LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

A Lambda Calculus with Letrecs and Barriers

Computation Structure Group Memo 395
January 10, 1997

Arvind	*MIT Lab for Computer Science*
J-W. Maessen	*MIT Lab for Computer Science*
R.S. Nikhil	*Digital Equipment Corp., Cambridge Research Lab*
J.E. Stoy	*Oxford University Computing Laboratory*

Invited Paper in 16th Foundations of Software and Theoretical Computer Science, Hyderabad, India, Dec. 1996

This research was conducted at the MIT Laboratory for Computer Science. Funding for this project was provided in part by the Advanced Research Projects Agency of the Department of Defense under Ft. Huachuca contract C-DABT63-95-C0150.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

A Lambda Calculus with Letrecs and Barriers

Arvind	*MIT Lab for Computer Science*
J-W. Maessen	*MIT Lab for Computer Science*
R.S. Nikhil	*Digital Equipment Corp., Cambridge Research Lab*
J.E. Stoy	*Oxford University Computing Laboratory*

January 10, 1997

1 Introduction

It is often said that pure functional languages like Haskell[9] are merely "syntactic sugar" for a version of the lambda calculus extended with constants and data structures. In fact the semantics of these languages is more complicated than that, but an understanding of the lambda calculus is nevertheless crucial in their study. The lambda calculus also plays an important role as a vehicle for explaining the semantics of a much broader class of languages such as Lisp[**?**], ML[11], Scheme[8], Id[12] and pH[2] (a parallel dialect of Haskell). The importance of a small, simple, formal system cannot be overstated: at the very least it helps in teaching the language and building intuitions about it; it is also an unambiguous point of appeal for reasoning about the correctness of program equivalences.

A practically important class of program equivalences are the optimizations performed by a compiler. The lambda calculus and its variants are used as the basis for kernel languages in some compilers. The task of designing the required transformations and optimizations is eased by restricting attention to a simpler and more regular language.

In this paper we will describe some extensions which make the pure lambda calculus much more useful, both for discussing the semantic essence of "real" languages, and also as an intermediate language for a compiler.

We will begin by a quick overview of the pure lambda calculus extended with constants. This is followed by a discussion of inadequacy of this system to address some pragmatic aspects of even purely functional languages (Sect. 3). In Sect. 4 we describe λ_{let}, a lambda calculus with recursive let blocks (i.e., letrecs), and discuss why it is more appropriate as a kernel language; we also discuss the complications letrecs add. In Sect. 5, we discuss λ_B which is λ_{let} extended with sequentializing constructs called barriers. We then mention some more extensions we have studied, which continue to be the target of active research.

2 The Pure λ Calculus with Constants

We begin by looking at the pure λ calculus. The calculus itself is very small, yet it is possible to express every sequential computable function as a λ-expression. This power comes from several important notions. First, we can write anonymous (unnamed) functions. Second, the calculus is higher-order. We can thus write functions which take functions as arguments and return functions as results. Finally, λ calculus does not limit *where* we are permitted to perform evaluation. As a result, we can discuss the effects of evaluation order on the calculus.

The material in this section will be very familiar to those who have encountered the λ calculus before. This presentation highlights the aspects of λ calculus which are important to later sections and introduces some notation which will recur throughout the paper.

2.1 Syntax

The λ calculus has by far the simplest syntax of any of the calculi described here (We concern ourselves only with abstract syntax—we use parentheses freely to clarify grouping. We also freely define syntactic categories as subsets or unions of other ones, trusting that this will cause no confusion):

