the definition of decreasing weighting was motivated by the need to make this case and the following case work out.

Case δ_0

In this case the redex is of the form $f^n\ V$, and we have:

$$\begin{aligned} & ||f^n\ V|| \\ = \ & ||f^n|| + ||V|| \\ > \ & ||\delta(f, V)|| \end{aligned}$$

Case $\triangleright\triangleright$

Note that $||x.M|| > ||M||$ irrespective of whether x is free in M. Thus we have

$$\begin{aligned} & ||(M \triangleright x.N) \triangleright y.P|| \\ = \ & (1 + \#(y.P))||M \triangleright x.N|| + ||y.P|| \\ = \ & (1 + \#P)((1 + \#(x.N))||M|| + ||x.N||) \\ & + ||y.P|| \\ = \ & (1 + \#P + \#N + \#P\#N)||M|| \\ & + (1 + \#P)||x.N|| + ||y.P|| \\ > \ & (1 + 1 + \#P)||M|| + (1 + \#P)||N|| \\ & + ||y.P|| \\ = \ & (1 + \#(x.N \triangleright y.P))||M|| \\ & + ||N \triangleright y.P|| \\ = \ & ||M \triangleright x.(N \triangleright y.P)|| \end{aligned}$$

Case $r\triangleright$

$$\begin{aligned} & ||(\mathbf{return}\ M) \triangleright x.P|| \\ = \ & ||M|| + ||x.P|| + \#(x.P)||M|| \\ > \ & ||M|| + ||x.P|| \\ = \ & ||(x.P)\ M|| \end{aligned}$$

Case $v\triangleright$

$$\begin{aligned} & ||(\mathbf{var}\ v.M) \triangleright x.P|| \\ = \ & (1 + \#P)||\mathbf{var}\ v.M|| + ||x.P|| \\ = \ & (1 + \#P)(1 + ||M|| + \#M) + ||x.P|| \\ = \ & 1 + \#P + (1 + \#P)||M|| + (1 + \#P)\#M \\ & + ||x.P|| \\ > \ & 1 + (1 + \#P) + (1 + \#P)||M|| + ||x.P|| \\ = \ & 1 + \#(M \triangleright x.P) + ||M \triangleright x.P|| \\ = \ & ||\mathbf{var}\ v.(M \triangleright x.P)|| \end{aligned}$$

Case $=:\triangleright$

Note that, since P has a decreasing weighting, $||[()/x]\ P|| \leq ||P||$. Thus we have:

$$\begin{aligned} & ||M =: N \triangleright x.P||,\ x \in fv\ P \\ = \ & (1 + \#(x.P))||M =: N|| + ||x.P|| \\ = \ & (1 + \#P)||M =: N|| + ||P|| + 2 \\ > \ & (1 + \#P)||M =: N|| + ||[()/x]\ P|| + 1 \\ = \ & (1 + \#P)||M =: N|| + ||z.[()/x]\ P||, \\ & \quad \text{where } z \notin fv\ P \\ = \ & ||(M =: N) \triangleright z.[()/x]\ P|| \\ = \ & ||M =: N\ ;\ [()/x]\ P|| \end{aligned}$$

Case f

Note that $\|M\| < \|M =: v\|$, so $2\|M =: v\| + \|M\| < 3\|M =: v\|$. Thus we have:

$$
\begin{aligned}
& \|M =: v \; ; \; v? \triangleright x.P\| \\
= \; & \|M =: v \triangleright z.v? \triangleright x.P\|, \\
& \quad \text{where } z \notin fv\ P \\
= \; & (1 + \#(z.v? \triangleright x.P))\|M =: v\| \\
& + \|z.v? \triangleright x.P\| \\
= \; & (1 + 1 + \#P)\|M =: v\| + 1 + \|v? \triangleright x.P\| \\
= \; & (2 + \#P)\|M =: v\| + 1 + (1 + \#P)\|v?\| \\
& + \|x.P\| \\
= \; & (2 + \#P)\|M =: v\| + 1 + (1 + \#P) \\
& + \|x.P\| \\
> \; & 3\|M =: v\| + 1 + \|x.P\| \\
> \; & 2\|M =: v\| + 1 + \|x.P\| + \|M\| \\
= \; & (1 + \#(z.(x.P)\ M))\|M =: v\| + \|z.(x.P)\ M\|, \\
& \quad \text{where } z \notin fv\ P \cup fv\ M \\
= \; & \|M =: v \; ; \; (x.P)\ M\|
\end{aligned}
$$

Case b_1

$$
\begin{aligned}
& \|M =: v \; ; \; w? \triangleright x.P\| \\
= \; & \|M =: v \triangleright z.w? \triangleright x.P\|, \quad \text{where } z \notin fv\ P \\
= \; & (1 + \#(z.w? \triangleright x.P))\|M =: v\| \\
& + \|z.w? \triangleright x.P\| \\
= \; & (1 + 1 + \#P)\|M =: v\| + 1 \\
& + \|w? \triangleright x.P\| \\
= \; & (2 + \#P)\|M =: v\| + 1 + (1 + \#(x.P))\|w?\| \\
& + \|x.P\| \\
= \; & (2 + \#P)\|M =: v\| + 1 + (1 + \#P) + \|x.P\| \\
& \ldots
\end{aligned}
$$

We first continue with the (abnormal) case in which $x \notin fv\ P$:

$$
\begin{aligned}
& \ldots \\
= \; & (2 + \#P)\|M =: v\| + 1 + (1 + \#P) \\
& + 1 + \|P\| \\
> \; & (2 + \#P) + 1 + (1 + \#P)\|M =: v\| \\
& + 1 + \|P\| \\
= \; & (2 + \#P) + \|x.M =: v \triangleright z.P\|, \\
& \quad \text{where } z \notin fv\ P \\
= \; & (1 + \#(x.M =: v \triangleright z.P))\|w?\| \\
& + \|x.M =: v \triangleright z.P\| \\
= \; & \|w? \triangleright x.M =: v \triangleright z.P\| \\
= \; & \|w? \triangleright x.M =: v \; ; \; P\|
\end{aligned}
$$

We now complete the argument for the usual case in which $x \in fv\ P$:

$$
\begin{aligned}
& \ldots \\
= \; & (2 + \#P)\|M =: v\| + 1 + (1 + \#P) \\
& + 2 + \|P\| \\
> \; & (2 + \#P) + 2 + (1 + \#P)\|M =: v\| \\
& + 1 + \|P\| \\
= \; & (2 + \#P) + 2 + (1 + \#P)\|M =: v\| \\
& + \|z.P\|, \\
& \quad \text{where } z \notin fv\ P \\
= \; & (2 + \#P) + 2 + \|M =: v \triangleright z.P\| \\
= \; & (1 + \#(x.M =: v \triangleright z.P))\|w?\| \\
& + \|x.M =: v \triangleright z.P\| \\
= \; & \|w? \triangleright x.M =: v \triangleright z.P\| \\
= \; & \|w? \triangleright x.M =: v \; ; \; P\|
\end{aligned}
$$

Case b_2

$$
\begin{aligned}
& \|\mathbf{var}\ v\ .\ w? \triangleright x.P\| \\
= \; & 1 + \#(w? \triangleright x.P) + \|w? \triangleright x.P\| \\
= \; & 1 + (1 + \#P) + (1 + \#P)\|w?\| + \|x.P\| \\
= \; & 3 + 2\#P + \|x.P\| \\
& \ldots
\end{aligned}
$$

We first continue with the case in which $x \notin fv\ P$:

$$
\begin{aligned}
& \ldots \\
= \; & 3 + 2\#P + 1 + \|P\| \\
> \; & 3 + 2\#P + \|P\| \\
= \; & 2 + \#P + 1 + \|P\| + \#P \\
= \; & (1 + \#P) + 1 + \|\mathbf{var}\ v.P\| \\
= \; & (1 + \#(x.\mathbf{var}\ v.P))\|w?\| + \|x.\mathbf{var}\ v.P\| \\
= \; & \|w? \triangleright x.\mathbf{var}\ v.P\|
\end{aligned}
$$

We now complete the argument for the case in which $x \in fv\ P$:

$$
\begin{array}{cl}
 & \ldots \\
= & 3 + 2\#P + 2 + \|P\| \\
> & 3 + 2\#P + 1 + \|P\| \\
= & 3 + \#P + 1 + \#P + \|P\| \\
= & (1 + \#P) + 2 + \|\mathbf{var}\ v.P\| \\
= & (1 + \#(x.\mathbf{var}\ v.P))\|w?\| + \|x.\mathbf{var}\ v.P\| \\
= & \|w? \triangleright x.\mathbf{var}\ v.P\|
\end{array}
$$

In the analysis of the remaining cases, which involve reductions of **pure** terms, we make use of the following fact:

For every state prefix S, there are integer constants $a_S \geq 0$, $b_S \geq 1$ such that, for all terms M,

$$\|S[M]\| = a_S + b_S\|M\|.$$

This can be shown by an easy induction over the structure of S. The remaining cases are then as follows:

Case p_c

$$
\begin{array}{cl}
 & \|\mathbf{pure}(S[c^n\ M_1\ \ldots\ M_k])\| \\
= & Eexp(1 + a_S + b_S(1 + \|M_1\| + \ \ldots\ + \|M_k\|)) \\
> & 1 + Eexp(1 + a_S + b_S\|M_1\|) + \ldots \\
 & + Eexp(1 + a_S + b_S\|M_k\|) \\
= & \|c^n\ (\mathbf{pure}\ (S[\mathbf{return}\ M_1]))\ \ldots \\
 & \quad (\mathbf{pure}\ (S[\mathbf{return}\ M_k]))\|
\end{array}
$$

Case p_λ

$$
\begin{array}{cl}
 & \|\mathbf{pure}(S[x.M])\| \\
= & Eexp(1 + a_S + b_S(1 + \|M\|)) \\
> & 1 + Eexp(1 + a_S + b_S\|M\|) \\
= & \|\mathbf{pure}\ (S[\mathbf{return}\ M])\|
\end{array}
$$

Case p_f

$$
\begin{array}{cl}
 & \|\mathbf{pure}\ (S[\mathbf{return}\ f^n])\| \\
= & Eexp(1 + a_S + b_S n) \\
> & n \\
= & \|f^n\|
\end{array}
$$

By the same argument, taking $n = 1$, one gets $\|\mathbf{pure}\ (S[\mathbf{return}\ f^n])\| > \|f\|$.

This concludes the case analysis and proves Lemma 3.2. ∎

Lemma 3.3 If $M \xrightarrow[!\beta_0\delta_0]{} N$, and M has a decreasing weighting, then M has a decreasing weighting.

Proof: If $M \xrightarrow[\beta_0]{} N$, the proposition is shown as in [1], Lemma 11.2.18(*ii*). The other cases are similar, but simpler. ∎

Proposition 3.4 On Λ'_{var} the reduction $\xrightarrow[!\beta_0\delta_0]{}$ is strongly normalizing.

Proof: As in [1], Proposition 11.2.20. ∎

Corollary 3.5 On Λ_{var} the reduction $\underset{!}{\rightarrow}$ is strongly normalizing.

Proof: Every (!)-reduction sequence in Λ_{var} is also a $(! \cup \beta_0 \cup \delta_0)$-sequence in Λ'_{var} and has (by Proposition 3.4) finite length. ∎

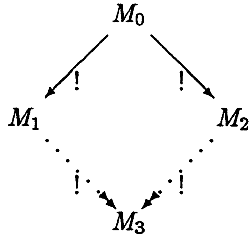
3.2 Church-Rosser

We establish that $\rightarrow$ is Church-Rosser by combining several subsidiary results. We first establish that $\underset{!}{\rightarrow}$ is Church-Rosser: this result is itself proved by using Newman's lemma to decompose the problem into a proof of strong normalization plus a proof of the weak Church-Rosser property. We then use the Hindley-Rosen lemma to deduce Church-Rosser for $\rightarrow$ from the separate Church-Rosser results for $\underset{!}{\rightarrow}$ and for $\xrightarrow[\beta\delta]{}$: this requires that we show that $\underset{!}{\twoheadrightarrow}$ commutes with $\underset{\beta\delta}{\twoheadrightarrow}$. We now begin this series of proofs.

Lemma 3.6 Let S be a state prefix and let Δ be a redex fully contained in S. If $S[M] \xrightarrow{\Delta} S'[M]$ then S' is a state prefix.

Proof: Since Δ is fully contained in S, there exists a term N, a tag v and pre-state prefixes R_1 and R_2 such that $S \equiv R_1[N =: v\ ;\ R_2]$ and $\Delta \subseteq N$. Assume $N \xrightarrow{\Delta} N'$. Then $S' \equiv R_1[N' =: v\ ;\ R_2]$ is a pre-state prefix. Moreover $bt\ S' = bt\ S$ and $wr\ S' = wr\ S$. Since S is a state prefix, $wr\ S' = wr\ S \subseteq bt\ S = bt\ S'$. Hence, S' is a state prefix. ∎

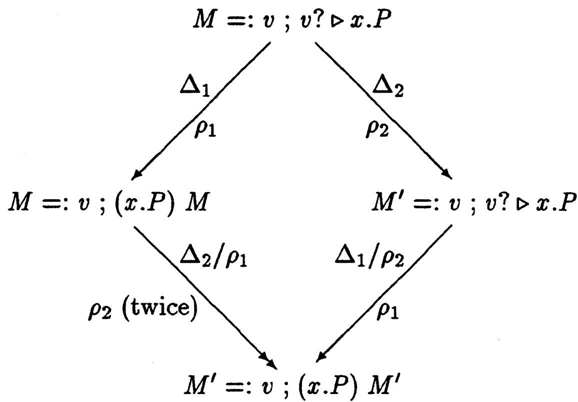
Lemma 3.7 $\underset{!}{\rightarrow}$ is weakly Church-Rosser, that is, $M_0 \underset{!}{\rightarrow} M_1$ and $M_0 \underset{!}{\rightarrow} M_2$ imply that there exists M_3 such that $M_1 \underset{!}{\twoheadrightarrow} M_3$ and $M_2 \underset{!}{\twoheadrightarrow} M_3$. The following diagram shows the situation:



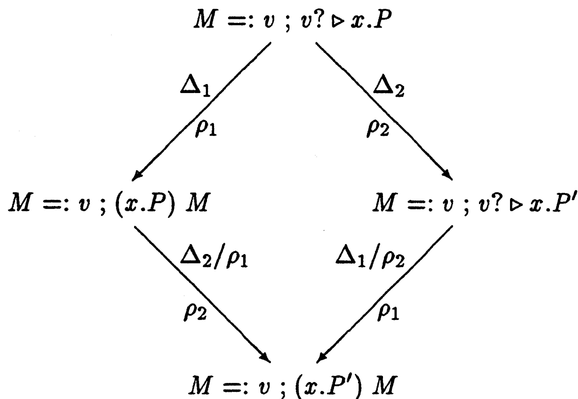
Proof: Assume $M_0 \xrightarrow[\rho_1]{\Delta_1} M_1$ and $M \xrightarrow[\rho_2]{\Delta_2} M_2$. We distinguish cases according to the relative positions of Δ_1 and Δ_2. If $\Delta_1 \equiv \Delta_2$ then $M_1 \equiv M_2 \equiv M_3$, since the left-hand sides of the rules for (!) in Figure 3 are non-overlapping. If Δ_1 and Δ_2 are disjoint, the proposition follows again from the fact that the left-hand sides of all rules in Figure 3 are non-overlapping. To show the proposition in case $\Delta_2 \subset \Delta_1$, we distinguish according to the form of Δ_1.

Case 1 (f) $\Delta_1 \equiv M =: v \; ; v? \triangleright x.P$

If $M \xrightarrow{\Delta_2} M'$, the following diagram commutes:

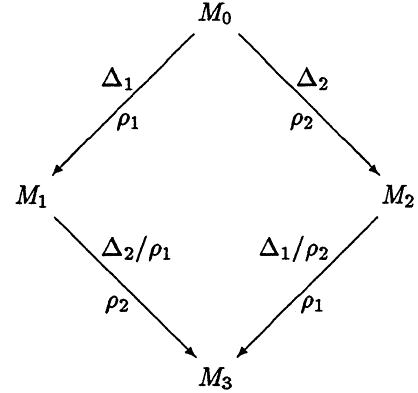


If, on the other hand, $P \xrightarrow{\Delta_2} P'$, the following diagram commutes:



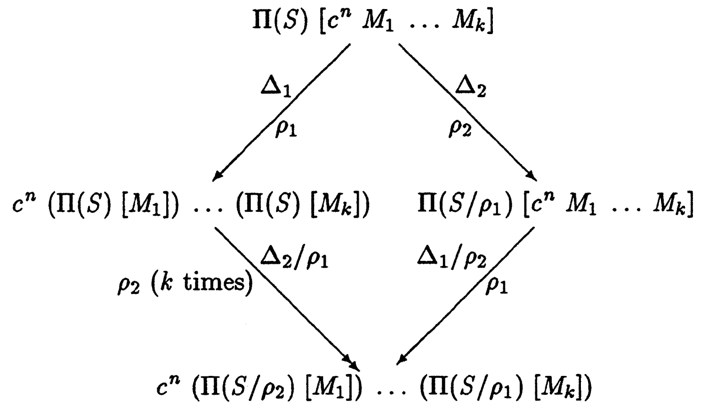
Case 2 Δ_1 is a b_1, b_2, $\triangleright\triangleright$, $r\triangleright$, $v\triangleright$ or $=:\triangleright$ redex

In these cases Δ_1 has exactly one residual with respect to ρ_2. Hence:



Case 3 (p_c) $\Delta_1 \equiv$ **pure** $(S[$**return** $c^n \; M_1 \; \ldots \; M_k]), k \leq n$

If $\Delta_2 \subseteq M_i$ for some i, $0 \leq i \leq k$, then we reason as in Case 2. If not, Δ_2 must be fully contained in S. By Lemma 3.6, S/ρ_2 is a state prefix. Hence the following diagram commutes. (We abbreviate **pure** $(S[$**return** $[\,]])$ by $\Pi(S)$).

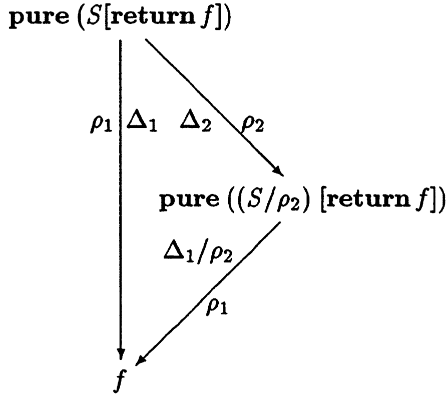


Case 4 (p_λ) $\Delta_1 \equiv$ **pure** $(S[$**return** $x.M])$

If $\Delta_1 \subseteq M$ then we reason as in Case 2. If not, Δ_1 must be fully contained in S. By Lemma 3.6, S/ρ_1 is a state prefix, and thus Δ_2/ρ_1 is a redex. Furthermore, Δ_1 has exactly one residual wrt ρ_2. From then on the proof is as in Case 2.

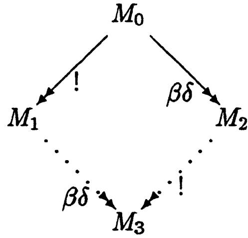
Case 5 (p_f) $\Delta_1 \equiv$ **pure** $(S[$**return** $f])$

In this case Δ_1 is fully contained in S By Lemma 3.6, S/ρ_1 is a state prefix. Hence, the following diagram commutes:

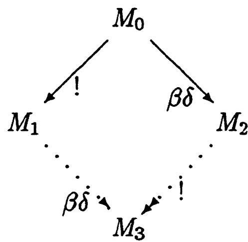


This concludes the case analysis for $\Delta_2 \subset \Delta_1$. The case $\Delta_1 \subset \Delta_2$ is handled symmetrically. ■

Lemma 3.8 $\xrightarrow[!]{}\!\!\!\!\rightarrow$ commutes with $\xrightarrow[\beta\delta]{}\!\!\!\!\rightarrow$:



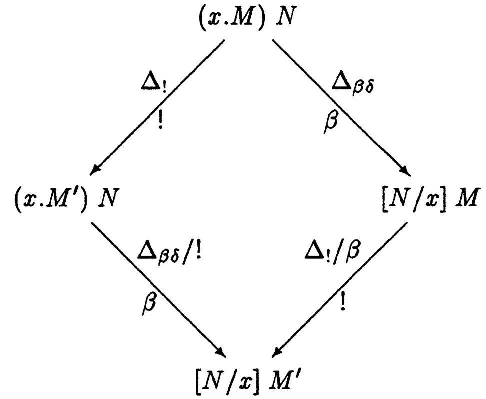
Proof: By Lemma 3.3.6 of [1], it suffices to show commutativity for the following diagram:



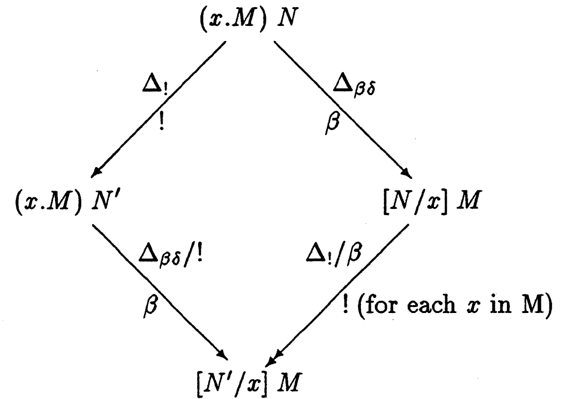
Assume then that $M_0 \xrightarrow[!]{\Delta_!} M_1$ and $M_0 \xrightarrow[\beta\delta]{\Delta_{\beta\delta}} M_2$. We distinguish cases according to the relative positions of $\Delta_!$ and $\Delta_{\beta\delta}$. Since the left-hand sides of !-reduction rules do not overlap with those of $\beta\delta$-reduction rules, $\Delta_! \not\equiv \Delta_{\beta\delta}$. If Δ_1 and $\Delta_{\beta\delta}$ are disjoint, the proposition again follows from the fact that the left-hand sides of all rules in Figure 3 are non-overlapping. To show the proposition in the case where $\Delta_! \subset \Delta_{\beta\delta}$, we distinguish cases according to the form of $\Delta_{\beta\delta}$.