E	$::=$	x	Identifiers
	$\mid$	$\lambda x.E$	Abstractions
	$\mid$	$E\ E$	Applications
	$\mid$	$\texttt{cond}(E,E,E)$	Conditionals
	$\mid$	$\mathrm{PF}_k(E_1,\cdots,E_k)$	Primitive function applications, $k \geq 1$
	$\mid$	CN_0	Constants
	$\mid$	$\mathrm{CN}_k(E_1,\cdots,E_k)$	Constructor applications, $k \geq 1$
PF_1	$::=$	$\texttt{not} \mid \texttt{Proj}_1 \mid \texttt{Proj}_2 \mid \cdots$	Primitive functions of arity 1
PF_2	$::=$	$\texttt{+} \mid \texttt{-} \mid \cdots$	Primitive functions of arity 2
$\vdots$			
CN_0	$::=$	$\mathit{Number} \mid \mathit{Boolean}$	Constructors of arity 1
CN_2	$::=$	$\texttt{cons} \mid \cdots$	Constructors of arity 2
$\vdots$			

2.2 Renaming: An Equivalence Rule

In many calculi there are some choices that, though leading to syntactically different terms, do not lead to terms that differ in any *essential* way. For example, in renaming a bound variable `x` to avoid free-variable capture, it does not really matter whether the fresh name chosen is `y` or `z`, even though the two choices lead to terms that are not syntactically "equal". λ calculus formalizes this notion as α reduction[1]:

$$\lambda x.E \quad \longrightarrow \quad \lambda x'.E[x/x'] \qquad\qquad x' \notin FV(E)$$

Here we read $E[x/x']$ as "substitute x' for for every free occurrence of x in E". Ordinarily, we assume that terms e_1 and e_2 interconvertible using only α renaming are *equivalent*, and write $e_1 \equiv e_2$. A reduction rule (to be introduced later) is applicable to a term if it is applicable to *any* equivalent term.

2.3 Reduction Rules for the λ Calculus

The most critical rule in λ calculus is the β rule:

$(\lambda x.e_1)\ e_2 \longrightarrow e_1'[e_2/x]$
where e_1' is a renaming of e_1 to avoid free variable capture

[1] As usual, the notation $e \longrightarrow e_1$ means that e reduces to e_1 by a single application of a reduction rule to e or to one of its subterms, and $e \twoheadrightarrow e_1$ means that e reduces to e_1 by a finite sequence (zero or more) of such steps. A term or subterm which is capable of reduction by one of the rules is called a *redex*.

Any expression to which the β rule can be applied is referred to as a β redex. Note that redexes can occur anywhere in a λ expression; for example, in $\lambda x.(\underline{(\lambda y.\underline{(\lambda z.z)\ y})\ x})\ x$ both the underlined subexpressions are β redexes.

δ rules specify the behavior of each primitive function. They are all of the form:

$$\mathrm{PF}_k(v_1,\cdots,v_k) \quad\longrightarrow\quad v \qquad\qquad (\delta)$$

Obviously we cannot list all the δ rules here, but an example is:

$$\texttt{+(}\underline{\texttt{n}}\texttt{, }\underline{\texttt{m}}\texttt{)} \quad\longrightarrow\quad \underline{\texttt{p}} \qquad \text{where } \underline{\texttt{n}}, \underline{\texttt{m}} \text{ and } \underline{\texttt{p}} \text{ represent numbers and } p = n + m$$

This rule should be read as saying: the term consisting of "+" applied to two numerical constants may be rewritten as the constant that represents the sum of those two numbers.

Projection functions select fields of constructed values:

$$\texttt{proj}_j\texttt{(}\mathrm{CN}_k\texttt{(}e_1\texttt{, }\cdots\texttt{, }e_k\texttt{))} \quad\longrightarrow\quad e_j \qquad 1 \leq j \leq k \qquad\qquad (\mathrm{Proj})$$

Cond rules specify the behavior of conditional expressions:

$$\begin{array}{lcll} \texttt{cond(True, }e_1\texttt{, }e_2\texttt{)} & \longrightarrow & e_1 & \qquad (\mathrm{CondT}) \\ \texttt{cond(False, }e_1\texttt{, }e_2\texttt{)} & \longrightarrow & e_2 & \qquad (\mathrm{CondF}) \end{array}$$