Case 1 $\Delta_{\beta\delta} \equiv (x.M)\ N$

If $M \xrightarrow{\Delta_!} M'$ then the following diagram commutes:

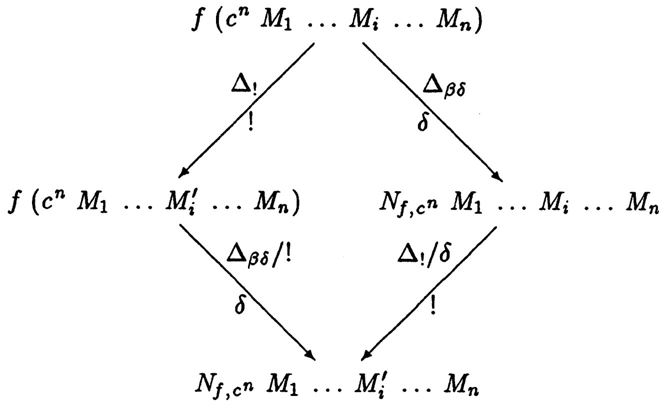


If, on the other hand, $N \xrightarrow{\Delta_!} N'$ then the following diagram commutes:



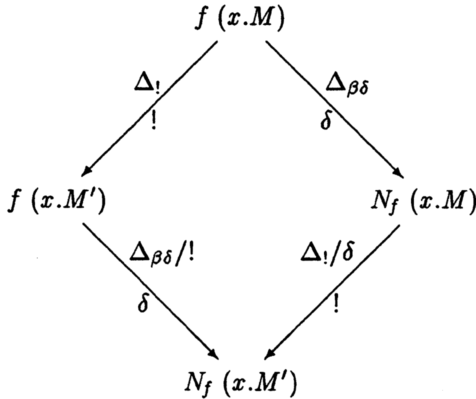
Case 2 $\Delta_{\beta\delta} \equiv f\ (c^n\ M_1\ \ldots\ M_n)$

In this case we must have $M_i \xrightarrow{\Delta_!} M_i'$, for some i, $0 \leq i \leq n$. Because of the restrictions on δ, $f\ (c^n\ M_1\ \ldots\ M_n) \xrightarrow{\Delta_{\beta\delta}} N_{f,c^n}\ M_1\ \ldots\ M_n$. Then the following diagram commutes:



Case 3 $\Delta_{\beta\delta} \equiv f\ (x.M)$

Then $M \xrightarrow{\Delta_!} M'$. Because of the restrictions on δ in Figure 3, we have $f\ (x.M) \xrightarrow{\Delta_{\beta\delta}} N_f\ (x.M)$. Then the following diagram commutes:



This concludes the case analysis for $\Delta_! \subset \Delta_{\beta\delta}$. The case $\Delta_{\beta\delta} \subset \Delta_!$ is handled as in the proof of Lemma 3.7. ■

Proposition 3.9 $\underset{!}{\rightarrow}$ is Church-Rosser: if $M_0 \underset{!}{\twoheadrightarrow} M_1$ and $M_0 \underset{!}{\twoheadrightarrow} M_2$ then there exists M_3 such that $M_1 \underset{!}{\twoheadrightarrow} M_3$ and $M_2 \underset{!}{\twoheadrightarrow} M_3$.

Proof: The relation $\underset{!}{\rightarrow}$ is weakly Church-Rosser (Lemma 3.7) and strongly normalizing (Corollary 3.5). By Newman's lemma ([1], Proposition 3.1.25), $\underset{!}{\rightarrow}$ is Church-Rosser. ■

This allows us to conclude:

Theorem 3.10 $\rightarrow$ is Church-Rosser.

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

Figure 5: Evaluation Contexts in λ_{var}

Proof: The purely functional reduction relation $\underset{\beta\delta}{\longrightarrow}$ is easily shown to be Church-Rosser (using Mitschke's theorem ([1], Theorem 15.3.3), for instance). By Proposition 3.9, $\underset{!}{\rightarrow}$ is Church-Rosser. By Lemma 3.8, $\underset{!}{\twoheadrightarrow}$ commutes with $\underset{\beta\delta}{\twoheadrightarrow}$. Then, by the lemma of Hindley and Rosen ([1], Proposition 3.3.5), the combined notion of reduction $\rightarrow$ is Church-Rosser. ■

3.3 Equational Theory

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

Definition. The theory λ_{var} has as formulas equations $M = N$ between terms $M, N \in \Lambda_{var}$. It is axiomatized as follows:

$$
\begin{array}{rcl}
M \rightarrow N & \Rightarrow & M = N \\
M & = & M \\
M = N & \Rightarrow & N = M \\
M = N, N = L & \Rightarrow & M = L
\end{array}
$$

Proposition 3.11 (Consistency of λ_{var}) There are closed terms M, N such that $\lambda_{var} \not\vdash M = N$.

Proof: Pick two different closed terms in normal form. Because $\rightarrow$ is Church-Rosser, M and N are not convertible. ■

3.4 Evaluation

An *evaluation function* is just a partial function from programs to answers. We define evaluation functions

via context machines. At every step, a context machine separates its argument term into a redex and an evaluation context and then performs a reduction on the redex. Evaluation stops once the argument is an answer. Given a notion of reduction $\rightarrow$ and a notion of "head redex", we define *eval* by:

$$
\begin{array}{lcl}
eval_{var}\ C[M] & = & eval_{var}\ C[N], \\
 & & \quad \text{if } M \text{ is head redex in } C[M] \\
 & & \quad \text{and } M \rightarrow N \\
eval\ A & = & A.
\end{array}
$$

Definition. An *evaluation context* E is a context that is of one of the forms given in Figure 5.

Definition. A redex Δ is an *evaluation redex* of a term M if $M \equiv E[\Delta]$, for some evaluation context E. The leftmost outermost evaluation redex in M is called *head redex*. Every other redex in M is called an *internal redex*.

Proposition 3.12 $eval_{var}$ is a recursive function.

Proof: Let $M \in \Lambda_{var}$. If M is an answer then $eval_{var}\ M \equiv M$. If M has a head-redex, say Δ_h, then there is a unique evaluation context E such that $M \equiv E[\Delta]$. Since no two left-hand sides of rules in Figure 3 are unifiable, there is a unique term N such that $E[\Delta] \xrightarrow{\Delta} E[N]$. In that case, $eval_{var}\ M \equiv eval_{var}\ E[N]$. If M has no head-redex, $eval_{var}\ M$ is undefined. Hence, $eval_{var}$ is a recursive function. ∎

Lemma 3.13 For any evaluation context E and redex Δ, $E[\Delta]$ has a unique head redex.

Proof: Direct from the definition of head redex. ∎

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. First, we need some rather technical lemmas that show that head-redexes are in some sense preserved by internal reductions:

Lemma 3.14 For any term M, procedure P, if $M \xrightarrow{\Delta} P$, and $M \not\equiv \Delta$, then M is a procedure.

Proof: An easy case analysis according to the form of reduction in $M \rightarrow P$. ∎

Lemma 3.15 If $M \rightarrow N$ and N has a head redex, then so has M.

Proof: Assume that N has a head redex Δ_h, that $M \xrightarrow{\Delta} N$, and that L is the result of reducing Δ. We distinguish cases according to the relative positions of L and Δ_h.

Case 1 L and Δ_h are disjoint.

In this case one can find evaluation contexts E, E', context C, bound variable x, and tag v such that M is one of the following:

(1) $E[(E'[\Delta_h])\ (C[\Delta])]$
(2) $E[C[\Delta] =: E'[\Delta_h]]$
(3) $E[E'[\Delta_h] \triangleright x.C[\Delta]]$
(4) $E[C[\Delta] \triangleright x.E'[\Delta_h]]$
where C is not an evaluation context
(5) $E[\mathbf{pure}\ (S[\mathbf{return} E'[\Delta_h]])]$
where $\Delta \subseteq S$

In each of these cases, Δ_h is an evaluation redex of M. By Lemma 3.13, M must thus have a head redex.

Case 2 $\Delta_h \subseteq L$

In this case there are contexts C_1, C_2 such that $M \equiv C_1[\Delta] \xrightarrow{\Delta} C_1[C_2[\Delta_h]]$.

Since Δ_h is head redex, $C_1 \cdot C_2$ is an evaluation context. Since evaluation contexts are inductively defined, C_1 is also an evaluation context. Lemma 3.13 then implies that M has a head redex.

Case 3 $L \subset \Delta_h$

In this case $M \equiv E[\tilde{M}] \xrightarrow{\Delta} E[\Delta_h]$, for some evaluation context E and term $\tilde{M}$ with $\Delta \subset \tilde{M}$. Distinguishing cases according to the form of Δ_h, we now show that $\tilde{M}$ always has an evaluation redex. For each case, we list the possible forms of $\tilde{M}$ and the corresponding evaluation redex. In the following, C_P stands for some context different from [ ]. The cases that refer to C_P make use of the fact that if P is a procedure and $C_P[\Delta] \xrightarrow{\Delta} P$ then $C_P[\Delta]$ is a procedure (Lemma 3.14).

Case 3.1 $\Delta_h \equiv (x.N_1)\ N_2$

$\tilde{M}$:	eval. redex:
$\Delta\ N_2$	Δ
$(x.C[\Delta])\ N_2$	$\tilde{M}$
$(x.N_1)\ (C[\Delta])$	$\tilde{M}$

Case 3.2 $\Delta_h \equiv f\ V$

$\tilde{M}$:	eval. redex:
$\Delta\ V$	Δ
$f\ \Delta$	Δ
$f\ (c^n\ M_1\ \ldots\ C[\Delta]\ \ldots\ M_n)$	$\tilde{M}$
$f\ (x.C[\Delta])$	$\tilde{M}$

Case 3.3 $\Delta_h \equiv N =: v\ ;\ v? \triangleright x.P$

$\tilde{M}$:	eval. redex:
$\Delta\ ;\ v? \triangleright x.P$	Δ
$C[\Delta] =: v\ ;\ v? \triangleright x.P$	$\tilde{M}$
$N =: \Delta\ ;\ v? \triangleright x.P$	Δ
$N =: v\ ;\ \Delta$	Δ
$N =: v\ ;\ \Delta \triangleright x.P$	Δ
$N =: v\ ;\ \Delta? \triangleright x.P$	Δ
$N =: v\ ;\ v? \triangleright x.\Delta$	Δ
$N =: v\ ;\ v? \triangleright x.C_P[\Delta]$	$\tilde{M}$

Case 3.4 $\Delta_h \equiv N =: v\ ;\ w? \triangleright x.P,\quad v \neq w$

$\tilde{M}$:	eval. redex:
$\Delta\ ;\ w? \triangleright x.P$	Δ
$C[\Delta] =: v\ ;\ w? \triangleright x.P$	$\tilde{M}$
$N =: \Delta\ ;\ w? \triangleright x.P$	Δ
$N =: v\ ;\ \Delta$	Δ
$N =: v\ ;\ \Delta \triangleright x.P$	Δ
$N =: v\ ;\ \Delta? \triangleright x.P$	Δ
$N =: v\ ;\ w? \triangleright x.\Delta$	Δ
$N =: v\ ;\ w? \triangleright x.C_P[\Delta]$	$\tilde{M}$

Case 3.5 $\Delta_h \equiv$ **var** $v.w? \triangleright x.P,\quad v \neq w$

$\tilde{M}$:	eval. redex:
var $v.\Delta$	Δ
var $v.\Delta \triangleright x.P$	Δ
var $v.\Delta? \triangleright x.P$	Δ
var $v.w? \triangleright x.\Delta$	Δ
var $v.w? \triangleright x.C_P[\Delta]$	$\tilde{M}$

Case 3.6 $\Delta_h \equiv (N_1 \triangleright x.N_2) \triangleright y.P$

$\tilde{M}$:	eval. redex:
$\Delta \triangleright y.P$	Δ
$(C[\Delta] \triangleright x.N_2) \triangleright y.P$	$\tilde{M}$
$(N_1 \triangleright \Delta) \triangleright y.P$	Δ
$(N_1 \triangleright x.C[\Delta]) \triangleright y.P$	$\tilde{M}$
$(N_1 \triangleright x.N_2) \triangleright y.\Delta$	Δ
$(N_1 \triangleright x.N_2) \triangleright y.C_P[\Delta]$	$\tilde{M}$

Case 3.7 $\Delta_h \equiv$ **return** $N \triangleright x.P$

$\tilde{M}$:	eval. redex:
$\Delta \triangleright x.P$	Δ
return $C[\Delta] \triangleright x.P$	$\tilde{M}$
(**return** $N) \triangleright x.\Delta$	Δ
(**return** $N) \triangleright C_P[\Delta]$	$\tilde{M}$

Case 3.8 $\Delta_h \equiv$ (**var** $v.N) \triangleright x.P$

$\tilde{M}$:	eval. redex:
$\Delta \triangleright x.P$	Δ
(**var** $v.C[\Delta]) \triangleright x.P$	$\tilde{M}$
(**var** $v.N) \triangleright x.\Delta$	Δ
(**var** $v.N) \triangleright x.C_P[\Delta]$	$\tilde{M}$

Case 3.9 $\Delta_h \equiv N_1 =: N_2 \triangleright x.P,\quad x \in fv\ P$

$\tilde{M}$:	eval. redex:
$\Delta \triangleright x.P$	Δ
$C[\Delta] =: N_2 \triangleright x.P$	$\tilde{M}$
$N_1 =: C[\Delta] \triangleright x.P$	$\tilde{M}$
$N_1 =: N_2 \triangleright x.\Delta$	Δ
$N_1 =: N_2 \triangleright x.C_P[\Delta]$	$\tilde{M}$

Case 3.10 $\Delta_h \equiv$ **pure** (S[**return** V])

An easy induction on the structure of state prefixes shows that, for any state prefix S, **pure** $S[\ ]$ and **pure** S[**return** [ ]] are evaluation contexts.

We distinguish further according to the relative positions of Δ and V.

Case 3.10.1 Δ and V disjoint

In this case $\tilde{M} \equiv$ **pure** ($S'[C[\Delta] \triangleright C'$[**return** V]]), for some state prefix S' and contexts C, C'. Furthermore, one of the following cases applies:

Case 3.10.1.1 $\tilde{M} \equiv$ **pure** $(S'[C[\Delta]{=:}v;\ C'[\textbf{return}\ V]])$

Then, since S is a state prefix, and since

$$S'[C[\Delta] =: v\ ;\ C'[\textbf{return}\ V]] \xrightarrow{\Delta} S'[\textbf{return}\ V],$$

$S'[C[\Delta] =: v\ ;\ C'[\,]]$ must also be a state prefix. Therefore, $\tilde{M}$ is a redex (and hence an evaluation redex of itself).

Case 3.10.1.2 $\tilde{M} \equiv$ **pure** $(S'[N =: \Delta\ ;\ C'[\textbf{return}\ V]])$

In this case, since **pure** S' is an evaluation context, so is **pure** $(S'[N =: [\,]\ ;\ C'[\textbf{return}\ V]])$. Hence, Δ is an evaluation redex of $\tilde{M}$.

Case 3.10.1.3 $\tilde{M} \equiv$ **pure** $(S'[\Delta \triangleright x.C'[\textbf{return}\ V]])$

In this case **pure** $(S'[[\,] \triangleright x.C'[\textbf{return}\ V]])$ is an evaluation context, and Δ is an evaluation redex of $\tilde{M}$.

Case 3.10.2 $\Delta \subseteq V$

In this case, since **pure** $(S[\textbf{return}\ [\,]])$ is an evaluation context, one of the following applies:

$\tilde{M}$:	eval. redex:
pure $(S[\textbf{return}\ \Delta])$	Δ
pure $(S[\textbf{return}\ c^n\ \ldots\ C[\Delta]\ \ldots])$	$\tilde{M}$
pure $(S[\textbf{return}\ x.C[\Delta]])$	$\tilde{M}$

Case 3.10.3 $V \subset \Delta$

In this case, since **pure** S' is an evaluation context, $\tilde{M}$ must have the form **pure** $(S'[\Delta])$, so Δ is an evaluation redex in $\tilde{M}$.

This concludes our case analysis. In every case, $\tilde{M}$ has an evaluation redex. Therefore,

$$M \equiv E[\tilde{M}] \equiv E[E'[\Delta']]$$

for some evaluation context E' and redex Δ'. Since E and E' are evaluation contexts, so is $E \cdot E'$. Hence, by Lemma 3.13, M contains a head redex. ■

Lemma 3.16 Let Δ_h be head redex and Δ_i be an internal redex in M. If $M \xrightarrow[\rho]{\Delta_i} N$ then Δ_h/ρ consists of one redex that is head redex in N.

Proof: We distinguish cases according to the relative positions of Δ_h and Δ_i.

Case 1 Δ_i and Δ_h are disjoint.

In this case the claim follows by a case analysis identical to the one in Lemma 3.15, *Case* 1.

Case 2 $\Delta_h \subseteq \Delta_i$

This case is not possible.

Case 3 $\Delta_i \subset \Delta_h$

In this case Δ_h/ρ is a redex, as is verified by inspecting the relevant clauses of the proof of Lemma 3.7. Furthermore, since $M \equiv E[\Delta_h] \xrightarrow[\rho]{} E[\Delta_h/\rho]$ for some evaluation context E, Δ_h/ρ is an evaluation redex of N. It remains to shown that it is a head redex. Assume that it is not; Then Δ_h/ρ is either properly contained in another evaluation redex Δ, or there is another evaluation redex Δ disjoint from, and to the left of, Δ_h.

In the first case there are evaluation contexts E_1, E_2, with $E_2 \not\equiv [\,]$ such that

$$N \equiv E_1[\Delta] \equiv E_1[E_2[\Delta_h/\rho]].$$

Since $\Delta_i \subset \Delta_h$, there is a context $C \not\equiv [\,]$ such that

$$M \equiv E_1[E_2[\Delta_h]] \equiv E_1[E_2[C[\Delta_i]]]$$

But then Lemma 3.17 implies that $E_2[\Delta_h]$ is a redex, contradicting the assumption that Δ_h is head redex in M.

In the second case, one can find evaluation contexts E, E', E'', bound variable x, tag v such that N is one of the following:

(1) $E[(E'[\Delta])\ (E''[\Delta_h/\rho])]$
(2) $E[E'[\Delta] =: E''[\Delta_h/\rho]]$
(3) $E[E'[\Delta] =: v \triangleright x.E''[\Delta_h/\rho]]$

Hence, M is one of the following:

(1) $E[(E'[\Delta])\ (E''[\Delta_h])]$
(2) $E[E'[\Delta] =: E''[\Delta_h]]$
(3) $E[E'[\Delta] =: v \triangleright x.E''[\Delta_h]]$

In each case, Δ an evaluation redex to the left of Δ_h. This contradicts the assumption that Δ_h is head redex. ■

Lemma 3.17 For any context $C \not\equiv [\,]$, evaluation context $E \not\equiv [\,]$, term M, and redex Δ, if $E[C[M]]$ is a redex and $C[\Delta]$ is a redex then $E[C[\Delta]]$ is also a redex.

Proof: If, in the left-hand-side of the reduction rule according to which $E[C[M]$ is a redex, M is contained in a meta-variable of $[E[C[M]]$, then clearly $E[C[N]]$

is a redex for any term N. A careful inspection of the reduction rules along with the definition of evaluation contexts E reveals that this is indeed the only way in which the hypotheses of the lemma can be satisfied.

■

Lemma 3.18 For any term M, internal redexes Δ_i, Δ_i' in M: If $M \xrightarrow[\rho]{\Delta_i} N$ then all elements of Δ_i/ρ are internal redexes in N.

Proof: Assume the contrary. Then there is a redex $\Delta \in \Delta_i/\rho$ that is head redex in N. By Lemma 3.15, M has a head redex, say Δ_h. By Lemma 3.16 and Lemma 3.13, Δ is the residual of Δ_h. This contradicts the assumption that $\Delta \in \Delta_i/\rho$. ■

Lemma 3.19 If $M \twoheadrightarrow N$, then there exist terms M_k, M_k' such that $M \xrightarrow{\Delta_{h,1}} M_1 \xrightarrow{\Delta_{h,2}} \dots \xrightarrow{\Delta_{h,m}} M_0' \xrightarrow{\Delta_{i,1}} M_1' \xrightarrow{\Delta_{i,2}} \dots \xrightarrow{\Delta_{i,n}} N$, where each $\Delta_{h,k}$ is head redex and each $\Delta_{i,k}$ is an internal redex.

Proof: Exactly as in [1], Lemma 11.4.4 – 11.4.6. One needs FD!, Lemma 3.15, Lemma 3.16, and Lemma 3.18. ■

We are now ready to prove the standardisation theorem:

Theorem 3.20 (Correspondence) For every closed term M and answer A,

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

Proof: By Lemma 3.19 $M \twoheadrightarrow M'$ using only head reductions, and $M' \twoheadrightarrow A$ using only internal reductions. Since there is no internal reduction $M \rightarrow A$, for any term M, we have $M \twoheadrightarrow M'$ using only head reductions. This implies $eval_{var}\, M \equiv A$. ■

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 6. The new basic constant **undef** is used to flag an unitialized variable. The rules in Figure 6 define a reduction relation λ_σ between terms in Λ_σ. This relation can be shown to be confluent.

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* [26].

4.1 Church-Rosser

We now embark on the proof that λ_σ is confluent. The proof follows the same outline as the corresponding proof for λ_{var} in Section 3, but differs in detail since we are dealing with a different calculus.

We first prove that the reduction $\xrightarrow[\beta_0\delta 0!]{}$ is strongly normalizing for λ_σ. We define terms with marked $\beta\delta$-redexes Λ_σ' and weighted terms Λ_σ^* just as we did for Λ_{var}. The definition of the norm $||\cdot||$ is just as for Λ_{var} with the addition of the rules

$$\begin{aligned} ||\sigma \cdot M|| &= Eexp(\#M||\sigma|| + ||M||) \\ ||\{\}|| &= 0 \\ ||\sigma \cup \{v : N\}|| &= ||\sigma|| + ||N|| \end{aligned}$$

and the revised rule

$$||\mathbf{pure}\, M|| = 1 + ||M||.$$

We note that every term in Λ_σ' can be completed to a term in Λ_σ^* with a decreasing weighting by working outward as before.

Lemma 4.1 For $M, N \in \Lambda_\sigma^*$, if M has a decreasing weighting, and $M \xrightarrow[!\beta_0\delta_0]{} N$ then $||M|| > ||N||$.

$$
\begin{array}{llcll}
\sigma_{var} & \sigma \cdot \textbf{var}\ v.P & \rightarrow & \sigma \cup \{v{:}\textbf{undef}\} \cdot P & \\
\sigma_{=:} & \sigma \cup \{v{:}M'\} \cdot M =: v\ ;P & \rightarrow & \sigma \cup \{v{:}M\} \cdot P & \\
\sigma_{?} & \sigma \cup \{v{:}M\} \cdot v? \rhd x.P & \rightarrow & \sigma \cup \{v{:}M\} \cdot (x.P)\ M & (M \not\equiv \textbf{undef}). \\
 & & & & \\
\sigma_{pure} & \textbf{pure}\ M & \rightarrow & \{\} \cdot M & \\
\sigma_{pc} & \sigma \cdot \textbf{return}\ c^n\ M_1\ ...\ M_k & \rightarrow & c^n\ (\sigma \cdot \textbf{return}\ M_1\ ...\ (\sigma \cdot \textbf{return}\ M_k) & (k \leq n) \\
\sigma_{p_\lambda} & \sigma \cdot \textbf{return}\ x.M & \rightarrow & x\ .\ (\sigma \cdot \textbf{return}\ M) & \\
\sigma_{p_f} & \sigma \cdot \textbf{return}\ [f] & \rightarrow & f &
\end{array}
$$

Figure 6: Modified reduction rules for λ_σ.

Proof: We proceed by case analysis on the form of single reduction steps.

Case $\beta_0, \delta_0, \rhd\!\rhd, r\rhd, v\rhd, =:\rhd$

In each of these cases, the corresponding case in the proof of Lemma 3.2 is independent of the presence of additional syntactic constructs in Λ_σ; hence each is a valid proof for the case in the present lemma.

Case σ_{var}

$$
\begin{array}{ll}
 & \|\sigma \cdot \textbf{var}\ v.M\| \\
= & Eexp((\#\textbf{var}\ v.M)\|\sigma\| + \|\textbf{var}\ v.M\|) \\
= & Eexp(\#M\|\sigma\| + 1 + \#M + \|M\|) \\
= & Eexp(1 + \#M(\|\sigma\| + 1) + \|M\|) \\
> & Eexp(\#M(\|\sigma\| + 1) + \|M\|) \\
= & Eexp(\#M\|\sigma \cup \{v : \textbf{undef}\}\| + \|M\|) \\
= & \|\sigma \cup \{v : \textbf{undef}\}\|
\end{array}
$$

Case $\sigma_{=:}$

$$
\begin{array}{ll}
 & \|\sigma \cup \{v : N'\} \cdot N =: v\ ;\ M\| \\
= & Eexp((\#N =: v\ ;\ M)\|\sigma \cup \{v : N'\}\| \\
 & \quad + \|N =: v\ ;\ M\|) \\
= & Eexp((1 + \#M)(\|\sigma\| + \|N'\|) \\
 & \quad + (1 + \#M)\|N =: v\| + \|M\|) \\
= & Eexp(\|\sigma\| + \|N'\| + \#M\|\sigma\| + \#M\|N'\| \\
 & \quad + (1 + \#M)(\|N\| + 1) + \|M\|) \\
> & Eexp(\#M\|\sigma\| + \#M\|N\| + \|M\|) \\
= & \|\sigma \cup \{v : N\} \cdot M\|
\end{array}
$$

Case $\sigma_{?}$

$$
\begin{array}{ll}
 & \|\sigma \cup \{v : N\} \cdot v?\ ;\ x.M\| \\
= & Eexp((\#v?\ ;\ x.M)\|\sigma \cup \{v : N\}\| \\
 & \quad + \|v?\ ;\ x.M\|) \\
= & Eexp((1 + \#M)(\|\sigma\| + \|N\|) \\
 & \quad + (1 + \#M)\|v?\| + \|x.M\|) \\
= & Eexp((1 + \#M)(\|\sigma\| + \|N\|) \\
 & \quad + 1 + \#M + \|x.M\|) \\
= & Eexp(\|\sigma \cup \{v : N\}\| + \#M\|\sigma\| \\
 & \quad + \#M\|N\| + 1 + \#M + \|x.M\|) \\
> & Eexp(\|\sigma \cup \{v : N\}\| + 1 + \|M\| + \|N\|) \\
= & Eexp((\#(x.M)\ N)\|\sigma \cup \{v : N\}\| \\
 & \quad + \|(x.M)\ N\|) \\
= & \|\sigma \cup \{v : N\} \cdot (x.M)\ N\|
\end{array}
$$

Case σ_{pure}

$$
\begin{array}{ll}
 & \|\textbf{pure}\ M\| \\
= & 1 + \|M\| \\
> & \|M\| \\
= & \|\emptyset \cdot M\|
\end{array}
$$

Case σ_{pc}

$$
\begin{array}{ll}
 & \|\sigma \cdot \textbf{return}\ (c^n\ M_1 \ldots M_k)\| \\
= & Eexp(\#\textbf{return}\ (c^n\ M_1 \ldots M_k))\|\sigma\| \\
 & \quad + \|\textbf{return}\ (c^n\ M_1 \ldots M_k)\|) \\
= & Eexp(\|\sigma\| + \|c^n\ M_1 \ldots M_k\|) \\
= & Eexp(\|\sigma\| + 1 + \|M_1\| \ldots \|M_k\|) \\
> & 1 + Eexp(\|\sigma\| + \|M_1\|) + \ldots \\
 & + Eexp(\|\sigma\| + \|M_k\|) \\
= & \|c^n\| + \|\sigma \cdot \textbf{return}\ M_1\| + \ldots \\
 & + \|\sigma \cdot \textbf{return}\ M_k\| \\
= & \|c^n(\sigma \cdot \textbf{return}\ M_1) \ldots (\sigma \cdot \textbf{return}\ M_k)\|
\end{array}
$$

Case σ_{p_λ}

$$
\begin{aligned}
& \|\sigma \cdot \mathbf{return}\, x.M\| \\
= \;& \mathit{Eexp}((\#\mathbf{return}\, x.M)\|\sigma\| + \|\mathbf{return}\, x.M\|) \\
= \;& \mathit{Eexp}(\|\sigma\| + 1 + \|M\|) \\
> \;& 1 + \mathit{Eexp}(\|\sigma\| + \|M\|) \\
= \;& \|x.\sigma \cdot \mathbf{return}\, M\|
\end{aligned}
$$

Case σ_{p_f}

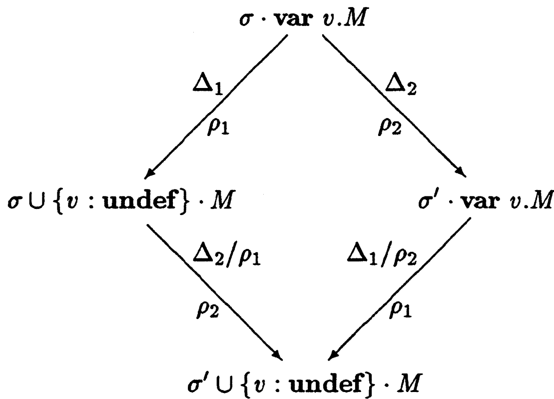
$$
\begin{aligned}
& \|\sigma \cdot \mathbf{return}\, f^n\| \\
= \;& \mathit{Eexp}(\|\sigma\| + \|f^n\|) \\
= \;& \mathit{Eexp}(\|\sigma\| + n) \\
> \;& n \\
= \;& \|f^n\|
\end{aligned}
$$

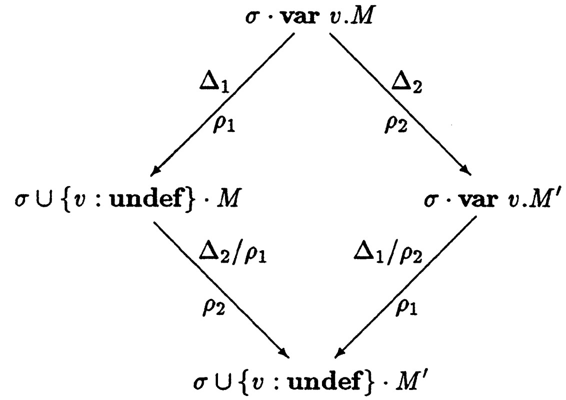
■

Lemma 4.2 The reduction relation $\overset{?}{\to}_\sigma$ is weakly Church-Rosser.

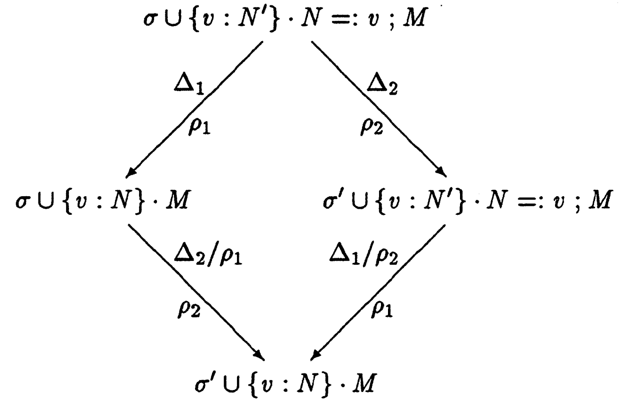
Proof: Assume $M \xrightarrow[\rho_1]{\Delta_1} M_1$ and $M \xrightarrow[\rho_2]{\Delta_2} M_2$. We distinguish cases according to the relative positions of Δ_1 and Δ_2. If $\Delta_1 \equiv \Delta_2$ then $M_1 \equiv M_2 \equiv M_3$, since the left-hand sides of the rules in Figure 6 do not overlap. If Δ_1 and Δ_2 are disjoint, then the desired property follows again from the lack of overlaps in the rules. We consider now the cases in which $\Delta_2 \subset \Delta_1$ by enumerating the possible forms of Δ_1. Wherever the redex Δ_2 occurs in a store σ, we write the reduction as $\sigma \xrightarrow{\Delta_2} \sigma'$ in the remainder of the proof.

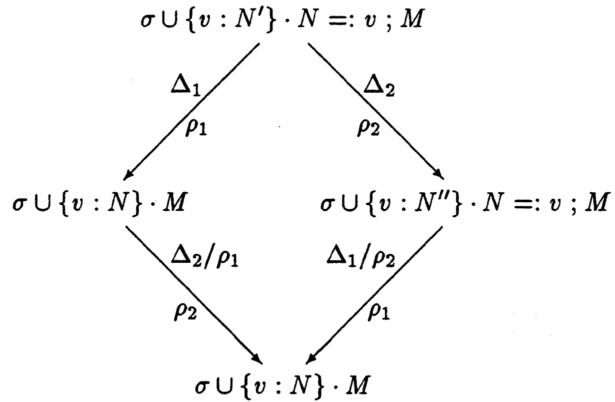
Case 1 (σ_{var}) $\Delta_1 \equiv \sigma \cdot \mathbf{var}\ v.M$ In this case, the possibilities for Δ_2 are $\sigma \xrightarrow{\Delta_2} \sigma'$ or $M \xrightarrow{\Delta_2} M'$. The following two diagrams show how the weak Church-Rosser property can be established in this case:

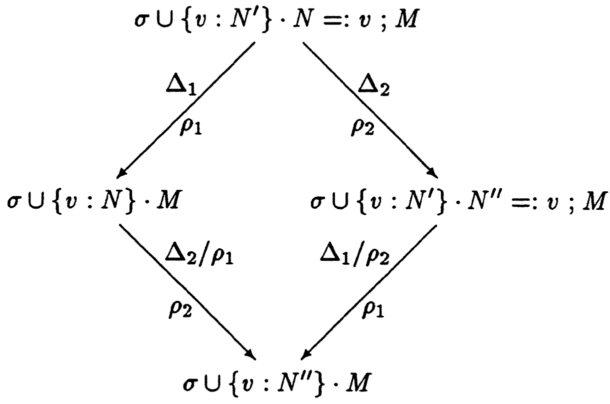


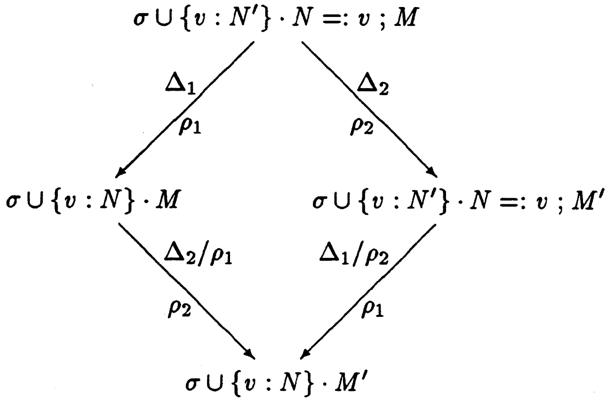


Case 2 ($\sigma_{=:}$) $\Delta_1 \equiv \sigma \cup \{v : N'\} \cdot N =: v\ ;\ M$ In this case Δ_2 can occur in σ, N', N, or M:

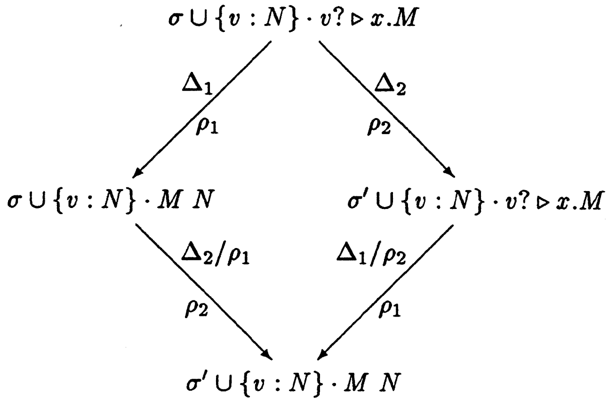


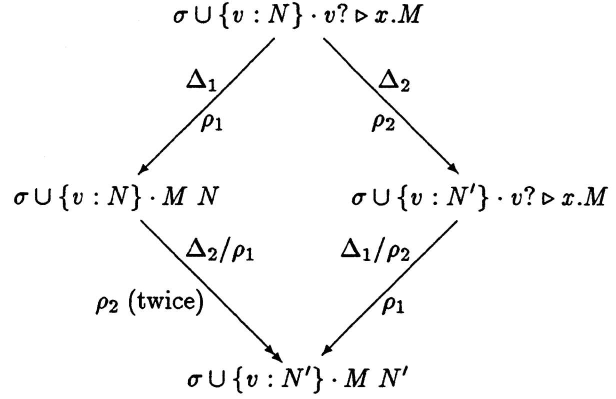


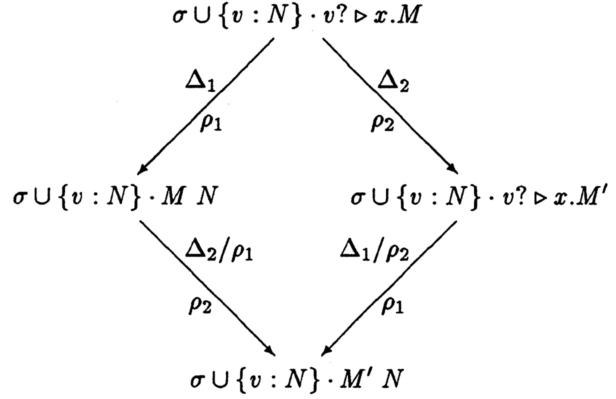




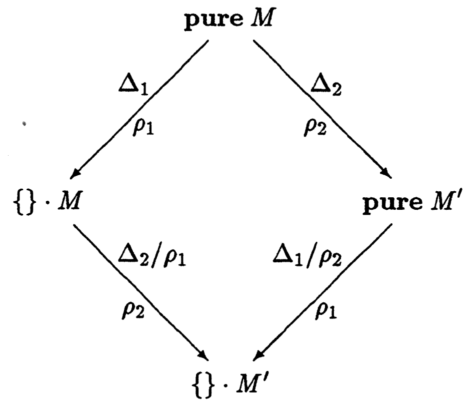
Case 3 ($\sigma_?$) $\Delta_1 \equiv \sigma \cup \{v : N\} \cdot v? \triangleright x.M$ In this case, Δ_2 can be in σ, N, or M:



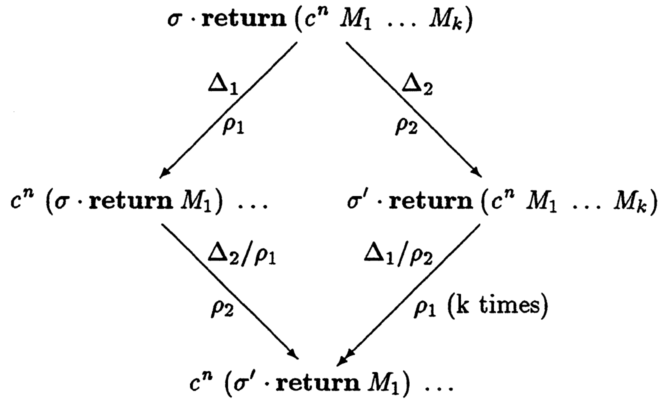


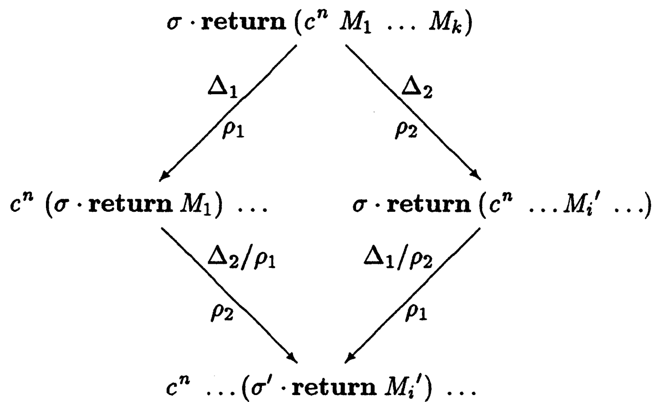


Case 4 (σ_{pure}) $\Delta_1 \equiv$ **pure** M

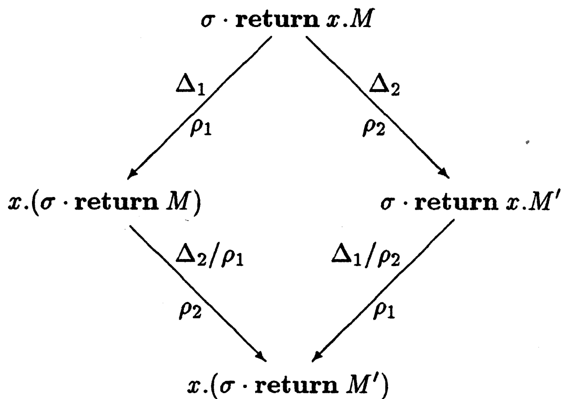


Case 5 (σ_{pc}) $\Delta_1 \equiv \sigma \cdot$ **return** $(c^n \; M_1 \; \ldots \; M_k)$

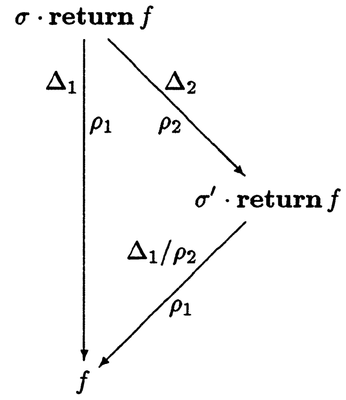




Case 6 (σ_{p_λ}) $\Delta_1 \equiv \sigma \cdot \textbf{return}\, x.M$



Case 7 (σ_{p_f}) $\Delta_1 \equiv \sigma \cdot \textbf{return}\, f$



■

Lemma 4.3 The reduction relations $\twoheadrightarrow_{\sigma!}$ and $\twoheadrightarrow_{\beta\delta}$ commute.

Proof: The proof is a case analysis directly analogous to that in the proof of Lemma 3.8. ■

Theorem 4.4 $\rightarrow_\sigma$ is Church-Rosser.

This follows from Lemma 4.2 and Lemma 4.3 by the reasoning used in the proof of Theorem 3.10.

4.2 Evaluation

We now proceed to prove that λ_σ possesses a standard order of evaluation. Our development here parallels the proof of Theorem 3.20.

The evaluation contexts for λ_σ are given by the following grammar:

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

Based on this definition of evaluation context, we define the notion of head redex and the standard evaluation function $eval_\sigma$ for programs in λ_σ as was done for λ_{var} in Section 3.4. $eval_\sigma$ closely corresponds to the usual notion of store-based computations with store access and update as single reduction steps. $eval_\sigma$ is a semi-decision procedure for equations between terms and answers in Λ_σ.

Lemma 4.5 If $M \rightarrow_\sigma N$ and N has a head redex, then so has M.

Proof: We proceed in the same manner as for Lemma 3.15.