2.4 Recursion

One thing λ calculus lacks is direct provision for recursion. For example, we cannot write the factorial function directly, as we might in an ordinary programming language:

$$\texttt{fact} \quad = \quad \lambda n.\texttt{cond(}n = 0\texttt{, 1, }n{*}\texttt{(fact (}n-1\texttt{)))}$$

The problem is that `fact` isn't well defined on the right hand side of the above equation. Instead, we have to abstract away the recursive call to `fact`:

$$\begin{array}{lcl} \texttt{fact} & = & \texttt{fact}'\ \texttt{fact} \\ \texttt{fact}' & = & \lambda f.\lambda n.\texttt{cond(}n = 0\texttt{, 1, }n{*}\texttt{(}f\texttt{ (}n-1\texttt{)))} \end{array}$$

Now the `fact` function we desire is the fixed point of `fact`$'$, so we can write `fact` using the fixed-point combinator Y:

$$\begin{array}{lcl} \texttt{fact} & = & Y\ \texttt{fact}' \\ Y & = & \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x)) \end{array}$$

2.5 Confluence and Equivalence

Because λ expressions can contain many redexes, the most obvious question to be asked is: Is the λ calculus confluent? That is, if we reduce different redexes, can we eventually bring the resulting terms back together? The Church-Rosser Theorem says that we can:

> The λ calculus is confluent: If $e_1 \twoheadrightarrow e_2$ and $e_1 \twoheadrightarrow e_3$, then there exists an e_4 such that $e_2 \twoheadrightarrow e_4$ and $e_3 \twoheadrightarrow e_4$.

Unfortunately, confluence does not guarantee that we *will* always reach e_4 when we reduce further; it merely states that it is possible to do so. Subterms may be infinitely reducible, distracting us from useful reduction elsewhere.

2.5.1 Normal Forms of λ Expressions

In functional languages like the λ calculus, a program itself represents the "answer" or "output" of computation. The process of reduction gradually transforms the initial program into a form that more manifestly represents an answer (e.g., the reduction of "2+3" to "5"). Note that the answer may even be infinite, such as an infinite list of actions for the operating system to perform. At some intermediate point of computation, it is possible that the answer has thus far been only partially manifested.

There are actually many possible definitions of "answer" even for the pure λ calculus. The most obvious definition is *normal form*—no further redexes exist anywhere in the expression. Because of confluence, we know that the normal form of any expression is unique (otherwise further reductions could be performed to bring the distinct terms together!). In addition, a normal-order reduction (where the leftmost outermost redex is reduced at every step) is guaranteed to find the normal form of a term if it exists.

However, many terms do not have a normal form at all! One infamous example is the term Ω, which reduces to itself:

$$\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x) \longrightarrow (\lambda x.x\ x)\ (\lambda x.x\ x) \longrightarrow \ldots$$

Thus, while normal form seems to provide a convenient notion of equality, it leaves open the question of how to treat terms which do not possess a normal form.

More commonly used is weak head normal form (WHNF), which can be defined syntactically (using the category A for "answer") as follows:

A	$::=$	V	Values
	$\mid$	P	Irreducible expressions
V	$::=$	$\lambda x.E$	Abstractions
	$\mid$	CN_0	Constants
	$\mid$	$\mathrm{CN}_k(E_1,\cdots E_k)$	Constructed values, $k \geq 1$
P	$::=$	x	Free variables
	$\mid$	$P\ E$	Applications of irreducibles

Values, V, play a prominent role in the reduction rules of our later calculi.

Unfortunately, weak head normal forms are *not* unique. For example, the following terms have the same normal form but are both already in WHNF:

$$\lambda x.(\lambda y.\lambda z.y)\ x\ x \qquad \text{and} \qquad \lambda x.(\lambda y.y)\ x$$

We thus still need to define equivalence between terms.