Case 1 L and Δ_h are disjoint

In this case, M must have one of the following forms:

(1) $E[(E'[\Delta_h])\ C[\Delta]]$
(2) $E[C[\Delta] =: E'[\Delta_h]]$
(3) $E[E'[\Delta_h] \triangleright x.C[\Delta]]$
(4) $E[C[\Delta] \triangleright x.E'[\Delta_h]]$
where C is not an evaluation context
(5) $E[\sigma \cup \{v : C[\Delta]\} \cdot E'[\Delta_h]]$
(6) $E[\sigma \cup \{v : C[\Delta]\} \cdot \mathbf{return}\ E'[\Delta_h]]$

In every case, Δ_h is an evaluation redex of M, and therefore (by the observation in Lemma 3.13), M has a head redex.

Case 2 $\Delta_h \subseteq L$

The reasoning for Lemma 3.15, Case 2, applies here without change.

Case 3 $L \subset \Delta_h$ In this case, $M \equiv \tilde{C}[\tilde{M}] \xrightarrow{\Delta} \tilde{C}[\Delta_h]$, for some term $\tilde{M}$ with $\Delta \subset \tilde{M}$. Distinguishing cases according to the form of Δ_h, we now show that $\tilde{M}$ always has an evaluation redex. For each case, we list the possible forms of $\tilde{M}$ and the corresponding left redex:

Case 3.1 $\Delta_h \equiv (x.M)\ N$

The reasoning for Lemma 3.15, Case 3.1, applies here without change.

Case 3.2 $\Delta_h \equiv f\ V$

The reasoning for Lemma 3.15, Case 3.2, applies here without change.

Case 3.3 $\Delta_h \equiv \sigma \cdot \mathbf{var}\ v.P$

$\tilde{M}$:	eval. redex:
$\sigma' \cup \{v' : C[\Delta]\} \cdot \mathbf{var}\ v.P$	$\tilde{M}$
$\sigma \cdot \Delta$	Δ
$\sigma \cdot \mathbf{var}\ v.C[\Delta]$	$\tilde{M}$

Case 3.4 $\Delta_h \equiv \sigma \cup \{v : N'\} \cdot N =: v \triangleright x.P$

$\tilde{M}$:	eval. redex:
$\sigma \cup \{v : N', v' : C[\Delta]\} \cdot N =: v \triangleright x.P$	$\tilde{M}$
$\sigma \cup \{v : C[\Delta]\} \cdot N =: v \triangleright x.P$	$\tilde{M}$
$\sigma \cup \{v : N'\} \cdot \Delta$	Δ
$\sigma \cup \{v : N'\} \cdot \Delta \triangleright x.P$	Δ
$\sigma \cup \{v : N'\} \cdot C[\Delta] =: v \triangleright x.P$	$\tilde{M}$
$\sigma \cup \{v : N'\} \cdot N =: \Delta \triangleright x.P$	Δ
$\sigma \cup \{v : N'\} \cdot N =: v \triangleright x.C[\Delta]$	$\tilde{M}$

Case 3.5 $\Delta_h \equiv \sigma \cup \{v : N\} \cdot v? \triangleright x.P$

$\tilde{M}$:	eval. redex:
$\sigma \cup \{v : N, v' : C[\Delta]\} \cdot v? \triangleright x.P$	$\tilde{M}$
$\sigma \cup \{v : C[\Delta]\} \cdot v? \triangleright x.P$	$\tilde{M}$
$\sigma \cup \{v : N\} \cdot \Delta$	Δ
$\sigma \cup \{v : N\} \cdot \Delta \triangleright x.P$	Δ
$\sigma \cup \{v : N\} \cdot \Delta? \triangleright x.P$	Δ
$\sigma \cup \{v : N\} \cdot v? \triangleright x.C[\Delta]$	$\tilde{M}$

Case 3.6 $\Delta_h \equiv \mathbf{pure}\ M$

$\tilde{M}$:	eval. redex:
$\mathbf{pure}\ C[\Delta]$	$\tilde{M}$

Case 3.7 $\Delta_h \equiv \sigma \cdot \mathbf{return}\ (c^n M_1\ \ldots\ M_k)$

$\tilde{M}$:	eval. redex:
$\sigma \cup \{v : C[\Delta]\} \cdot \mathbf{return}\ (c^n M_1\ \ldots\ M_k)$	$\tilde{M}$
$\sigma \cdot \Delta$	Δ
$\sigma \cdot \mathbf{return}\ \Delta$	Δ
$\sigma \cdot \mathbf{return}\ (\Delta M_j\ \ldots\ M_k)$ & $(0 \leq j \leq k)$	Δ
$\sigma \cdot \mathbf{return}\ (c^n\ \ldots\ C[\Delta]\ \ldots)$	$\tilde{M}$

Case 3.8 $\Delta_h \equiv \sigma \cdot \mathbf{return}\ x.M$

$\tilde{M}$:	eval. redex:
$\sigma \cup \{v : C[\Delta]\} \cdot \mathbf{return}\ x.M$	$\tilde{M}$
$\sigma \cdot \Delta$	Δ
$\sigma \cdot \mathbf{return}\ \Delta$	Δ
$\sigma \cdot \mathbf{return}\ x.C[\Delta]$	$\tilde{M}$

Case 3.9 $\Delta_h \equiv \sigma \cdot \textbf{return}\, f$

$\tilde{M}$:	eval. redex:
$\sigma \cup \{v : C[\Delta]\} \cdot \textbf{return}\, f$	$\tilde{M}$
$\sigma \cdot \Delta$	Δ
$\sigma \cdot \textbf{return}\, \Delta$	Δ

Case 3.10 $\Delta_h \equiv (P \triangleright x.Q) \triangleright y.R$

Case 3.11 $\Delta_h \equiv \textbf{return}\, P \triangleright x.Q$

Case 3.12 $\Delta_h \equiv (\textbf{var}\ v.P) \triangleright x.Q$

The reasoning for Lemma 3.15, Cases 3.6 to 3.8, applies here without change.

■

Lemma 4.6 Let Δ_h be the head redex and Δ_i be some interior redex in M. If $M \xrightarrow[\rho]{\Delta_i} N$ then Δ_h/ρ consists of one redex that is head redex in N.

Proof: The proof is entirely analogous to the proof of Lemma 3.16. ■

Lemma 4.7 For any term M, internal redexes Δ_i, Δ_i' in M: If $M \xrightarrow[\rho]{\Delta_i} N$ then all elements of Δ_i/ρ are internal redexes in N.

Proof: As for Lemma 3.18. ■

Theorem 4.8 (Correspondence) For every closed term M and answer A,

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

Proof: As for Theorem 3.20. ■

4.3 Simulation of λ_{var}-evaluation by λ_σ-evaluation

Since every term in Λ_{var} is also a member of the language Λ_σ, it makes sense to apply λ_σ-standard reduction to λ_{var} terms. We now show that both reduction relations are equivalent if we consider only observable results. This equivalence can be interpreted as establishing that $eval_\sigma$ is a correct implementation of $eval_{var}$. Combined with our informal understanding that $eval_\sigma$ can be implemented by an abstract machine having a store, we will thus be reasonably convinced that λ_{var}-evaluation can also be implemented by such a machine.

We will first prove some auxiliary lemmas that establish that λ_{var}-reduction preserves certain features of state prefixes. We will then introduce a precise notion of correspondence between state prefixes in λ_{var} and explicit stores in λ_σ. After proving some properties of the correspondence relation we will prove this section's main theorem that the two calculi simulate one another.

Define a *bubble context* to be one generated by the grammar $B ::= [\,] \ \mid\ v? \triangleright x.B$.

Lemma 4.9 Assume $\lambda_{var} \vdash M \twoheadrightarrow M'$.
(*i*) If $M \equiv \textbf{var}\ v.N$ then there is a term $N' \in \Lambda_{var}$ and a bubble context $B[\,]$ such that $M' \equiv B[\textbf{var}\ v.N']$.
(*ii*) If $M \equiv N =: v\ ;\ P$ then there are terms N', $P' \in \Lambda_{var}$ and a bubble context $B[\,]$ such that $M' \equiv B[N' =: v\ ;\ P']$.
(*iii*) If $M \equiv \textbf{return}\ N$ then there is a term $N' \in \Lambda_{var}$ such that $M' \equiv \textbf{return}\ N'$.
(*iv*) If $M \equiv v? \triangleright x.P$ then there is a term $P' \in \Lambda_{var}$ such that $M' \equiv v? \triangleright x.P'$

Proof: By inspection of the reduction rules for λ_{var}, it is clear that only the **pure**-rules (p_c), (p_λ), and (p_f) can remove a **var** or =: from a term. Furthermore, there is no rule that can reduce a term with one of these constructs as a prefix to one that has them only within the context of a **pure**. The only remaining reduction rules involving terms of the forms dealt with in (*i*) and (*ii*) are the bubble rules (b_1) and (b_2), which clearly act to embed the former top-level construct within a bubble context. Similar reasoning establishes (*iii*) and (*iv*). ■

Lemma 4.10 Let $M \in \Lambda_{var}$ and assume that $M \equiv E[\Delta_h] \twoheadrightarrow A$, where Δ_h is a (f), (b_1), or (b_2) redex that is head redex in M. Then there is an evaluation context E' and a pre-state prefix R' such that $M \equiv E'[\textbf{pure}\ (R'[\Delta_h])]$.

Proof: Assume the contrary, that is, that M is not of the desired form. We then consider the remaining possible structures of M and show that none of them can reduce to an answer.

If $E \equiv R'$ for some pre-state prefix R', then Lemma 4.9 applies to show that $M \not\twoheadrightarrow A$, and we have a contradiction. Hence $E \equiv E_1 \cdot E_2 \cdot R'$, where E_1 is an evaluation context, E_2 is a non-empty evaluation context that is not a pre-state prefix, and R' is a pre-state prefix. Without loss of generality assume that $E_1 \equiv [\,]$ and that E_2 is minimal, that is, E_2 can be obtained by exactly one of the productions in Figure 5. We distinguish cases according to the form of E_2:

$\boxed{Case\ E_2 \equiv [\,]\ N}$

Assume $R'[\Delta_h]\ N \twoheadrightarrow A$. By Theorem 4.8, we can choose the reduction to reduce Δ_h first. By Lemma 4.9, the reduction can only yield a term of the form $B[R''[N]]$. Such a residual term can only interact with its context R' by means of the bubble-rules (b_1) and (b_2); thus, there is no way to reduce $R'[\Delta_h]$ to an abstraction or primitive-function name, and hence no way to reduce the application to an answer. This is a contradiction.

$\boxed{Case\ E_2 \equiv f\ [\,]}$

We argue similarly to the previous case, noting instead that $R'[\Delta_h]$ can reduce neither to an abstraction nor to a constructed value.

$\boxed{Case\ E_2 \equiv [\,]?}$

$\boxed{Case\ E_2 \equiv N =: [\,]}$

$\boxed{Case\ E_2 \equiv \mathbf{pure}\ (S[\mathbf{return}\ [\,]])}$

These cases are all similar to the first two cases.

$\boxed{Case\ E_2 \equiv [\,] \triangleright x.N}$

In this case $E_2[R'[\Delta_h]]$ is a redex, contradicting the assumption that Δ_h is head redex.

$\boxed{Case\ E_2 \equiv N \triangleright x.[\,]}$

For this case to apply, N cannot have the form $E'[\Delta]$, else Δ_h could not be the head redex. Also, $N \triangleright x.\Delta_h$ itself cannot be a redex. Thus N cannot have any of the forms $N_1 \triangleright y.N_2$, $\mathbf{return}\ N_1$, $\mathbf{var}\ v.N_1$, or $N_1 =: N_2$. Now by Lemma 4.9, $\Delta_h \twoheadrightarrow B[R''[N_1]]$, so a head-reduction sequence will bubble the tag-lookups in B to the left, where they cannot interact with any of the remaining possible forms for N, hence the entire term cannot reduce to an answer.

$\boxed{Case\ E_2 \equiv \mathbf{var}\ v.[\,]}$

$\boxed{Case\ E_2 \equiv N =: v \triangleright x.[\,]}$

$\boxed{Case\ E_2 \equiv \mathbf{return}\ [\,]}$

These cases all contradict the assumption that E_2 is not a pre-state prefix.

The only remaining case is $E_2 \equiv \mathbf{pure}\ [\,]$, hence we must have

$$M \equiv E_1[E_2[R'[\Delta_h]]] \equiv E_1[\mathbf{pure}\ (R'[\Delta_h])].$$

■

Lemma 4.11 Let $M \in \Lambda_{var}$ with $M \twoheadrightarrow A$ and $M \equiv E[\mathbf{pure}\ (R[\Delta_h])]$, for some evaluation context E, pre-state prefix R, and head redex Δ_h. Let Δ_h have one of the forms

$$\Delta_h \equiv N =: v\ ;\ P$$
$$\Delta_h \equiv v? \triangleright x.P.$$

Then $v \in bt\ R$.

Proof: We only spell out the proof for the case $\Delta_h \equiv N =: v\ ;\ P$, the other case is shown by an analogous argument.

Assume the contrary, i.e. $v \notin bt\ R$. Take any term M' such that $M \twoheadrightarrow M'$. We show that M' is not an answer.

Assume that M' is an answer. By Theorem 3.20, $M \underset{h}{\twoheadrightarrow} A$, using only head reductions $\underset{h}{\rightarrow}$. Assume that $M \underset{h}{\rightarrow} M_1$ is the first step of this reduction sequence. We have $M \equiv E[\mathbf{pure}\ (R[N =: v\ ;\ P])]$. By Lemma 4.9, $M_1 \equiv E[\mathbf{pure}\ (R[B[N' =: v\ ;\ P']])]$, for some terms N', P' and bubble context B. If B is nonempty, beginning with $v'?$ for some tag v', then $v' \in bt\ R$ by the second part of the lemma. Otherwise, since by assumption $v \notin bt\ R$, we have that $R[N' =: v\ ;\ [\,]]$ is not part of a state prefix, which implies that $\mathbf{pure}\ (R[N' =: v\ ;\ P'])$ is not a redex. Therefore, the head redex of M_1, if it exists, is contained in $N' =: v\ ;\ P'$. By induction on the length of the head-reduction sequence we can thus show that $M' \equiv E[\mathbf{pure}\ (R[N'' =: v\ ;\ P''])]$, for some terms N'' and P'', which contradicts the assumption that M' is an answer. ■

Lemma 4.12 Let $M \in \Lambda_{var}$ with $M \twoheadrightarrow A$ and $M \equiv E[\mathbf{pure}\ (R[N])]$, for some evaluation context E, some term N and some pre-state prefix R. If M has a head redex then R is a state prefix.

Proof: Essentially the same as the proof of Lemma 4.11. ■

Definition. We define a *correspondence* relation ($\hat{=}$) between terms and state-prefixes in Λ_{var} and terms and states in Λ_σ as follows:

(i) For any state prefix S, state σ, $S \hat{=} \sigma$ iff

$$\begin{aligned} &dom\ \sigma = bt\ S,\\ &S \equiv S'[N =: v\ ;\ R[\,]], v \notin wr\ R \Rightarrow\\ &\qquad \exists N'\ .\ (v : N') \in \sigma \wedge N \hat{=} N',\\ &v \in bt\ S, v \notin wr\ S \Rightarrow (v : \mathbf{undef}) \in \sigma \end{aligned}$$

(ii) On terms, $\doteq$ is axiomatized as follows:

$$
\begin{array}{lcl}
M \equiv M' & \Rightarrow & M \doteq M' \\
S \doteq \sigma, M \doteq M' & \Rightarrow & \mathbf{pure}(S[M]) \doteq \sigma \cdot M' \\
M \doteq M', N \doteq N' & \Rightarrow & M\,N \doteq M'\,N' \\
M \doteq M' & \Rightarrow & x.M \doteq x.M' \\
M \doteq M' & \Rightarrow & \mathbf{var}\ v.M \doteq \mathbf{var}\ v.M' \\
M \doteq M' & \Rightarrow & M? \doteq M'? \\
M \doteq M', N \doteq N' & \Rightarrow & M =: N \doteq M' =: N' \\
M \doteq M', N \doteq N' & \Rightarrow & M \rhd x.N \doteq M' \rhd x.N' \\
M \doteq M' & \Rightarrow & \mathbf{return}\ M \doteq \mathbf{return}\ M' \\
M \doteq M' & \Rightarrow & \mathbf{pure}\ M \doteq \mathbf{pure}\ M'
\end{array}
$$

Lemma 4.13 For any term $M \in \Lambda_\sigma$ there exists $var(M) \in \Lambda_{var}$ such that $var(M) \doteq M$. For any state $\sigma \in \Lambda_\sigma$ there exists a state prefix $S \equiv var(\sigma)$ in Λ_{var} such that $S \doteq \sigma$.

Proof: An easy construction by structural induction. The base cases for terms are obvious. Define $var(\{v_1 : M_1, \ \ldots \ v_n : M_n\}) \equiv \mathbf{var}\ v_1. \ \ldots \ \mathbf{var}\ v_n.var(M_1) =: v_1 \ ; \ \ldots \ var(M_n) =: v_n \ ; [\,]$. ■

Lemma 4.14 For any term $M \in \Lambda_{var}$ and state prefix S, there is a state σ such that $S \doteq \sigma$ and

$$\mathbf{pure}\ (S[M]) \underset{\sigma}{\twoheadrightarrow} \sigma \cdot N.$$

Proof: Applying a σ_{pure} reduction one gets $\mathbf{pure}\ (S[M]) \rightarrow \{\} \cdot M$. Then, repeatedly applying σ_{var} and $\sigma_{=:}$ reductions, one gets $\{\} \cdot M \twoheadrightarrow \sigma \cdot M$ for some state σ with the required properties. ■

Lemma 4.15 Assume $M \in \Lambda_{var}$, $M_\sigma, N_\sigma \in \Lambda_\sigma$ such that $M \doteq M_\sigma$ and $M_\sigma \underset{\sigma}{\rightarrow} N_\sigma$. Then there is a term $N \in \Lambda_{var}$ such that the following diagram commutes:

$$
\begin{array}{ccc}
M_\sigma & \xrightarrow{\ \sigma\ } & N_\sigma \\
\doteq \Big| & & \vdots \doteq \\
M & \overset{var}{\cdots\cdots\twoheadrightarrow} & N
\end{array}
$$

Proof: We distinguish cases according to the reduction rule by which $M_\sigma \underset{\sigma}{\rightarrow} N_\sigma$. Since ($\doteq$) distributes through all Λ_σ productions, we can assume without loss of generality that the redex in $M_\sigma \underset{\sigma}{\rightarrow} N_\sigma$ is the whole term M_σ. If M_σ is a (b_1), (b_2), ($\rhd\rhd$), ($r\rhd$), ($v\rhd$), or ($=:\rhd$) redex, the result follows immediately, since these are also reductions in λ_{var}. The remaining cases are as follows (we use Lemma 4.13 freely without mention):

Case $M_\sigma \equiv \sigma \ \cdot \mathbf{var}\ v.P_\sigma$

In this case $M \equiv \mathbf{pure}\ (S[\mathbf{var}\ v.P])$ for some state-prefix S and term P such that $S \doteq \sigma$ and $P \doteq P_\sigma$. The lemma then follows from the diagram:

$$
\begin{array}{ccc}
M_\sigma & \longrightarrow & \sigma \cup \{v : \mathbf{undef}\} \cdot P_\sigma \\
\doteq \Big| & \diagup \doteq & \\
M & &
\end{array}
$$

Case $M_\sigma \equiv \sigma \cup \{v : N'_\sigma\} \cdot N_\sigma =: v \ ; P_\sigma$

In this case $M \equiv \mathbf{pure}\ (S[N =: v \ ; P])$ for some state-prefix S and terms P, N such that $S \doteq \sigma \cup \{v : N'_\sigma\}$, $P \doteq P_\sigma$ and $N \doteq N_\sigma$. The lemma then follows from the diagram:

$$
\begin{array}{ccc}
M_\sigma & \longrightarrow & \sigma \cup \{v : N_\sigma\} \cdot P_\sigma \\
\doteq \Big| & \diagup \doteq & \\
M & &
\end{array}
$$

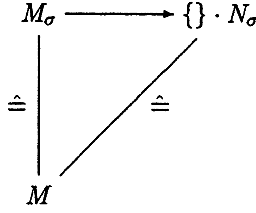
Case $M_\sigma \equiv \sigma \cup \{v : N_\sigma\} \cdot v? \rhd x.P_\sigma$

In this case $M \equiv \mathbf{pure}\ (S[v? \rhd x.P])$ for some state-prefix S and term P such that $S \doteq \sigma \cup \{v : N_\sigma\}$ and $P \doteq P_\sigma$. We have some choice in the ordering of the elements of S, so we choose to make the assignment to v be the innermost part of the state prefix. Since M_σ is a redex, $N_\sigma \not\equiv \mathbf{undef}$. Since $S \doteq \sigma \cup \{v : N_\sigma\}$, there is an assignment $N =: v$ in S such that $N \doteq N_\sigma$. The lemma then follows from the diagram:

$$
\begin{array}{ccc}
M_\sigma & \longrightarrow & \sigma \cup \{v : N_\sigma\} \cdot (x.P_\sigma)\ N_\sigma \\
\doteq \Big| & & \Big| \doteq \\
M & \underset{fb_1 b_2}{\twoheadrightarrow} & \mathbf{pure}\ (S[(x.P)\ N])
\end{array}
$$

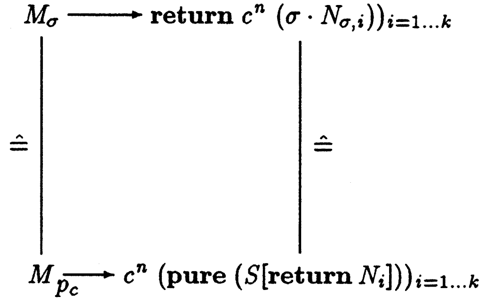
Case $M_\sigma \equiv \mathbf{pure}\ N_\sigma$

In this case $M \equiv \mathbf{pure}\ N$, for some term N such that $N \doteq N_\sigma$. The lemma then follows from the diagram:



$\boxed{\textit{Case } M_\sigma \equiv \sigma \cdot \textbf{return } c^n\ N_{\sigma,1}\ ...\ N_{\sigma,k}}$

In this case $M \equiv \textbf{pure}\ (S[\textbf{return } c^n\ N_1\ ...\ N_k])$ for some state-prefix S and term N_i such that $S \hat{=} \sigma$ and $N_i \hat{=} N_{\sigma,i}$ for $1 \leq i \leq k$. The lemma then follows from the diagram:



The cases where M_σ is a p_λ or p_f redex are shown analogously to the last case. ■

Theorem 4.16 (Simulation) For every closed term $M \in \Lambda_{var}$ and answer A,

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

Proof: We first prove "$\Rightarrow$". Assume $\lambda_{var} \vdash M = A$. By Theorem 3.20 $M \underset{h}{\twoheadrightarrow} A$ using only head reductions. We use an induction on the length of the head reduction sequence from M to A.

Base case. If $M \equiv A$, there is nothing to show.

Induction step. If $M \not\equiv A$, then we have (in λ_{var}):

$$M \equiv E[\Delta] \underset{h}{\rightarrow} E[L] \equiv M' \underset{h}{\twoheadrightarrow} A$$

for some evaluation context E, head redex Δ, terms L, M'. We show that $\lambda_\sigma \vdash M = M'$ by a case analysis on the form of Δ. If Δ is a β, δ, $\triangleright\!\triangleright$, $r\triangleright$ or $v\triangleright$ redex the result follows immediately since these are also reductions in λ_σ. The remaining cases are:

$\boxed{\textit{Case } \Delta \equiv N =: v\ ;\ v? \triangleright x.P}$

By Lemma 4.10 and Lemma 4.12 there is an evaluation context E and a state prefix S such that

$$M \equiv E[\textbf{pure}\ (S[N =: v\ ;\ v? \triangleright x.P])].$$

By Lemma 4.14 $M \underset{\sigma}{\twoheadrightarrow} M_\sigma$, where

$$M_\sigma \equiv E[\sigma_S \cdot N =: v\ ;\ v? \triangleright x.P]$$

and σ_S is as in Lemma 4.14. By Lemma 4.11, $v \in bt\ S$, hence $v \in dom\ \sigma_S$. Assume, without loss of generality, that $\sigma_S \equiv \sigma' \cup \{v : N'\}$. Then,

$$\begin{array}{rcl} M_\sigma & \equiv & E[\sigma' \cup \{v : N'\} \cdot N =: v\ ;\ v? \triangleright x.P] \\ & \underset{\sigma_{=:}}{\longrightarrow} & E[\sigma' \cup \{v : N\} \cdot v? \triangleright x.P] \\ & \underset{\sigma_?}{\longrightarrow} & E[\sigma' \cup \{v : N\} \cdot (x.P)\ N] \\ & \underset{\sigma}{\twoheadleftarrow} & E[\textbf{pure}\ (S[N =: v\ ;\ (x.P)\ N])] \\ & \equiv & M' \end{array}$$

$\boxed{\textit{Case } \Delta \equiv N =: v\ ;\ w? \triangleright x.P\ ,\ v \not\equiv w}$

As before,

$$M \equiv E[\textbf{pure}\ (S[N =: v\ ;\ w? \triangleright x.P])] \underset{\sigma}{\twoheadrightarrow} M_\sigma$$

where

$$M_\sigma \equiv E[\sigma_S \cdot N =: v\ ;\ w? \triangleright x.P]$$

By Lemma 4.11, both v and w are in $bt\ S$ and hence in $dom\ \sigma_S$. Assume without loss of generality that $\sigma_S \equiv \sigma' \cup \{v : N'\}$. Then,

$$\begin{array}{rcl} M_\sigma & \equiv & E[\sigma' \cup v : N' \cdot N =: v\ ;\ w? \triangleright x.P] \\ & \rightarrow_{\sigma_{=:}} & E[\sigma \cup v : N \cdot w? \triangleright x.P] \\ & \rightarrow_{\sigma_?} & E[\sigma \cup v : N \cdot (x.P)\ (\sigma\ w)] \\ & \rightarrow_\beta & E[\sigma \cup v : N \cdot [(\sigma\ w)/x]\ P] \\ & \leftarrow_{\sigma_{=:}} & E[\sigma \cup v : N' \cdot N =: v\ ;\ [(\sigma\ w)/x]\ P], \\ & & (x \notin fv\ N) \\ & \leftarrow_\beta & E[\sigma \cup v : N' \cdot (x.\ N =: v\ ;\ P)\ (\sigma\ w)] \\ & \leftarrow_{\sigma_?} & E[\sigma \cup v : N' \cdot w? \triangleright N =: v\ ;\ x.P] \\ & \twoheadleftarrow_\sigma & E[\textbf{pure}\ (S[w? \triangleright x.N =: v\ ;\ P])] \\ & \equiv & M' \end{array}$$

$\boxed{\textit{Case } \Delta \equiv \textbf{var } v.w? \triangleright x.P\ ,\ v \not\equiv w}$

This case is proved similarly to the last case.

$\boxed{\textit{Case } \Delta \equiv \textbf{pure}\ (S[\textbf{return } x.M])}$.

Again using Lemma 4.14:

$$\begin{array}{rcl} M & \equiv & E[\textbf{pure}\ (S[\textbf{return } x.M])] \\ & \underset{\sigma}{\twoheadrightarrow} & E[\sigma_S \cdot \textbf{return } x.M] \\ & \underset{\sigma_{pc}}{\twoheadrightarrow} & E[x.(\sigma \cdot \textbf{return } M)] \\ & \underset{\sigma}{\twoheadleftarrow} & E[x.\textbf{pure}\ (S[\textbf{return } M])] \end{array}$$

$\boxed{Case\ \Delta \equiv \textbf{pure}\ (S[\textbf{return}\ c^n M_1 ... M_n])}$.

$\boxed{Case\ \Delta \equiv \textbf{pure}\ (S[\textbf{return}\ f])}$.

These cases are similar to the previous case.

This concludes the proof in the direction "$\Rightarrow$".

We now show "$\Leftarrow$". Assume $M \in \Lambda_{var}$ such that $\lambda_\sigma \vdash M = A$. By the definition of ($\hat{=}$), $M \hat{=} M$. Also by the definitions of $\hat{=}$, $A \hat{=} A' \Rightarrow A \equiv A'$. The proposition then follows by pasting together the diagrams provided by Lemma 4.15:

$$
\begin{array}{ccccc}
M & \xrightarrow[\sigma]{} & M'_1 \cdots\cdots & \xrightarrow[\sigma]{} & \cdots A \\
\hat{=} \Big| & & \hat{=} \Big| & & \Big| \hat{=} \\
M & \xrightarrow[var]{} & M_1 \cdots\cdots & \xrightarrow[var]{} & \cdots A
\end{array}
$$

■

5 Operational equivalence

Operational equivalence is intended to reflect the notion of interchangeability of program fragments. It equates strictly more terms than does convertibility.

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,

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

Lemma 5.1 $\lambda_* \vdash M = N$ implies $\lambda_* \models M \cong N$.

Proof: Assume $\lambda_* \vdash M = N$, and suppose $\lambda_* \vdash C[M] = A$. Since convertibility is closed under term-formation, $\lambda_* \vdash M = N$ implies $\lambda_* \vdash C[M] = C[N]$, for any context C. The symmetry and transitivity of convertibility then imply that $\lambda_* \vdash C[N] = A$. The symmetric argument proves the converse implication, thus establishing $\lambda_* \vdash C[M] = A \Leftrightarrow \lambda_* \vdash C[N] = A$, which is the definition of $\lambda_* \models M \cong N$. ■

Lemma 5.2 For any context C, $\lambda_* \models M \cong N$ implies $\lambda_* \models C[M] \cong C[N]$.

Proof: Assume that $\lambda_* \models M \cong N$, and suppose that $\lambda_* \vdash C'[C[M]] = A$, for some context $C'[\ ]$ such that $C'[C[M]]$ and $C'[C[N]]$ are closed. Then the assumed operational equivalence implies that $\lambda_* \vdash C'[C[N]] = A$, and similar reasoning establishes the reverse implication. Thus $\lambda_* \models C[M] \cong C[N]$. ■

Lemma 5.3 For any variable x, $\lambda_* \models M \cong N \Leftrightarrow \lambda_* \models \lambda x.M \cong \lambda x.N$.

Proof: The left-to-right direction is a special case of Lemma 5.2.

To prove the converse, suppose that $\lambda_* \models \lambda x.M \cong \lambda x.N$, and suppose that $\lambda_* \vdash C[M] = A$, where $C[M]$ is closed. If $x \in fv\ M$, then $\lambda_* \vdash (\lambda x.M)x = M$, and $x \in bv(C[\ ])$. Let $C'[\ ] \equiv C[[\ ]x]$. Then $\lambda_* \vdash C'[\lambda x.M] = C[M] = A$. Since $\lambda_* \models \lambda x.M \cong \lambda x.N$ it follows that $\lambda_* \vdash C'[\lambda x.N] = A$, and therefore also $\lambda_* \vdash C[N] = A$. Since C was arbitrary, $\lambda_* \models M \cong N$. If $x \notin fv\ M$, choose $C'[\ ] = C[[\ ]c]$ for any constant c. ■

Proposition 5.4 The following are operational equivalences in λ_{var}:

$$
\begin{array}{lrcl}
(1) & v? \triangleright x\ .\ w? \triangleright y\ .\ M & \cong & w? \triangleright y\ .\ v? \triangleright x\ .\ M \\
(2) & N =: v\ ;\ N' =: w\ ;\ M & \cong & N' =: w\ ;\ N =: v\ ;\ M \\
 & & & (v \neq w) \\
(3) & \textbf{var}\ v\ .\ N =: w\ ;\ M & \cong & N =: w\ ;\ \textbf{var}\ v\ .\ M \\
 & & & (v \neq w, v \notin ft\ N) \\
(4) & \textbf{var}\ v\ .\ \textbf{var}\ w\ .\ M & \cong & \textbf{var}\ w\ .\ \textbf{var}\ v\ .\ M \\
(5) & N =: v\ ;\ N' =: v\ ;\ M & \cong & N' =: v\ ;\ M \\
(6) & S[P] & \cong & P \\
 & & & (bt\ S \cap ft\ (S[P]) = \emptyset) \\
(7) & P \triangleright x.\textbf{return}\ x & \cong & P
\end{array}
$$

We will prove these operational equivalances soon, but first we describe what they mean in terms of informal programming concepts.

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.

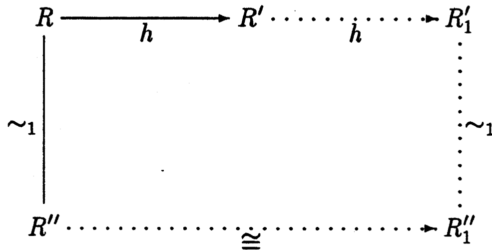
Equation (6) implements "garbage collection": it says that a state prefix S of an expression $S[P]$ can be dropped if no variable written in S is used in P. Note that, using the "bubble" conversion laws and the commutative laws (2), (3) and (4), garbage can always be moved to a state prefix.

Proof: The definition of operational equivalence demands that we prove statements that are quantified over all λ_{var}-contexts; the statements themselves are quantified over all answer-producing reductions. We use a proof technique developed by Odersky [20] to reduce the burden of this proof.

The technique in question requires us only to prove certain properties of the interaction between the proposed operation equivalence and the reduction rules of the calculus.

For each proof, we construct a collection of symmetric rewrite rules that define a relation, *similarity*, intended to characterize the operational equivalence to be proved. In our present proposition, this collection will always consist of a single rule: the operational equivalence as stated. We then show that, in all cases in which the similarity rules have a critical overlap with the reduction rules of the calculus, there exists a parallel application of (already-established) operational equivalences to the similar term that yields a term similar to the result of the reduction. This aspect of the technique resembles the definition of the *bisimulations* used in concurrency theory [17]. The notion of critical overlap arises in the theory of term-rewriting systems [6].

The main proof requirements for each operational equivalence are explained in the following diagram, in which the given relations appear as solid lines and the relations to be established appear as dotted lines:



where $\sim_1$ is the *parallel similarity* derived from $\sim$; in our proofs it will suffice to think of this relation as being the union $\sim \cup \equiv$.

For each proof below, we will give concrete meta-terms and reductions to instantiate this diagram.

Applying the proof technique requires in addition that we establish some minor technical conditions. First, we must verify that the evaluation contexts of λ_{var} are *downward closed*, that is, that $E = C_1 \cdot C_2$ implies that C_2 is itself an evaluation context. This is obvious from inspection of the definition of λ_{var}'s evaluation contexts in Figure 5. Second, we must establish, for each similarity relation we introduce, that the relation *preserves evaluation contexts* and is *answer-preserving*. Preservation of evaluation contexts means that the similarity rules do not overlap with the defining productions of evaluation contexts. An answer-preserving relation satisfies $M \sim A \Rightarrow M \twoheadrightarrow A$; this is easily seen to be true for all our proposed similarity relations, so we will omit further mention of this issue.

We now proceed to prove each of the proposed operational equivalences. For most of the individual proofs, the operational equivalence required to construct the bottom edge of the above diagram is easily established by exhibiting a conversion; we only remark on the exceptions to this observation.

(1) We adopt the similarity relation implied by $\mathcal{S} = \{v? \rhd x.w? \rhd y.M \sim w? \rhd y.v? \rhd x.M\}$.

The rules in $\mathcal{S}$ do not overlap with any of the productions defining evaluation contexts in Figure 5. Hence, $\sim$ preserves evaluation contexts.

We now enumerate the various ways in which $\sim$ can intefere with $\rightarrow$; for each case, we show how to instantiate the diagram so as to prove that the compatible equivalence closure of $\sim$ is an operational equivalence.

Case 1.1:

$$\begin{array}{rcl} R & \equiv & N =: u \; ; v? \rhd x \; . \; w?y \; . \; M \\ & & (u \not\equiv v, u \not\equiv w, v \not\equiv w) \\ R'' & \equiv & N =: u \; ; w? \rhd y \; . \; v? \rhd x \; . \; M \\ R' & \equiv & v? \rhd x \; . \; N =: u \; ; w? \rhd y \; . \; M \\ R'_1 & \equiv & v? \rhd x.w? \rhd y.N =: u \; ; M \\ R''_1 & \equiv & w? \rhd y.v? \rhd x.N =: u \; ; M \end{array}$$

$$\begin{array}{rcl} R & \xrightarrow[b_1]{} & R' \\ R' & \xrightarrow[b_1]{} & R'_1 \\ R'' & \xrightarrow[b_1]{} & w? \rhd y.N =: u \; ; v? \rhd x.M \\ & \xrightarrow[b_1]{} & R''_1 \end{array}$$

Case 1.2:

$$\begin{array}{rcl} R & \equiv & N =: w \; ; v? \rhd x.w? \rhd y.M \\ & & (u \not\equiv v, v \not\equiv w) \\ R'' & \equiv & N =: w \; ; w? \rhd y.v? \rhd x.M \\ R' & \equiv & v?x.N =: w \; ; w? \rhd y.M \\ R'_1 \equiv R''_1 & \equiv & v? \rhd x.N =: w \; ; [N/y]M \end{array}$$

$$\begin{array}{rcl} R & \xrightarrow[b_1]{} & R' \\ R' & \xrightarrow[f]{} & v? \rhd x.N =: w \; ; (y.M)N \\ & \xrightarrow[\beta]{} & R'_1 \end{array}$$

$$\begin{array}{rcl}
R'' & \xrightarrow[f]{} & N =: w \; ; \; (y.v? \triangleright x.M)N \\
 & \xrightarrow[\beta]{} & N =: w \; ; \; v? \triangleright x.[N/y]M \\
 & \xrightarrow[b_1]{} & R''_1
\end{array}$$

Case 1.3:

$$\begin{array}{rcl}
R & \equiv & N =: v \; ; \; v? \triangleright x.w? \triangleright y.M \\
 & & (v \not\equiv w) \\
R'' & \equiv & N =: v \; ; \; w? \triangleright y.v? \triangleright x.M \\
R' & \equiv & N =: v \; ; \; (x.w? \triangleright y.M)N \\
R'_1 \equiv R''_1 & \equiv & w? \triangleright y.N =: v \; ; \; [N/x]M
\end{array}$$

$$\begin{array}{rcl}
R & \xrightarrow[f]{} & R' \\
R'' & \xrightarrow[b_1]{} & w? \triangleright y.N =: v \; ; \; v? \triangleright x.M \\
 & \xrightarrow[f,\beta]{} & R''_1 \\
R' & \xrightarrow[\beta]{} & N =: v \; ; \; w? \triangleright y.[N/x]M \\
 & \xrightarrow[b_1]{} & R'_1
\end{array}$$

Case 1.4:

$$\begin{array}{rcl}
R & \equiv & N =: v \; ; \; v? \triangleright x.v? \triangleright y.M \\
 & & (u \not\equiv v, u \not\equiv w) \\
R' & \equiv & v? \triangleright x.N =: u \; ; \; v? \triangleright y.M \\
R'' & \equiv & N =: u \; ; \; v? \triangleright x.v? \triangleright y.M \\
R'_1 \equiv R''_1 & \equiv & R'
\end{array}$$

$$\begin{array}{rcl}
R & \xrightarrow[b_1]{} & R' \\
R'' & \xrightarrow[b_1]{} & R''_1
\end{array}$$

Case 1.5:

$$\begin{array}{rcl}
R & \equiv & N =: v \; ; \; v? \triangleright x.v? \triangleright y.M \\
R' & \equiv & N =: v \; ; \; (x.v? \triangleright y.M)N \\
R'' & \equiv & R \\
R'_1 & \equiv & N =: v \; ; \; v? \triangleright y.[N/x]M \\
R''_1 & \equiv & R'_1
\end{array}$$

$$\begin{array}{rcl}
R & \xrightarrow[f]{} & R' \\
R' & \xrightarrow[\beta]{} & R'_1 \\
R'' & \xrightarrow[f,\beta]{} & R''_1
\end{array}$$

For the remaining operational equivalences, we give only the basic data needed to complete the diagram.

(2) $\mathcal{S} = \{N =: v \; ; \; N' =: w \; ; \; M \sim N' =: w \; ; \; N =: v \; ; \; M\}, (v \not\equiv w)$.