3 Problems with the λ Calculus

There are a number of problems with the λ calculus that prevent it from closely mirroring the operation of an actual programming language implementation—even a purely functional language such as Haskell. While the existence of the Y combinator is mathematically fascinating, fixed points do not provide a simple encapsulation of recursion. For example, we would like to be able to declare mutually recursive functions and data structures in such a way that their definitions are clear and readable; the need to re-shape such definitions as fixed points plays havoc with such an endeavor.

More problematic is that the λ calculus handles sharing very poorly indeed. The problem is in β reduction itself. In the application $(\lambda x.e_1)\ e_2$ the traditional β rule makes copies of the argument expression e_2 for each occurrence of the formal parameter x.

In an effort to solve this problem, several researchers have added let blocks to the language and replaced the β rule with β_G[4, 10, 1, 3]:

$$(\lambda x.e_1)\ e_2 \longrightarrow \{\ x' = e_2 \text{ in } e_1[x'/x]\ \} \qquad (\beta_G)$$

where x' is a variable which does not otherwise occur.

A central feature of the β_G rule is that it performs no substitution at all; instead we add new rules for *instantiating the values of variables.* These new rules can be designed so that sharing of subexpressions is preserved. This also means that the β_G rule never *discards* the argument. This will affect the semantics of barriers and side-effects. In short, the β_G rule preserves the "graph structure" of the term, with shared subterms, instead of the "tree structure" of the traditional λ calculus, with replicated subterms. This is why we use the subscript G, for "graph." In the next section we will use the β_G rule as the core of a new calculus, λ_{let}.

The idea of sharing subexpressions to make normal-order reduction efficient was first tackled by Wadsworth in his D.Phil. Thesis as far back as 1971 [13]. In his seminal work, which came to be known as *graph reduction*, he used an explicit graph notation instead of a textual term notation. Since the source language was standard λ-calculus, there was no issue of recursive or cyclic terms—the *sharing* he obtained was just a way to avoid duplicating the function argument when the formal parameter was used multiple times in the λ body. Since then several different let-block extensions to the λ calculus have been presented in the literature to model sharing. Recently, Launchbury reformulated Wadsworth's system without using a graph notation by introducing "let" expressions into the term syntax [10]. Again, since the source language was standard λ calculus, there was no issue of recursive or cyclic terms. Launchbury did not separate *calculus* from *strategy*—he directly gave a set of rules to implement lazy evaluation.

In [3], Ariola et al. presented a calculus which did allow the definition of cyclic terms, but it was restricted to disallow substitution of a let-bound identifier in the right-hand side of its own binding, thus side-stepping a full treatment of cyclic terms. This work also did not address data structures.

In [1], Abadi et al. also presented a calculus that dealt with cyclic terms, but it deviated quite far from traditional λ-calculus in that it used de Bruijn notation, introduced *environments* and a calculus of environments, etc.

The most recent work by Ariola [5] is the first to introduce a calculus of recursive let-expressions. Our λ_{let} is essentially the same as Ariola's system, with some small differences in the details of the rules of the calculus.

4 λ_{let}: a λ Calculus with Letrec Blocks

λ_{let} extends the λ calculus with "letrec blocks", allowing us to define shared recursive bindings by binding an expression to an identifier and then using that identifier in a number of places. Our plan of action for this section is as follows. First we present the λ_{let} *calculus*, i.e., the syntax of λ_{let} terms and the reduction rules that allow us to transform terms and perform "computation". We then define a notion of printing, which captures the "important" information in a term. Finally, we discuss how we might define a standard reduction strategy which evaluates let bindings in parallel.

4.1 Syntax

λ_{let} augments the syntax of λ calculus with letrec blocks:

E	$::=$	$\cdots$	as before
	$\mid$	$\{\ S$ `in` $E\ \}$	Letrec blocks
S	$::=$	ϵ	Empty statements
	$\mid$	$x = E$	Bindings
	$\mid$	$S\ ;\ S$	Parallel

The S in a Letrec block is a composition of parallel statements, at the leaves of which are bindings and empty statements. The identifiers on the left-hand sides of all these bindings must be pair-wise distinct. The empty statement is used only as a technical convenience in λ_{let}; it will become useful in λ_B. We refer to the final expression in a block as the *return expression* of the block.