Case 2.1:

$$\begin{array}{rcl}
R & \equiv & N =: v \; ; \; N' =: w \; ; \; w? \triangleright x.M' \\
 & & (v \not\equiv w) \\
R' & \equiv & N =: v \; ; \; N' =: w \; ; \; (x.M)N' \\
R'_1 & \equiv & R' \\
R'' & \equiv & N' =: w \; ; \; N =: v \; ; \; w? \triangleright x.M' \\
R''_1 & \equiv & N' =: w \; ; \; N =: v \; ; \; (x.M)N'
\end{array}$$

$$\begin{array}{rcl}
R & \xrightarrow[f]{} & R' \\
R'' & \xrightarrow[b_1]{} & N' =: w \; ; \; w? \triangleright x.n =: v \; ; \; M' \\
 & & \text{note that } x \notin fv \; N \\
 & \xrightarrow[f]{} & N' =: w \; ; \; (x.N =: v \; ; \; M')N' \\
 & \xrightarrow[\beta]{} & N' =: w \; ; \; N =: v \; ; \; [N'/x]M' \\
 & \xleftarrow[\beta]{} & R''_1
\end{array}$$

Case 2.2:

$$\begin{array}{rcl}
R & \equiv & N =: v \; ; \; N' =: w \; ; \; v? \triangleright x.M \\
 & & (v \not\equiv w) \\
R' & \equiv & N =: v \; ; \; v? \triangleright x.N' =: w \; ; \; M' \\
 & & (x \notin fv \; N') \\
R'' & \equiv & N' =: w \; ; \; N =: v \; ; \; v? \triangleright x.M' \\
R'_1 & \equiv & N =: v \; ; \; N' =: w \; ; \; [N/x]M' \\
R''_1 & \equiv & N' =: w \; ; \; N =: v \; ; \; [N/x]M'
\end{array}$$

$$\begin{array}{rcl}
R & \xrightarrow[b_1]{} & R' \\
R' & \xrightarrow[f]{} & N =: v \; ; \; (x.N' =: w \; ; \; M')N \\
 & \xrightarrow[\beta]{} & R'_1 \\
R'' & \xrightarrow[f]{} & N' =: w \; ; \; N =: v \; ; \; (x.M')N \\
 & \xrightarrow[\beta]{} & R''_1
\end{array}$$

Case 2.3:

$$\begin{array}{rcl}
R & \equiv & N =: v \; ; \; N' =: w \; ; \; u? \triangleright x.M' \\
 & & (u \not\equiv w, u \not\equiv v) \\
R' & \equiv & N =: v \; ; \; u? \triangleright x.N' =: w \; ; \; M' \\
R'' & \equiv & N' =: w \; ; \; N =: v \; ; \; u? \triangleright x.M' \\
R'_1 & \equiv & u? \triangleright x.N =: v \; ; \; N' =: w \; ; \; M' \\
R''_1 & \equiv & u? \triangleright x.N' =: w \; ; \; N =: v \; ; \; M'
\end{array}$$

$$\begin{array}{rcl}
R & \xrightarrow[b_1]{} & R' \\
R' & \xrightarrow[b_1]{} & R'_1 \\
R'' & \xrightarrow[b_1]{} & R''_1
\end{array}$$

(3) $\mathcal{S} = \{\mathbf{var}\ v.N =: w \ ;\ M \sim N =: w \ ;\ \mathbf{var}\ v.N \mid v \not\equiv w, v \notin \mathit{ft}\ N\}$

Case 3.1:

$$\begin{array}{rcl}
R & \equiv & N =: w \ ;\ \mathbf{var}\ v.u? \triangleright x.M \\
 & & (u \not\equiv w, u \not\equiv v) \\
R' & \equiv & N =: w \ ;\ u? \triangleright x.\mathbf{var}\ v.M \\
R'' & \equiv & \mathbf{var}\ v.N =: w \ ;\ u? \triangleright x.M \\
R'_1 & \equiv & u? \triangleright x.N =: w \ ;\ \mathbf{var}\ v.M \\
R''_1 & \equiv & u? \triangleright x.\mathbf{var}\ v.N =: w \ ;\ M
\end{array}$$

$$\begin{array}{rcl}
R & \underset{b_2}{\longrightarrow} & R' \\
R' & \underset{b_1}{\longrightarrow} & R'_1 \\
R'' & \underset{b_1}{\longleftarrow} & \mathbf{var}\ v.u? \triangleright x.N =: w \ ;\ M \\
 & \underset{b_2}{\longrightarrow} & R''_1
\end{array}$$

Case 3.2:

$$\begin{array}{rcl}
R & \equiv & N =: w \ ;\ \mathbf{var}\ v.w? \triangleright x.M \\
 & & (u \not\equiv v) \\
R' & \equiv & N =: w \ ;\ w? \triangleright x.\mathbf{var}\ v.M \\
R'' & \equiv & \mathbf{var}\ v.N =: w \ w? \triangleright x.M \\
R'_1 & \equiv & N =: w \ ;\ \mathbf{var}\ v.[N/x]M \\
R''_1 & \equiv & \mathbf{var}\ v.N =: w \ ;\ [N/x]M
\end{array}$$

$$\begin{array}{rcl}
R & \underset{b_2}{\longrightarrow} & R' \\
R' & \underset{f}{\longrightarrow} & N =: w \ ;\ (x.\mathbf{var}\ v.M)N \\
 & \underset{\beta}{\longrightarrow} & R'_1 \\
R'' & \underset{f}{\longrightarrow} & \mathbf{var}\ v.N =: w \ ;\ (x.M)N \\
 & \underset{\beta}{\longrightarrow} & R''_1
\end{array}$$

Case 3.3:

$$\begin{array}{rcl}
R & \equiv & \mathbf{var}\ v.N =: w \ ;\ u? \triangleright x.P \\
 & & (u \not\equiv w, u \not\equiv v) \\
R' & \equiv & \mathbf{var}\ v.u? \triangleright x.N =: w \ ;\ P \\
R'' & \equiv & N =: w \ ;\ \mathbf{var}\ v.u? \triangleright x.P \\
R'_1 & \equiv & u? \triangleright x.\mathbf{var}\ v.N =: w \ ;\ P \\
R''_1 & \equiv & u? \triangleright x.N =: w \ ;\ \mathbf{var}\ v.P
\end{array}$$

$$\begin{array}{rcl}
R & \underset{b_1}{\longrightarrow} & R' \\
R' & \underset{b_2}{\longrightarrow} & R'_1 \\
R'' & \underset{b_2}{\longrightarrow} & N =: w \ ;\ u? \triangleright x.\mathbf{var}\ v.P \\
 & \underset{b_1}{\longrightarrow} & R''_1
\end{array}$$

Case 3.4:

$$\begin{array}{rcl}
R & \equiv & \mathbf{var}\ v.N =: w \ ;\ w? \triangleright x.P \\
R' & \equiv & \mathbf{var}\ v.N =: w \ ;\ (x.P)N \\
R'' & \equiv & N =: w \ ;\ \mathbf{var}\ v.w? \triangleright x.P \\
R'_1 & \equiv & \mathbf{var}\ v.N =: w \ ;\ [N/x]P \\
R''_1 & \equiv & N =: w \ ;\ \mathbf{var}\ v.[N/x]P
\end{array}$$

$$\begin{array}{rcl}
R & \underset{f}{\longrightarrow} & R' \\
R' & \underset{\beta}{\longrightarrow} & R'_1 \\
R'' & \underset{b_2}{\longrightarrow} & N =: w \ ;\ w? \triangleright x.\mathbf{var}\ v.P \\
 & \underset{f}{\longrightarrow} & N =: w \ ;\ (x.\mathbf{var}\ v.P)N \\
 & \underset{\beta}{\longrightarrow} & R''_1
\end{array}$$

Case 3.5:

$$\begin{array}{rcl}
R & \equiv & (\mathbf{var}\ v.N =: w \ ;\ P) \triangleright x.Q \\
R' & \equiv & \mathbf{var}\ v.N =: w \ ;\ P \triangleright x.Q \\
R'' & \equiv & (N =: w \ ;\ \mathbf{var}\ v.P) \triangleright x.Q \\
R'_1 & \equiv & R' \\
R''_1 & \equiv & N =: w \ ;\ \mathbf{var}\ v.P \triangleright x.Q
\end{array}$$

$$\begin{array}{rcl}
R & \underset{v\triangleright}{\longrightarrow} & R' \\
R'' & \underset{\triangleright\triangleright}{\longrightarrow} & R''_1
\end{array}$$

(4) $\mathcal{S} = \{\mathbf{var}\ v.\mathbf{var}\ w.M \sim \mathbf{var}\ w.\mathbf{var}\ v.M\}$

Some of the critical pairs we encounter in the course of proving this operational equivalence require more refined reasoning than we have yet encountered.

Case 4.1:

$$\begin{array}{rcl}
R & \equiv & (\mathbf{var}\ v.\mathbf{var}\ w.M) \triangleright x.N \\
R' & \equiv & \mathbf{var}\ v.(\mathbf{var}\ w.M) \triangleright x.N \\
R'' & \equiv & (\mathbf{var}\ w.\mathbf{var}\ v.M) \triangleright x.N \\
R'_1 & \equiv & \mathbf{var}\ v.\mathbf{var}\ w.(M \triangleright x.N) \\
R''_1 & \equiv & \mathbf{var}\ w.\mathbf{var}\ v.(M \triangleright x.N)
\end{array}$$

$$\begin{array}{rcl}
R & \underset{v\triangleright}{\longrightarrow} & R' \\
R' & \underset{v\triangleright}{\longrightarrow} & R'_1 \\
R'' & \underset{v\triangleright}{\longrightarrow} & \mathbf{var}\ w.(\mathbf{var}\ v.M) \triangleright x.N \\
 & \underset{v\triangleright}{\longrightarrow} & R''_1
\end{array}$$

Case 4.2:

$$\begin{array}{rcl}
R & \equiv & \mathbf{var}\ v.\mathbf{var}\ w.u? \triangleright x.M \\
 & & (u \not\equiv v, u \not\equiv w)
\end{array}$$

$$
\begin{array}{rcl}
R' & \equiv & \mathbf{var}\ v.u?\triangleright x.\mathbf{var}\ w.M \\
R'' & \equiv & \mathbf{var}\ w.\mathbf{var}\ v.u?\triangleright x.M \\
R_1' & \equiv & u?\triangleright x.\mathbf{var}\ v.\mathbf{var}\ w.N \\
R_1'' & \equiv & u?\triangleright x.\mathbf{var}\ w.\mathbf{var}\ v.N
\end{array}
$$

$$
\begin{array}{rcl}
R & \xrightarrow[b_2]{} & R' \\
R' & \xrightarrow[b_2]{} & R_1' \\
R'' & \xrightarrow[b_2]{} & \mathbf{var}\ w.u?\triangleright x.\mathbf{var}\ v.M \\
 & \xrightarrow[b_2]{} & R_1''
\end{array}
$$

Case 4.3:

$$
\begin{array}{rcl}
R & \equiv & \mathbf{var}\ v.\mathbf{var}\ w.v?\triangleright x.M \\
 & & (v \not\equiv w) \\
R' & \equiv & \mathbf{var}\ v.v?\triangleright x.\mathbf{var}\ w.M \\
R'' & \equiv & \mathbf{var}\ w.\mathbf{var}\ v.v?\triangleright x.M \\
R_1' & \equiv & R' \\
R_1'' & \equiv & R_1'
\end{array}
$$

$$
\begin{array}{rcl}
R & \xrightarrow[b_2]{} & R' \\
R' & \xrightarrow[b_2]{} & R_1' \\
R'' & \cong & R'
\end{array}
$$

by the following argument:

Informally, both R'' and R' are stuck terms: the only way a program containing either one as a subterm can reduce to an answer is to throw the term away, since no rule (including the δ-rule), can examine the substructure of terms constructed with **var** (or any other term except a value or a fully-applied constructor). Thus one would expect that putting both terms in the same context would yield a computation that either gets stuck or yields the same result regardless of the terms in question.

Formally, we apply the critical-pair method once again. We define an auxiliary similarity relation $\mathcal{S}' = \{\mathbf{var}\ v.v?\triangleright x.P \sim \mathbf{var}\ w.\mathbf{var}\ v.v?\triangleright x.P' \mid v \not\equiv w\}$. This relation interferes with no reduction rules whatsoever.

(5) $\mathcal{S} = \{N =: v\ ;\ N' =: v\ ;\ M \sim N' =: v\ ;\ M\}$

Case 5.1:

$$
\begin{array}{rcl}
R & \equiv & N' =: v\ ;\ v?\triangleright x.M \\
R' & \equiv & N' =: v\ ;\ (x.M)N' \\
R'' & \equiv & N =: v\ ;\ N' =: v\ ;\ v?\triangleright x.M \\
R_1' & \equiv & R' \\
R_1'' & \equiv & N =: v\ ;\ N' =: v\ ;\ (x.M)N'
\end{array}
$$

$$
\begin{array}{rcl}
R & \xrightarrow[f]{} & R' \\
R'' & \xrightarrow[f]{} & R_1''
\end{array}
$$

Case 5.2:

$$
\begin{array}{rcl}
R & \equiv & N' =: v\ ;\ w?\triangleright x.M \\
R' & \equiv & w?\triangleright x.N' =: v\ ;\ M \\
R'' & \equiv & N =: v\ ;\ N' =: v\ ;\ w?\triangleright x.M \\
R_1' & \equiv & R' \\
R_1'' & \equiv & w?\triangleright x.N =: v\ ;\ N' =: v\ ;\ M
\end{array}
$$

$$
\begin{array}{rcl}
R & \xrightarrow[b_1]{} & R' \\
R'' & \xrightarrow[b_1]{} & N =: v\ ;\ w?\triangleright x.N' =: v\ ;\ M \\
 & \xrightarrow[b_1]{} & R_1''
\end{array}
$$

Case 5.3:

$$
\begin{array}{rcl}
R & \equiv & N =: v\ ;\ N' =: v\ ;\ v?\triangleright x.M \\
R' & \equiv & N =: v\ ;\ N' =: v\ ;\ (x.M)N' \\
R'' & \equiv & N' =: v\ ;\ v?\triangleright x.M \\
R_1' & \equiv & R' \\
R_1'' & \equiv & N =: v\ ;\ N' =: v\ ;\ (x.M)N'
\end{array}
$$

$$
\begin{array}{rcl}
R & \xrightarrow[f]{} & R' \\
R'' & \xrightarrow[f]{} & R_1''
\end{array}
$$

Case 5.4:

$$
\begin{array}{rcl}
R & \equiv & N =: v\ ;\ N' =: v\ ;\ u?\triangleright x.M \\
 & & (u \not\equiv v) \\
R' & \equiv & N =: v\ ;\ u?\triangleright x.N' =: v\ ;\ M \\
R'' & \equiv & N' =: v\ ;\ u?\triangleright x.M \\
R_1' & \equiv & u?\triangleright x.N =: v\ ;\ N' =: w\ ;\ M \\
R_1'' & \equiv & u?\triangleright x.N' =: v\ ;\ M
\end{array}
$$

$$
\begin{array}{rcl}
R & \xrightarrow[b_1]{} & R' \\
R' & \xrightarrow[b_1]{} & R_1' \\
R'' & \xrightarrow[b_1]{} & R_1''
\end{array}
$$

(6) $\mathcal{S} = \{\mathcal{S}[P] \sim P \mid \mathcal{S}$ a state context, $bt\ \mathcal{S} \cap ft\ \mathcal{S}' = \emptyset\}$ This similarity relation can interfere both with the purification rules and with the assignment rules. All the critical pairs having to do with the purification laws yield essentially the same diagram verification task, so we give only a single example:

$$\begin{array}{rcl} R & \equiv & \textbf{pure}(S[\textbf{return}\ x.M]) \\ R' & \equiv & x.\textbf{pure}(S[\textbf{return}\ M]) \\ R'' & \equiv & \textbf{pure}(\textbf{return}\ x.M) \\ R'_1 & \equiv & R' \\ R''_1 & \equiv & x.\textbf{pure}(\textbf{return}\ M) \end{array}$$

$$\begin{array}{rcl} R & \xrightarrow[p\lambda]{} & R' \\ R'' & \xrightarrow[p\lambda]{} & R''_1 \end{array}$$

The uses of $\sim$ in the converse direction are equally simple to verify.

The verification of the proof conditions in the cases of interference with assignment rules requires that we invoke our knowledge concerning the structure of S'. By the definition of a state prefix (Section 2), all assigments in S' must be to tags defined in S'. Also, we have assumed that S' is nonempty, so the immediate context of the metavariable M in the definition of S is either of the form **var** $v.[\,]$ or the form $N =: v\ ;\ [\,]$.

We work out one of the three simple cases of interference with an assignment rule:

$$\begin{array}{rcl} R & \equiv & S[N =: v\ ;\ w? \triangleright x.M] \\ R' & \equiv & S[w? \triangleright x.N =: v\ ;\ M] \\ R'' & \equiv & N =: v\ ;\ w? \triangleright x.M \\ R'_1 & \equiv & R' \\ R''_1 & \equiv & w? \triangleright x.N =: v\ ;\ M \end{array}$$

$$\begin{array}{rcl} R & \xrightarrow[b_1]{} & R' \\ R'' & \xrightarrow[b_1]{} & R''_1 \end{array}$$

(7) For this equivalence, we adopt the similarity relation defined by $\mathcal{S} = \{P \triangleright x.\textbf{return}\ x \sim P\}$, where the metavariable P ranges over procedures only. This similarity relation only interferes with the rule ($\triangleright\triangleright$).

$$\begin{array}{rcl} R & \equiv & (P \triangleright x.\textbf{return}\ x) \triangleright y.Q \\ R' & \equiv & P \triangleright x.(\textbf{return}\ x \triangleright y.Q) \\ R'' & \equiv & P \triangleright y.Q \\ R'_1 & \equiv & P \triangleright x.(y.Q)x \\ R''_1 & \equiv & R'_1 \end{array}$$

$$\begin{array}{rcl} R & \xrightarrow[\triangleright\triangleright]{} & R' \\ R' & \xrightarrow[r\triangleright]{} & R'_1 \\ R'' & \cong & R''_1 \end{array}$$

To establish the auxiliary operational equivalence $R'' \cong R''_1$, we define a similarity relation by $\mathcal{S}' = \{y.M \sim x.(y.M)x \mid x \notin fvM\}$. This similarity only interferes with the β-rule:

$$\begin{array}{rcl} R & \equiv & (x.(y.M)x)N \\ R' & \equiv & (y.M)N \\ R'' & \equiv & (y.M)N \\ R'_1 & \equiv & R' \\ R''_1 & \equiv & R'' \end{array}$$

$$R \xrightarrow[\beta]{} R'$$

■

6 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 fixed points. 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[5]. The only provision on this result is that the underlying set of constructors and basic function symbols needs to be "sufficiently rich".

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. (That is, we can always find "fresh" constructors that are not used in the reduction of some given program).

[5]The latter property can be regarded as the preservation of *distinctions* or *inequivalences* between terms.

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

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

(That is, one can test for a constructor, and can take apart constructed terms).

(c) One can define in λ_* a function projector $proj_f$ such that

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

(That is, one can test for a function).

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 [4].

In the following, we will assume that λ_{var} has a sufficiently rich set of constants that is shared by all other calculi in consideration (i.e. λ_σ, $\lambda\nu$, and λ)

The rest of this section is devoted to the proof that λ_{var} is an operational extension of λ. This proof is based on finding an implementation of λ_{var} in λ. We will show below (in Theorem 6.15) that existence of an implementation implies operational extension, provided the implementation is sound and complete and maps closed λ terms to themselves. We call such an implementation a syntactic embedding.

Definition 6.1 (Extended Term) Given an inductively defined term language Λ_*, an *extended term* is formed from the inductive definition of Λ_* and [ ]. (Hence, both terms and contexts are extended terms).

Definition 6.2 A term M is λ-closed iff $FV(M) = \emptyset$.

Definition 6.3 (Syntactic Embedding) Let λ_* and λ_0 be extensions of λ with the same set of answers and suppose that $\Lambda_* \supseteq \Lambda_0$. Let $\mathcal{E}$ be a syntactic mapping from extended λ_*-terms to extended λ_0-terms. Then $\mathcal{E}$ is a *syntactic embedding* of λ_* in λ_0 if it satisfies the following two requirements.

1. $\mathcal{E}$ preserves λ-closed λ_0-subterms up to convertibility. For all λ_*-contexts C, λ-closed λ_0-terms M,

$$\lambda_0 \vdash \mathcal{E}[\![C[M]]\!] = \mathcal{E}[\![C]\!]\,[M].$$

2. $\mathcal{E}$ preserves semantics. For all closed λ_*-terms M, answers A,

$$\lambda_* \vdash M = A \Leftrightarrow \lambda_0 \vdash \mathcal{E}[\![M]\!] = A.$$

Syntactic embeddings generalize the syntactic (macro) expansions of [7]. They are useful for two reasons. First, a syntactic embedding $\mathcal{E}$ of λ_* in λ_0 allows us to extend models of λ_0 to all of λ_*, simply by composing the denotation function of λ_0 with $\mathcal{E}$. Second, and somewhat related to the first point, the existence of a syntactic embedding between two theories guarantees that all operational equivalences of the base theory are preserved in the extension. Indeed, we have:

Theorem 6.4 Let λ_* and λ_0 be two extensions of λ with the same set of answers and suppose that $\Lambda_* \supseteq \Lambda_0$. If there is a syntactic embedding of λ_* in λ_0 then for any two terms $M, N \in \Lambda_0$,

$$\lambda_0 \models M \cong N \;\Rightarrow\; \lambda_* \models M \cong N.$$

Proof: Assume that $\lambda_0 \models M \cong N$ Then, for all answers A and λ_0-contexts C_0 such that $C_0[M]$ and $C_0[N]$ are closed,

$$\lambda_0 \vdash C_0[M] = A \Leftrightarrow \lambda_0 \vdash C_0[N] = A.$$

Assume first that both M and N are λ-closed. Let $\mathcal{E}$ be a syntactic embedding of λ_* in λ_0. Let C be an arbitrary λ_*-context such that $C[M]$ and $C[N]$ are closed, and let A be an answer such that $C[M] = A$. Then,

$$\begin{array}{ll} & \lambda_* \vdash C[M] = A \\ \Leftrightarrow & (\mathcal{E} \text{ preserves semantics}) \\ & \lambda_0 \vdash \mathcal{E}[\![C[M]]\!] = A \\ \Leftrightarrow & (\mathcal{E} \text{ preserves } \lambda\text{-closed } \lambda_0 \text{ terms}) \\ & \lambda_0 \vdash \mathcal{E}[\![C]\!]\,[M] = A \\ \Leftrightarrow & (\text{premise: } \lambda_0 \models M \cong N) \\ & \lambda_0 \vdash \mathcal{E}[\![C]\!][N] = A \\ \Leftrightarrow & (\text{reverse the argument}) \\ & \lambda_* \vdash C[N] = A. \end{array}$$

Since A and C were arbitrary, $\lambda_* \models M \cong N$. Now let M and N be arbitrary λ_0 terms, with $fv\ M \cup fv\ N = \{x_1, \ldots, x_n\}$. Then,

$$\begin{array}{ll} & \lambda_0 \vdash M \cong N \\ \Rightarrow & (\text{Lemma 5.2}) \\ & \lambda_0 \vdash \lambda x_1. \ldots \lambda x_n.M \cong \lambda x_1. \ldots \lambda x_n.N \\ \Rightarrow & (\text{first part of proof}) \\ & \lambda_* \vdash \lambda x_1. \ldots \lambda x_n.M \cong \lambda x_1. \ldots \lambda x_n.N \\ \Rightarrow & (\text{Lemma 5.3}) \\ & \lambda_* \vdash M \cong N. \end{array}$$

■

Before we state and prove the operational extension theorem, let us quickly examine why finding a syntactic embedding of λ_{var} in λ is non-trivial. A natural implementation of λ_{var} in λ maps readers and assignments to read and write operations on a store. This suggests that we should work with a calculus with explicit stores. Because all operational equivalences of λ_{var} are preserved in λ_σ (Theorem 4.16), we can use just as well λ_σ instead of λ_{var} as the source language of the embedding. A store in λ_σ can be represented as a mapping from locations to terms. Variables in λ_σ are then mapped to locations. But what is a location? Conventionally, it is an element of some index type such as an integer. This is sufficient if there is only one global store, but the scheme breaks down for λ_{var}. The problem is that a λ_{var} term can contain multiple **pure** blocks. This makes it necessary to distinguish in the implementation between locations that are defined in a given **pure** block and locations defined in other blocks. The distinction is needed since only local variables may be read or written, and only global locations may be returned from a **pure** block.

Of course, local and global variables can be distinguished if they are represented by different locations (or, equivalently, if locations are tagged with a unique identifier associated with the block in which they are allocated). The problem then is how to ensure that all variable instances in a program (as opposed to those a single **pure** block) are mapped to unique locations. The obvious idea of passing a name supply to a **pure** block is not open to us, since then functions would need additional implicit arguments for the name-supply. This would violate the property that a syntactic embedding maps λ-closed terms in the base language to themselves.