λ_{let} has *"letrec"* scope rules—any identifier on the left-hand side of any binding in a block may be referred to in any right-hand side, as well as in the return expression of the block. In other words, the collection of bindings in a block should be regarded as simultaneous, recursive bindings.

In λ_{let} we ensure that constructors are evaluated exactly once, after which they become "values" and are written using an underline. The complete syntactic category of expressions therefore contains these terms too:

E	$::=$	$\cdots$	
	$\mid$	$\underline{\mathrm{CN}_k}(E_1,\cdots,E_k)$	Constructor values, $k \geq 1$

Note, however, that the underlined versions of constructors never occur in unevaluated λ_{let} terms. For the rest of this section, we will simply appeal to the reader's intuition on the distinction between constructors with and without underlines.

4.1.1 Values and Simple Expressions

In our calculi, as in the λ calculus, we need to substitute various *uses* of an identifier with the identifier's definition. This is called *instantiation*. The way this operation is defined can have an impact on sharing. Suppose we have the following term:

```
(1)        {  x = (f a);
(2)           y = x;
           in
              w }
```

Suppose we substitute the use of `x` on line (2) by its definition from line (1):

```
(1)        {  x = (f a);
(2)           y = (f a);
           in
              w }
```

We now have two copies of "`f a`," which may be an arbitrarily complicated computation, i.e., we have replicated the computation, instead of *sharing* it. In versions of the calculus that include barriers and side-effects, this sharing is required and not merely an issue of efficiency.

Thus, our instantiation rules must be very careful about which expressions can be substituted. To maintain sharing, we identify two subsets of λ_{let} terms that are substitutable—*values* and *identifiers*, collectively known as *simple expressions*:

SE	$::=$	x	Identifiers
	$\mid$	V	Values

4.2 λ_{let} Equivalence Rules

The first two equivalence rules are analogous to α renaming; they express the idea that the particular choice of a name for a bound variable is not important:

$$\begin{array}{lcl} \lambda x.e & \equiv & \lambda x'.(e[x'/x]) \\ \{\ SC[x = e]_S \text{ in } e_0\ \} & \equiv & \{\ SC[x' = e]_S \text{ in } e_0\ \}[x'/x] \end{array}$$

where x' does not otherwise occur in the program.

The second set of rules states that the grouping of statements combined with semicolons is unimportant, as is the presence or absence of empty statements.

$$\begin{array}{lcl} S_1\ ;\ S_2 & \equiv & S_2\ ;\ S_1 \\ S_1\ ;\ (S_2\ ;\ S_3) & \equiv & (S_1\ ;\ S_2)\ ;\ S_3 \\ \epsilon\ ;\ S & \equiv & S \end{array}$$

Recall that we regard a reduction rule to be applicable to a term if it is applicable to any equivalent term.

4.3 λ_{let} Reduction Rules

We are now ready to look at the reduction rules themselves. They can be divided into several groups:

(1) instantiation;
(2) function application;
(3) lifting;
(4) δ rules, conditionals, and data structures.

4.3.1 Instantiation (Substitution)

Given a statement that binds an identifier x to a simple expression a, the following rules permit a use of x to be replaced by a. Note that in all these rules a is a *simple expression*, and the x is free in $C[x]$.