Perhaps surprisingly, there is a way to ensure uniqueness of locations without passing a name supply to **pure**-blocks. This is shown in [21], where a calculus $\lambda\nu$ is studied. $\lambda\nu$ is an extension of the λ-calculus with a binding construct for names. The term language of $\lambda\nu$ extends Λ by three new constructs:

$$M \quad ::= \quad \dots \mid n \mid \nu n.M \mid M_1 == M_2$$

Names form a new countable alphabet. A name n is bound in a construct $\nu n.M$. Like identifiers in λ (and variables in λ_{var}), names can be α-renamed. The only operation applicable to names is the equality test $n_1 == n_2$. $(==)$ is syntactic equality, its behavior is given by the reduction rules

$$\begin{array}{lcll} n == n & \rightarrow & \mathit{true} & \\ n == m & \rightarrow & \mathit{false} & \textbf{if } n \neq m \end{array}$$

The other reductions that are particular to $\lambda\nu$ involve ν-terms:

$$\begin{array}{lcll} \nu n.\lambda x.M & \rightarrow & \lambda x.\nu n.M & \\ \nu n.c^n\ M_1 \dots M_n & \rightarrow & c^n\ (\nu n.M_1) \dots (\nu n.M_n) & \\ \nu n.m & \rightarrow & m & \textbf{if } n \neq m \end{array}$$

Standard reduction in $\lambda\nu$ is defined as in λ, with the added production

$$E ::= \dots \mid \nu n.E$$

in the grammar for evaluation contexts.

For a more comprehensive discussion of $\lambda\nu$, see [21][6]. Also in [21] it is shown that $\lambda\nu$ is an operational extension of λ. The proof of this theorem relies on a a syntactic embedding that maps names to level numbers. Similarly to a DeBruijn number, a level number indicates the number of intervening ν abstractions between the occurrence of a name and its definition.

Proposition 6.5 There exists a syntactic embedding $\mathcal{E}_\nu$ of $\lambda\nu$ in λ.

Proof: See [21]. ∎

Since λ_σ preserves all operational equivalences of λ, it suffices for the proof of operational extension to give a syntactic embedding of λ_σ in λ. 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 6 carry over, except that the first three rules are now defined on lists (with $+\!\!+$ as the *append* operator) rather than sets:

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

Clearly, Theorem 4.16 holds for the new just as for the original λ_σ calculus.

The embedding of this variant of λ_σ in λ, $\mathcal{E}_\sigma$ is given by

$$\mathcal{E}_\sigma[\![M]\!] \quad = \quad \mathcal{E}_\nu[\![\ \mathcal{F}[\![M]\!]\bot\ ]\!]$$

where the mapping $\mathcal{F}$ is defined in Figure 7. $\mathcal{F}[\![M]\!]$ takes as argument an environment ρ that maps identifiers and

[6] The calculus defined in [21] is a slight variation of the calculus described here, with unary constructors and pairs instead of the n-ary constructors described here. The difference does not influence the operational extension results.

$$
\begin{array}{lll}
\mathcal{F}[\![f]\!]\rho & = & f \\
\mathcal{F}[\![c^n]\!]\rho & = & c^n \\
\mathcal{F}[\![x]\!]\rho & = & \rho x \\
\mathcal{F}[\![v]\!]\rho & = & \rho v \\
\mathcal{F}[\![x.M]\!]\rho & = & \lambda y.\mathcal{F}[\![M]\!]\rho[x \mapsto y] \qquad (y \notin range\ \rho) \\
\mathcal{F}[\![MN]\!]\rho & = & (\mathcal{F}[\![M]\!]\rho)\ (\mathcal{F}[\![N]\!]\rho) \\
\mathcal{F}[\![\mathbf{var}\ v.M]\!]\rho & = & \nu t.newref\ t\ (\mathcal{F}[\![M]\!]\rho[v \mapsto t]) \qquad (t \notin range\ \rho) \\
\mathcal{F}[\![M?]\!]\rho & = & deref\ (\mathcal{F}[\![M]\!]\rho) \\
\mathcal{F}[\![M =: N]\!]\rho & = & assign\ (\mathcal{F}[\![M]\!]\rho)\ (\mathcal{F}[\![N]\!]\rho) \\
\mathcal{F}[\![\mathbf{return}\ M]\!]\rho & = & return\ (\mathcal{F}[\![M]\!]\rho) \\
\mathcal{F}[\![M \triangleright x.N]\!]\rho & = & bind\ (\mathcal{F}[\![M]\!]\rho)\ (\mathcal{F}[\![x.N]\!]\rho) \\
\mathcal{F}[\![\mathbf{pure}\ M]\!]\rho & = & pure\ [\![M]\!]\rho \\
\mathcal{F}[\![\sigma \cdot M]\!]\rho & = & \nu t_1. \ldots \nu t_n.block\ (\mathcal{F}[\![M]\!]\rho')\ (\mathcal{F}[\![\sigma]\!]\rho') \\
 & & \mathbf{where}\ \{v_1, \ldots, v_n\} = dom\ \sigma \\
 & & \qquad \rho' = \rho[v_1 \mapsto t_1] \ldots [v_n \mapsto t_n] \\
 & & \qquad t_1, \ldots, t_n \notin range\ \rho
\end{array}
$$

$$
\mathcal{F}[\![[v_1 : M_1, \ldots, v_n : M_n]]\!]\rho = upd\ (Def\ \mathcal{F}[\![M_n]\!]\rho)\ (\mathcal{F}[\![v_n]\!]\rho)\ (\ldots(upd\ (Def\ \mathcal{F}[\![M_1]\!]\rho)\ (\mathcal{F}[\![v_1]\!]\rho)\ \bot)\ldots)
$$

$$
\begin{array}{llll}
block\ p\ s & & = & unwrap\ (p\ (\lambda x.\lambda s.(x, s))\ s) \\
unwrap\ (a, s) & & = & \mathbf{case}\ a\ \mathbf{of} \\
 & & & \quad Res\ (c^n\ M_1 \ldots M_n) \to c^n\ (unwrap\ (Res\ M_1, s)) \ldots (unwrap\ (Res\ M_n, s)) \\
 & & & \quad Res\ f \to \lambda x.unwrap\ (f\ x, s) \qquad (f \text{ a function}) \\
pure\ p & & = & block\ p\ \bot \\
return\ a & k\ s & = & k\ (Res\ a)\ s \\
bind\ p\ q & k\ s & = & p\ (\lambda x.q\ (strip\ x)\ k)\ s \\
 & & & \mathbf{where}\ strip\ (Res\ x) = x \\
 & & & \qquad\quad strip\ Unit = () \\
newref\ t\ p & k\ s & = & p\ k\ (upd\ Undef\ t\ s) \\
deref\ t & k\ s & = & \mathbf{case}\ s\ t\ \mathbf{of}\ Def\ a \to k\ (Res\ a)\ s \\
assign\ a\ t & k\ s & = & \mathbf{case}\ s\ t\ \mathbf{of}\ Undef, Def\ y \to k\ Unit\ (upd\ (Def\ a)\ t\ s) \\
upd\ a\ t\ s & & = & \lambda u.\mathbf{if}\ t == u\ \mathbf{then}\ a\ \mathbf{else}\ s\ u
\end{array}
$$

Figure 7: Environment mapping $\mathcal{F}$

variables in λ_σ to identifiers and names in $\lambda\nu$. $\perp$ denotes the totally undefined environment, and $\rho[x \mapsto y]$ denotes the environment ρ extended by the binding that sends x to y. To save parentheses, we adopt the convention that environment extension binds more strongly than function application, i.e. $\mathcal{F}[\![M]\!]\rho[x \mapsto y]$ parses as $\mathcal{F}[\![M]\!](\rho[x \mapsto y])$.

Some other remarks on Figure 7:

- $\mathcal{F}$ maps an applicative term in λ_σ to itself (modulo α renaming). Procedures are mapped to functions that access and modify an explicit state.

- A state is a mapping from $\lambda\nu$ names to either *Def M* where M is a term, or *Undef*. *Undef* is returned if a variable is uninitialized. A store $\{v_1 : M_1, \ldots, v_n : M_n\}$ in λ_σ is translated to the state that maps each (image of) v_i to *Undef*, if $M_i \equiv \mathbf{undef}$, and to *Def* $(\mathcal{F}[\![M_i]\!]\rho)$ otherwise. The translated state is undefined on all other names. Such a function can be constructed from $\perp$ (the totally undefined state) and *upd*. Note that the definition of *upd* relies on the equality test (==) on names. Note also that an uninitialized variable is different from a variable containing $\perp$ (where $\perp$ is any λ-term without weak head normal form). A procedure that reads from an uninitialized variable always gets stuck and hence never reduces to an answer. If it reads a variable whose value is $\perp$ the procedure might still reduce to an answer, namely if the value of the variable is not demanded in the subsequent computation.

- A variable v in λ_σ is mapped to a $\lambda\nu$ name n.

- A procedure in λ_σ is mapped to a state-transformer. A state-transformer is a function from continuations and states to results. A continuation is a function from intermediate results and states to final results. Both intermediate and final results are wrapped in a constructor *Res* (or the result is *Unit* if it comes from an assignment; see below). It is assumed that *Res* and *Unit* are reserved, i.e. that they are not available for the reduction of pure $\lambda\nu$ terms. This can always be achieved since λ_{var} has by assumption a sufficiently rich set of constants that is shared by the other calculi in consideration.

  Wrapping results in reserved constructors serves to keep state transformers apart from other $\lambda\nu$ functions. Without it, a term such as **pure** f, where f is a $\lambda\nu$ function that behaves like a state-transformer, would be translated to *pure* f, which might well reduce to a result. On the other hand, the original term **pure** f will always get stuck, since f is not a procedure. Hence, $\mathcal{E}_\sigma$ would fail to preserve semantics. Note that in a typed setting equivalent abstraction barriers could be provided by abstract data types.

- The definition of $\mathcal{F}$ makes use of the auxiliary functions given in the second half of Figure 7. To help readability, we give these functions in a sugared notation similar to Haskell [14] instead of in pure $\lambda\nu$.

- Function *block* invokes its state-transformer argument p with its state argument s and the continuation $\lambda x.\lambda s.(x, s)$. This continuation simply returns the intermediate result and the state passed to it. The result portion of this pair is then unwrapped using function *unwrap*. This is similar to the actions of productions σ_{pc}, σ_{p_λ} and σ_{p_f}.

- The term $\nu n_1. \ldots .\nu n_m.unwrap\ p\ k\ s$ corresponds to a λ_σ configuration with state s and procedure $p \triangleright k$. The domain of s is $\{n_1, \ldots, n_m\}$.

- The state-transformer *return* x invokes its continuation in the current state with *Res* x as intermediate result.

- The state-transformer *bind* p q invokes the state-transformer p with q as a prefix of the continuation.

- A variable abstraction **var** $v.M$ is mapped to the $\lambda\nu$-term $\nu n.newref\ n\ (\mathcal{F}[\![M]\!]\rho[v \mapsto n])$. Function *newref* n p invokes its state-transformer argument p in a state in which its name argument n is bound to *Undef*.

- The state-transformer *deref* n forces a read of n in the current state. This corresponds to the strict behavior of readers in λ_{var} where the read is performed even if its result is not demanded by the subsequent computation.

- The state-transformer *assign* x n also forces a read of n, but then performs the same action regardless of the value associated with n. This ensures that the name n has in fact been entered (via *newref*) in the current state and thus guards against non-local assignments. *assign* updates the state by the binding that sends n to *Def* x and invokes its continuation argument with *Unit* as intermediate result. If the assignment appears as first argument of a *bind* combinator, its *Unit* result will be translated (by function *strip*) to (). Otherwise, if the assignment occurs as last action in a state-transformer, reduction gets stuck, as it would in the λ_σ program.

We now proceed to prove that $\mathcal{E}_\sigma$ is a syntactic embedding.

Definition 6.6 $Env(M)$, the set of *admissible* environments for a λ_σ term M, is the set of all injective mappings from identifiers to identifiers and variables to names that bind all free identifiers and variables in M.

Lemma 6.7 For all terms M, N, environments $\rho \in Env((x.M)\ N)$,

$$\mathcal{F}[\![[N/x]\,M]\!]\rho \;\equiv\; [\mathcal{F}[\![N]\!]\rho/y]\;(\mathcal{F}[\![M]\!]\rho[x \mapsto y]).$$

Proof: A routine induction on the structure of M. ∎

Lemma 6.8 $\mathcal{F}$ is stable under reduction: For all terms $M, N \in \Lambda_\sigma$, environments $\rho \in Env(M)$,

$$\lambda_\sigma \;\vdash\; M \to N \;\Rightarrow\; \lambda\nu \models \mathcal{F}[\![M]\!]\rho \cong \mathcal{F}[\![N]\!]\rho.$$

Proof: We use a case analysis on the form of reduction in $M \to N$. We only show three example cases; the other cases are similar. To simplify notation, we will place a tilde ˜ on identifiers in the target language and write $\tilde{M}$ for

$$\mathcal{F}[\![M]\!]\rho[x \mapsto \tilde{x}]_{x \in fv\ M \setminus dom\ \rho}.$$

For instance, $\lambda\tilde{x}.\tilde{M}$ is short for $\lambda\tilde{x}.\mathcal{F}[\![M]\!]\rho[x \mapsto \tilde{x}]$. Also, we will use infix notation liberally, with 'f' in backquotes denoting the infix version of f.

Case $\triangleright\triangleright$ In this case, the λ_σ reduction is

$$(M \triangleright x.N) \triangleright y.P \;\to\; M \triangleright x.(M \triangleright y.P)$$

and we have in $\lambda\nu$:

$$\begin{aligned}
& \mathcal{F}[\![(M \triangleright x.N) \triangleright y.P]\!]\rho \\
\equiv\ & \text{(by definition of } \mathcal{F}) \\
& (\tilde{M}\ \text{`}\mathit{bind}\text{`}\ \lambda\tilde{x}.\tilde{N})\ \text{`}\mathit{bind}\text{`}\ \lambda\tilde{y}.\tilde{P} \\
=\ & \text{(by definition of } \mathit{bind}, \beta) \\
& \lambda k.(\tilde{M}\ \text{`}\mathit{bind}\text{`}\ \lambda\tilde{x}.\tilde{N})\ (\lambda\tilde{y}.\tilde{P}\ (\mathit{strip}\ \tilde{y})\ k) \\
=\ & \text{(by definition of } \mathit{bind}, \beta) \\
& \lambda k.(\lambda k'.\tilde{M}\ (\lambda\tilde{x}.\tilde{N}\ (\mathit{strip}\ \tilde{x})\ k'))\ (\lambda\tilde{y}.\tilde{P}\ (\mathit{strip}\ \tilde{y})\ k) \\
=\ & \text{(by } \beta\text{-reduction)} \\
& \lambda k.\tilde{M}\ (\lambda\tilde{x}.\tilde{N}\ (\mathit{strip}\ \tilde{x})\ (\lambda\tilde{y}.\tilde{P}\ (\mathit{strip}\ \tilde{y})\ k)) \\
=\ & \text{(by definition of } \mathit{bind}, \beta) \\
& \tilde{M}\ \text{`}\mathit{bind}\text{`}\ (\lambda\tilde{x}.\tilde{N}\ \text{`}\mathit{bind}\text{`}\ \lambda\tilde{y}.\tilde{P}) \\
\equiv\ & \text{(by definition of } \mathcal{F}) \\
& \mathcal{F}[\![M \triangleright x.(N \triangleright y.P)]\!]\rho
\end{aligned}$$

Case $\sigma_{=:}$ In this case, the λ_σ reduction is

$$\sigma +\!\!+ [v\!:\!N'] +\!\!+ \sigma' \cdot N =: v\ ;\ M \;\to\; \sigma +\!\!+ [v\!:\!N] +\!\!+ \sigma' \cdot M$$

Let $L = [n_1, \ldots, n_m] = [\rho v \mid v \leftarrow dom\ \sigma]$, We abbreviate $\nu n_1.\ \ldots\ \nu n_m.M$ by $\nu L.M$. Let $k = \lambda x.\lambda s.(x, s)$. Let N^* be *Undef* if $N \equiv$ **undef**, and *Def* N otherwise. Then we have in $\lambda\nu$:

$$\begin{aligned}
& \mathcal{F}[\![\sigma +\!\!+ [v\!:\!N'] +\!\!+ \sigma' \cdot N =: v\ ;\ M]\!]\rho \\
\equiv\ & \text{(by definition of } \mathcal{F}) \\
& \nu L.\mathit{block}\ ((\mathit{assign}\ \tilde{N}\ \tilde{v}\ \text{`}\mathit{bind}\text{`}\ \lambda().\tilde{M}) \\
& \qquad (\mathcal{F}[\![\sigma +\!\!+ [v\!:\!N'] +\!\!+ \sigma']\!]\rho)) \\
\cong\ & \text{(by definition of } \mathit{block}, \mathit{upd}) \\
& \nu L.\mathit{unwrap}\ ((\mathit{assign}\ \tilde{N}\ \tilde{v}\ \text{`}\mathit{bind}\text{`}\ \lambda().\tilde{M})\ k \\
& \qquad (\mathit{upd}\ N^*\ \tilde{v}\ (\mathcal{F}[\![\sigma +\!\!+ \sigma']\!]\rho))) \\
=\ & \text{(by definition of } \mathit{bind}) \\
& \nu L.\mathit{unwrap}\ (\mathit{assign}\ \tilde{N}\ \tilde{v}\ (\lambda x.(\lambda().\tilde{M})\ (\mathit{strip}\ x)\ k) \\
& \qquad (\mathit{upd}\ N^*\ \tilde{v}\ (\mathcal{F}[\![\sigma +\!\!+ \sigma']\!]\rho))) \\
=\ & \text{(by definition of } \mathit{assign}) \\
& \nu L.\mathit{unwrap}\ ((\lambda x.(\lambda().\tilde{M})\ (\mathit{strip}\ x)\ k)\ \mathit{Unit} \\
& \qquad (\mathit{upd}\ (\mathit{Def}\ \tilde{N})\ \tilde{v}\ (\mathit{upd}\ N^*\ \tilde{v}\ (\mathcal{F}[\![\sigma +\!\!+ \sigma']\!]\rho)))) \\
\cong\ & \text{(by } \beta \text{ reduction (twice), definition of } \mathit{strip}) \\
& \nu L.\mathit{unwrap}\ (\tilde{M}\ k \\
& \qquad (\mathit{upd}\ (\mathit{Def}\ \tilde{N})\ \tilde{v}\ (\mathit{upd}\ N^*\ \tilde{v}\ (\mathcal{F}[\![\sigma +\!\!+ \sigma']\!]\rho)))) \\
\cong\ & \text{(by } \beta \text{ reduction, definition of } \mathit{upd}) \\
& \nu L.\mathit{unwrap}\ (\tilde{M}\ k \\
& \qquad (\mathit{upd}\ (\mathit{Def}\ \tilde{N})\ \tilde{v}\ \mathcal{F}[\![\sigma +\!\!+ \sigma']\!]\rho)) \\
\cong\ & \text{(by definition of } \mathit{block}, \mathit{upd}) \\
& \nu L.\mathit{block}\ (\tilde{M}\ k \\
& \qquad (\mathcal{F}[\![\sigma +\!\!+ [v\!:\!N] +\!\!+ \sigma']\!]\rho)) \\
\equiv\ & \text{(by definition of } \mathcal{F}) \\
& \mathcal{F}[\![\sigma +\!\!+ [v\!:\!N] +\!\!+ \sigma' \cdot M]\!]\rho
\end{aligned}$$

Case σ_{p_λ} In this case the λ_σ reduction is

$$\sigma \cdot \mathbf{return}\ x.M \;\to\; x.(\sigma \cdot \mathbf{return}\ M)$$

Define k, L and $\nu L.M$ as above. Then we have in $\lambda\nu$:

$$\begin{array}{ll} & \mathcal{F}[\![\sigma \cdot \mathbf{return}\, x.M]\!]\rho \\ = & \text{(by definition of } \mathcal{F}, \textit{block}) \\ & \nu L.\mathit{unwrap}\ (\mathit{return}\ (\lambda\tilde{x}.\tilde{M})\ k\ \tilde{\sigma}) \\ = & \text{(by definition of } \mathit{return}) \\ & \nu L.\mathit{unwrap}\ (k\ (\mathit{Res}\ (\lambda\tilde{x}.\tilde{M}))\ \tilde{\sigma}) \\ = & \text{(by definition of } k, \beta\text{-reduction)} \\ & \nu L.\mathit{unwrap}\ (\mathit{Res}\ (\lambda\tilde{x}.\tilde{M}), \tilde{\sigma}) \\ = & \text{(by definition of } \mathit{unwrap} \text{ and } \beta) \\ & \nu L.\lambda\tilde{x}.\mathit{unwrap}\ (\mathit{Res}\ \tilde{M}, \tilde{\sigma}) \\ = & \text{(by reduction in } \lambda\nu) \\ & \lambda\tilde{x}.\nu L.\mathit{unwrap}\ (\mathit{Res}\ \tilde{M}, \tilde{\sigma}) \\ = & \text{(by definition of } \mathit{unwrap}, k, \mathit{return}, \text{ and } \beta) \\ & \lambda\tilde{x}.\nu L.\mathit{unwrap}\ (\mathit{return}\ \tilde{M}\ k\ \tilde{\sigma}) \\ = & \text{(by definition of } \mathcal{F}, \textit{block}) \\ & \mathcal{F}[\![x.(\sigma \cdot \mathbf{return}\, M)]\!]\rho \end{array}$$

■

Lemma 6.8 shows that equal λ_σ terms get mapped to equal (wrt $\cong$) $\lambda\nu$ terms. We also have to show the converse, that different λ_σ terms get mapped to different $\lambda\nu$ terms. To this purpose, we define in Figure 8 a left inverse $\mathcal{F}^{-1}$ of $\mathcal{F}$ and show that $\mathcal{F}^{-1}$ is stable under standard reduction in $\lambda\nu$.

To define $\mathcal{F}^{-1}$, we will assume that all auxiliary functions in Figure 7 are primitive functions (in all their arguments). This is admissible since all these functions are strict. Hence, we invalidate no operational equivalences, nor do we create new ones. We assume that these primitive functions define the following δ reductions:

- $\mathit{block}\ p\ s$, $\mathit{unwrap}\ (a, s)$, $\mathit{bind}\ p\ q\ k\ s$, and $\mathit{newref}\ n\ p\ k\ s$ all reduce in one step to their right hand sides given in Figure 7.
- $\mathit{return}\ a\ (\lambda x.q)\ s$ reduces in one step to $[\mathit{Res}\ a/x]q\ s$.
- $\mathit{deref}\ n\ (\lambda x.q)\ s$ reduces in one step to $[\mathit{Res}\ a/x]q\ s$ if $s\ n$ is of the form $\mathit{Def}\ a$, and gets stuck otherwise.
- $\mathit{assign}\ a\ n\ (\lambda x.q)\ s$ reduces in one step to $[\mathit{Res}\ a/x]q\ (\mathit{upd}\ (\mathit{Def}\ a)\ n\ s)$ if if $s\ n$ is defined, and gets stuck otherwise.
- $\mathit{return}\ a\ k\ s$, $\mathit{deref}\ n\ k\ s$, and $\mathit{assign}\ a\ n\ k\ s$ get stuck if k is not a λ-abstraction.

Definition 6.9 Let ρ^{-1} be the inverse of environment ρ. ρ^{-1} is a well-defined (partial) function since ρ is injective. Define $\mathit{Env}^{-1}(M) = \{\rho^{-1} \mid \rho \in \mathit{Env}(M)\}$.

The proofs of the following three lemmas are all easy inductions on the structure of M:

Lemma 6.10 $\mathcal{F}^{-1}[\![\ ]\!]\rho^{-1}$ is a left inverse of $\mathcal{F}[\![\ ]\!]\rho$: For all $M \in \Lambda_\sigma$, $\rho \in \mathit{Env}(M)$,

$$\mathcal{F}^{-1}[\![\mathcal{F}^{-1}[\![M]\!]\rho]\!]\rho^{-1} \equiv M.$$

Lemma 6.11 For all terms $M \in \Lambda\nu$, environments $\rho \in \mathit{Env}^{-1}(M)$, identifiers $x \notin \mathit{fv}\ M$, y,

$$\mathcal{F}^{-1}[\![M]\!]\rho[x \mapsto y] \equiv \mathcal{F}^{-1}[\![M]\!]\rho$$

provided either side is defined.

Lemma 6.12 For all terms $M, N \in \Lambda\nu$, environments $\rho \in \mathit{Env}^{-1}((\lambda x.M)\ N)$,

$$\mathcal{F}^{-1}[\![[N/x]\ M]\!]\rho \equiv [\mathcal{F}^{-1}[\![N]\!]\rho/y]\ (\mathcal{F}^{-1}[\![M]\!]\rho[x \mapsto y]).$$

provided either side is defined.

Lemma 6.13 $\mathcal{F}^{-1}$ is stable under standard reduction: For all terms $M, N \in \Lambda\nu$, answers A, environments $\rho \in \mathit{Env}(M)$, if

$$\lambda\nu \vdash M \underset{s}{\rightarrow} N \underset{s}{\twoheadrightarrow} A$$

and $\mathcal{F}^{-1}[\![M]\!]\rho$ is defined, then so is $\mathcal{F}^{-1}[\![N]\!]\rho$ and

$$\lambda_\sigma \models \mathcal{F}^{-1}[\![M]\!]\rho \cong \mathcal{F}^{-1}[\![N]\!]\rho.$$

Proof: The proof is by a case analysis on the form of the redex Δ in $M \rightarrow N$. We only show one case; the other cases are all similar, but tend to be simpler.

Case $\Delta = \mathit{assign}\ a\ n\ k\ s$

Since the reduction appears in a standard reduction sequence to an answer, Δ must occur in a superterm of the form $\mathit{unwrap}\ \Delta$, k must be a continuation of the form $\lambda x.q\ (\mathit{strip}\ x)\ k'$ and s must be state, otherwise reduction would get stuck.

In $\lambda\nu$ the reduct R of Δ is

$$[()/x]q\ k'\ (\mathit{upd}\ (\mathit{Def}\ x)\ n\ s),$$

and we have to show that, for any $\rho \in \mathit{Env}^{-1}(\Delta)$,

$$\lambda_\sigma \models \mathcal{F}^{-1}[\![\mathit{unwrap}\ \Delta]\!]\rho \cong \mathcal{F}^{-1}[\![\mathit{unwrap}\ R]\!]\rho.$$

The left hand side of this operational equivalence expands to

$$\mathcal{S}^{-1}[\![s]\!]\rho\ \cdot\ \mathcal{P}^{-1}[\![\mathit{assign}\ a\ n\ k]\!]\rho. \tag{1}$$

$$
\begin{array}{lcll}
\mathcal{F}^{-1}[\![f]\!]\rho & = & f & \\
\mathcal{F}^{-1}[\![c^n]\!]\rho & = & c^n & \\
\mathcal{F}^{-1}[\![x]\!]\rho & = & \rho x & \\
\mathcal{F}^{-1}[\![t]\!]\rho & = & \rho t & \\
\mathcal{F}^{-1}[\![\lambda x.M]\!]\rho & = & y.\mathcal{F}^{-1}[\![M]\!]\rho[x \mapsto y] & (y \notin \mathit{range}\ \rho) \\
\mathcal{F}^{-1}[\![MN]\!]\rho & = & (\mathcal{F}^{-1}[\![M]\!]\rho)\ (\mathcal{F}^{-1}[\![N]\!]\rho) & \\
\mathcal{F}^{-1}[\![\nu t.M]\!]\rho & = & \mathcal{F}^{-1}[\![M]\!]\rho[t \mapsto v] & (v \notin \mathit{range}\ \rho) \\
\mathcal{F}^{-1}[\![\mathit{newref}\ t\ M]\!]\rho & = & \mathbf{var}\ \rho t.\mathcal{F}^{-1}[\![M]\!]\rho & \\
\mathcal{F}^{-1}[\![\mathit{deref}\ M]\!]\rho & = & (\mathcal{F}^{-1}[\![M]\!]\rho)? & \\
\mathcal{F}^{-1}[\![\mathit{assign}\ M\ N]\!]\rho & = & \mathcal{F}^{-1}[\![M]\!]\rho =: \mathcal{F}^{-1}[\![N]\!]\rho & \\
\mathcal{F}^{-1}[\![\mathit{return}\ M]\!]\rho & = & \mathbf{return}\ (\mathcal{F}^{-1}[\![M]\!]\rho) & \\
\mathcal{F}^{-1}[\![\mathit{bind}\ M\ N]\!]\rho & = & \mathcal{F}^{-1}[\![M]\!]\rho \triangleright \mathcal{F}^{-1}[\![N]\!]\rho & \\
\mathcal{F}^{-1}[\![\mathit{pure}\ M]\!]\rho & = & \mathbf{pure}\ (\mathcal{F}^{-1}[\![M]\!]\rho) & \\
\mathcal{F}^{-1}[\![\mathit{unwrap}\ (\mathit{Res}\ a, s)]\!]\rho & = & \mathcal{S}^{-1}[\![s]\!]\rho\ \cdot\ \mathbf{return}\ (\mathcal{F}^{-1}[\![a]\!]\rho) & \\
\mathcal{F}^{-1}[\![\mathit{unwrap}\ (p\ s)]\!] & = & \mathcal{S}^{-1}[\![s]\!]\rho\ \cdot\ \mathcal{P}^{-1}[\![p]\!]\rho &
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{S}^{-1}[\![\bot]\!]\rho & = & \emptyset & \\
\mathcal{S}^{-1}[\![\mathit{upd}\ \mathit{Undef}\ t\ s]\!]\rho & = & \mathit{UPD}\ \mathbf{undef}\ (\rho t)\ (\mathcal{S}^{-1}[\![s]\!]\rho) & \\
\mathcal{S}^{-1}[\![\mathit{upd}\ (\mathit{Def}\ a)\ t\ s]\!]\rho & = & \mathit{UPD}\ (\mathcal{F}^{-1}[\![a]\!]\rho)\ (\rho t)\ (\mathcal{S}^{-1}[\![s]\!]\rho) & \\
 & & \mathbf{where}\ \mathit{UPD}\ a\ v\ [\,] \ =\ [v : a] & \\
 & & \qquad \mathit{UPD}\ a\ v\ ([v : b] +\!\!+ \sigma) \ =\ [v : a] +\!\!+ \sigma & \\
 & & \qquad \mathit{UPD}\ a\ v\ ([w : b] +\!\!+ \sigma) \ =\ [w : b] +\!\!+ \mathit{UPD}\ a\ v\ \sigma & (v \neq w)
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{P}^{-1}[\![p\ (\lambda x.\lambda s.(x, s)]\!]\rho & = & \mathcal{F}^{-1}[\![p]\!]\rho & \\
\mathcal{P}^{-1}[\![p\ (\lambda x.q\ (\mathit{strip}\ x)\ k)]\!] & = & \mathcal{F}^{-1}[\![p]\!]\rho \triangleright \lambda y.\mathcal{P}^{-1}[\![q\ k]\!]\rho[x \mapsto y] & (y \notin \mathit{range}\ \rho)
\end{array}
$$

Figure 8: Reverse mapping $\mathcal{F}^{-1}$

We first turn to the state portion of this term. By the definition of $\mathcal{F}$, there are terms $a_1,\dots,a_m$ and names $n_1,\dots,n_m$ such that

$$s \equiv upd\ a_m\ n_m\ (\ \dots\ (upd\ a_1\ n_1\ \bot)\ \dots\ ).$$

Then by the definition of $\mathcal{S}^{-1}$,

$$\mathcal{S}^{-1}[\![s]\!]\rho \equiv [\rho n_1 : \mathcal{F}^{-1}[\![a_1]\!]\rho,\ \dots\ , \rho n_m : \mathcal{F}^{-1}[\![a_m]\!]\rho].$$

Since *assign a n k* is by assumption a redex, n is in the domain of s, hence is equal to at least one of $n_1,\dots,n_m$. Therefore, there is a term M and states σ, σ' such that

$$\mathcal{S}^{-1}[\![s]\!]\rho \equiv \sigma +\!\!+ [\rho n : M] +\!\!+ \sigma'. \qquad (2)$$

We now turn to the procedure part of (1). Assume first that x occurs free in the prefix q of the continuation $k = \lambda x.q\ (strip\ x)\ k'$ (an easy induction on the form of state-transformers shows that x cannot appear free in k'). Then we have in λ_σ:

$$
\begin{aligned}
& \mathcal{S}^{-1}[\![s]\!]\rho \ \cdot\ \mathcal{P}^{-1}[\![assign\ a\ n\ (\lambda x.q\ (strip\ x)\ k')]\!]\rho \\
\equiv\ & \text{(by (2) and the definition of } \mathcal{P}^{-1}) \\
& \sigma +\!\!+ [\rho n : M] +\!\!+ \sigma' \cdot \\
& \qquad \mathcal{F}^{-1}[\![a]\!]\rho =: \rho n \rhd \lambda y.\mathcal{P}^{-1}[\![q\ k']\!]\rho[x \mapsto y] \\
=\ & \text{(by } =:\rhd \text{ and } \beta \text{ reduction)} \\
& \sigma +\!\!+ [\rho n : M] +\!\!+ \sigma' \cdot \\
& \qquad \mathcal{F}^{-1}[\![a]\!]\rho =: \rho n\ ;\ [()/y]\ (\mathcal{P}^{-1}[\![q\ k']\!]\rho[x \mapsto y]) \\
\equiv\ & \text{(by Lemma 6.12)} \\
& \sigma +\!\!+ [\rho n : M] +\!\!+ \sigma' \cdot \\
& \qquad \mathcal{F}^{-1}[\![a]\!]\rho =: \rho n\ ;\ (\mathcal{P}^{-1}[\![[()/x](q\ k')]\!]\rho[x \mapsto y]) \\
\equiv\ & \text{(by Lemma 6.11)} \\
& \sigma +\!\!+ [\rho n : M] +\!\!+ \sigma' \cdot \\
& \qquad \mathcal{F}^{-1}[\![a]\!]\rho =: \rho n\ ;\ (\mathcal{P}^{-1}[\![[()/x]\ (q\ k')]\!]\rho) \\
=\ & \text{(by } \sigma_{=:} \text{ reduction)} \\
& \sigma +\!\!+ [\rho n : \mathcal{F}^{-1}[\![a]\!]\rho] +\!\!+ \sigma' \cdot \\
& \qquad \mathcal{P}^{-1}[\![[()/x]\ (q\ k')]\!]\rho \\
\equiv\ & \text{(since } x \text{ is not free in } k') \\
& \sigma +\!\!+ [\rho n : \mathcal{F}^{-1}[\![a]\!]\rho] +\!\!+ \sigma' \cdot \\
& \qquad \mathcal{P}^{-1}[\![[()/x]q\ k']\!]\rho \\
\equiv\ & \dots
\end{aligned}
$$

Here, k' is a continuation, i.e. it is of one of the forms $\lambda y.q'\ (strip\ y)\ k''$, $\lambda x.\lambda s.(x, s)$. If k' is of the form $\lambda y.q'\ (strip\ y)\ k''$, this can be continued as follows:

$$
\begin{aligned}
& \dots \\
\equiv\ & \sigma +\!\!+ [\rho n : \mathcal{F}^{-1}[\![a]\!]\rho] +\!\!+ \sigma' \cdot \\
& \qquad \mathcal{P}^{-1}[\![[()/x]q\ (\lambda y.q'\ (strip\ y)\ k'')]\!]\rho \\
\equiv\ & \text{(by definition of } \mathcal{P}^{-1}) \\
& \sigma +\!\!+ [\rho n : \mathcal{F}^{-1}[\![a]\!]\rho] +\!\!+ \sigma' \cdot \\
& \qquad \mathcal{F}^{-1}[\![[()/x]q]\!]\rho \rhd \lambda z.\mathcal{P}^{-1}[\![q'\ k'']\!]\rho[y \mapsto z] \\
\cong\ & \text{(by definition of } UPD) \\
& UPD\ (\mathcal{F}^{-1}[\![a]\!]\rho)\ n\ (\mathcal{S}^{-1}s\rho) \cdot \\
& \qquad \mathcal{F}^{-1}[\![[()/x]q]\!]\rho \rhd \lambda z.\mathcal{P}^{-1}[\![q'\ k'']\!]\rho[y \mapsto z] \\
\equiv\ & \text{(by definition of } \mathcal{F}^{-1}, \mathcal{S}^{-1}, \mathcal{P}^{-1}) \\
& \mathcal{F}^{-1}[\![unwrap\ ([()/x]\ q) \\
& \qquad (\lambda y.q'\ (strip\ y)\ k'')\ (upd\ (Def\ a)\ n\ s))]\!]\rho \\
\equiv\ & \mathcal{F}^{-1}sqR\rho
\end{aligned}
$$

If, on the other hand, the continuation k' is of the form $\lambda x.\lambda y.(x, s)$, (...) can be continued as follows:

$$
\begin{aligned}
& \dots \\
\equiv\ & \sigma +\!\!+ [\rho n : \mathcal{F}^{-1}[\![a]\!]\rho] +\!\!+ \sigma' \cdot \\
& \qquad \mathcal{P}^{-1}[\![[()/x]q\ (\lambda x.\lambda s.(x, s))]\!]\rho \\
\equiv\ & \text{(by definition of } \mathcal{P}^{-1}) \\
& \sigma +\!\!+ [\rho n : \mathcal{F}^{-1}[\![a]\!]\rho] +\!\!+ \sigma' \cdot \\
& \qquad \mathcal{F}^{-1}[\![[()/x]q]\!]\rho \\
\equiv\ & \text{(by definition of } \mathcal{F}^{-1}, \mathcal{S}^{-1}) \\
& \mathcal{F}^{-1}[\![unwrap\ ([()/x]q) \\
& \qquad (\lambda x.\lambda s.(x, s))\ (upd\ (Def\ a)\ n\ s))]\!]\rho \\
\equiv\ & \mathcal{F}^{-1}[\![R]\!]\rho
\end{aligned}
$$

This proves the case where x occurs free in q. The case where x does not occur free in q is identical, except that the $=:\rhd$ reduction step is left out.

■

Proposition 6.14 $\mathcal{E}_\sigma$ is a syntactic embedding of λ_σ in λ.

Proof: We first show that $\mathcal{E}_\sigma$ preserves λ-programs. Let M be a closed λ-term. Then by an easy induction on the structure of M, $\mathcal{F}[\![M]\!]\bot \equiv M$. Since $\mathcal{E}_\nu$ is a syntactic embedding, $\mathcal{E}_\nu[\![M]\!] \equiv M$. Hence, $\mathcal{E}_\sigma[\![M]\!] \equiv M$.

We now show that $\mathcal{E}_\sigma$ preserves semantics. Since $\mathcal{E}_\nu$ is a syntactic embedding this follows from

$$\lambda_\sigma \vdash M = A \ \Leftrightarrow\ \lambda\nu \vdash \mathcal{F}[\![M]\!]\bot = A \qquad (3)$$

for all terms $M \in \Lambda_\sigma$, answers A.

We show each direction of "$\Leftrightarrow$" in (3) separately. "$\Rightarrow$": Assume $M = A$. Then by an induction on the length of reduction from M to A, using Lemma 6.8 at each step, $\mathcal{F}[\![M]\!]\rho \cong \mathcal{F}[\![A]\!]\rho$. But the latter term equals A.

"$\Leftarrow$": Assume $\mathcal{F}[\![M]\!]\bot = A$. Then by an induction on the length of standard reduction from $\mathcal{F}[\![M]\!]\bot$ to A, $\mathcal{F}^{-1}[\![\mathcal{F}[\![M]\!]\bot]\!]\bot \cong \mathcal{F}^{-1}[\![A]\!]$. The right-hand side of this equivalence equals A, whereas by Lemma 6.10 the left hand side equals M. ■

Theorem 6.15 λ_σ is a conservative operational extension of λ: For any two terms $M, N \in \Lambda$,

$$\lambda \models M \cong N \;\Leftrightarrow\; \lambda_\sigma \models M \cong N.$$

Proof: By Proposition 6.14, there is a syntactic embedding of λ_σ in λ. By Theorem 6.4, this implies that λ_σ is an operational extension of λ. It remains to show that the extension is conservative.

Assume $\lambda_\sigma \models M \cong N$. Then we have

$$\lambda_\sigma \;\vdash\; C[M] = A \Leftrightarrow \lambda_\sigma \;\vdash\; C[N] = A$$

for all contexts C in Λ_σ 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 $\lambda \models M \cong N$. ■

Corollary 6.16 λ_{var} is a conservative operational extension of λ: For any two terms $M, N \in \Lambda$,

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

Proof: Immediate from Theorem 6.15 and Theorem 4.16. ■

7 Related Work

Hoare *et. al.* [12] present a normalizing set of equations for an imperative language with assignment, conditional and nondeterministic choice. Functional abstraction is not considered. Field [10] 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 [8, 9] 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[23] 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 [15, 16] 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 [27]. 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).

Recent work by Jon Riecke [25] also addresses the problem of conservative extension of the λ-calculus by a language with effects. His techniques are unlike ours based on the denotational semantics of a typed extension of PCF with parallel or. His method depends on a full abstraction result for the denotational semantics.

A programming language with motivation similar to that of λ_{var} is Forsythe [24]. 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. (a simpler type system is found in [32]). 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 is the study of type systems for λ_{var}. One possible appraoch make λ_{var} to add types to λ_{var} is outlined in a companion report [3]. The

treatment in the present report was untyped in order that many of our results may be applied immediately to versions of λ_{var} with arbitrary 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 besides [3] are [11, 19, 27, 28, 30]).

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

[3] Kung Chen and Martin Odersky. A type system for a lambda calculus with assignment. To appear as a Yale Research Report, May 1993.

[4] William Clinger and Jonathan Rees. Revised[4] report on the algorithmic language Scheme. Distributed electronically, November 1991.

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

[6] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 6, pages 243–320. MIT Press, Cambridge, Massachusetts, 1990.

[7] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

[8] Matthias Felleisen and Daniel 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.

[9] Matthias Felleisen and Robert 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.

[10] John 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.

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

[12] 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.

[13] Paul Hudak and Dan 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.

[14] Paul Hudak and Philip L. Wadler. Report on the programming language Haskell: a non-strict, purely functional language, version 1.0. Technical Report YALEU/DCS/RR-777, Yale University Department of Computer Science, April 1990.

[15] Ian Mason and Carolyn 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.

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

[17] Robin Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[18] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[19] Martin 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.

[20] Martin Odersky. A syntactic method for proving operational equivalences. Research Report YALEU/DCS/RR-964, Department of Compter Science, Yale University, May 1993.

[21] Martin Odersky. A syntactic theory of local names. Technical Report TR-965, Yale University, May 1993.

[22] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, January 10–13, 1993*, pages 71–84. ACM Press, January 1993.

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

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

[25] Jon G. Riecke. Delimiting the scope of effects. In *Functional Programming and Computer Architecture*, June 1993.

[26] David A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, 1985.

[27] Vipin Swarup, Uday S. Reddy, and Evan Ireland. Assignments for applicative languages. In John 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.

[28] 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.

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

[30] Philip 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.

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

[32] Stephen Weeks and Matthias Felleisen. On the orthogonality of assignments and procedures in Algol. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, January 10–13, 1993*, pages 57–70. ACM Press, January 1993.