$$\{\ x = a\ ;\ S \text{ in } C[x]\ \} \quad \longrightarrow \quad \{\ x = a\ ;\ S \text{ in } C[a]\ \} \qquad \text{(Inst1)}$$

$$(x = a\ ;\ SC[x]) \quad \longrightarrow \quad (x = a\ ;\ SC[a]) \qquad \text{(Inst2)}$$

$$x = a \quad \longrightarrow \quad x = C[a] \qquad \text{(Inst3)}$$

where $a = C[x]$

The context $C[\,]$ in which instantiation takes place is an expression with a hole (written as $[\,]$):

$$\begin{array}{lcll} C[\,] & ::= & [\,] & \mid\ \lambda x.C[\,] \\ & \mid & C[\,]\ E & \mid\ E\ C[\,] \\ & \mid & \{\ S \ \mathtt{in}\ C[\,]\ \} & \mid\ \{\ SC[\,]\ \mathtt{in}\ E\ \} \\ & \mid & \mathrm{PF}_k(\ldots, C[\,], \ldots) & \\ & \mid & \mathrm{CN}_k(\ldots, C[\,], \ldots) & \\ & \mid & \underline{\mathrm{CN}}_k(\ldots, C[\,], \ldots) & \end{array}$$

In this definition, $SC[\,]$ is a *statement* context for an expression; that is, a statement with a hole in place of one of the subexpressions in it. It is defined as follows:

$$SC[\,] \quad ::= \quad x = C[\,] \quad \mid \quad SC[\,]\ ;\ S$$

4.3.2 Function Application (β Reduction)

As mentioned before, a λ expression can be applied to an argument using the β_G rule:

$$(\lambda x.e_1)\ e_2 \longrightarrow \{\ x' = e_2 \text{ in } e_1[x'/x]\ \} \qquad (\beta_G)$$

where x' is a variable which does not otherwise occur.

4.3.3 Expression Lifting

The following rules "lift" expressions that are nested in certain ways. In all these rules "$\{S'$ in $e'\}$" represents an α-renaming of "$\{S$ in $e\}$" to avoid name conflicts with the surrounding scope, and the t's represent variables which do not otherwise occur. The side conditions $e \notin SE$ prevent infinite reduction.

$\{\ \epsilon \text{ in } e\ \}$	$\longrightarrow$	e		(Lift0)
$x = \{\ S \text{ in } e\ \}$	$\longrightarrow$	$(x = e'\ ;\ S')$		(Lift1)
$\{\ S \text{ in } e\ \}$	$\longrightarrow$	$\{\ S\ ;\ t = e \text{ in } t\ \}$	$e \notin SE$	(Lift2)
$(e\ e_2)$	$\longrightarrow$	$\{\ t = e \text{ in } (t\ e_2)\ \}$	$e \notin SE$	(Lift3)
$(e_1\ e)$	$\longrightarrow$	$\{\ t = e \text{ in } (e_1\ t)\ \}$	$e \notin SE$	(Lift4)
$\texttt{cond}(e,\ e_1,\ e_2)$	$\longrightarrow$	$\{\ t = e \text{ in } \texttt{cond}(t,\ e_1,\ e_2)\ \}$	$e \notin SE$	(Lift5)
$PF_k(\cdots\ e,\ \cdots)$	$\longrightarrow$	$\{\ t = e\ \texttt{in}\ PF_k(\cdots\ t,\ \cdots)\ \}$	$e \notin SE$	(Lift6)

The utility of these rules may be seen by considering the following example:

```
{  f = { S1 in λx.e1 };                    (1)
   x = f a;                                (2)
in
     { S2 in λx.e2 } e3 }                  (3)
```

Without Lift1, we would not be able to substitute $\lambda x.e_1$ for `f` in line 2. Similarly, without Lift2, Lift3 and Lift1, we would not be able to apply $\lambda x.e_2$ to e_3.

4.3.4 δ Rules, Conditionals, and Data Structures

A rule needs to be added to turn a nonunderlined constructor into the equivalent underlined constructor.

$$\mathrm{CN}_k(e_1, \cdots, e_k) \longrightarrow \{\ t_1 = e_1;\ \cdots\ ;\ t_k = e_k\ \texttt{in}\ \underline{\mathrm{CN}_k}(t_1,\ \cdots,\ t_k)\ \}$$

The δ and conditional rules from the λ calculus carry over unmodified to λ_{let}.

4.4 (Non-)Confluence

The following theorem by Ariola and Klop [6] states that the introduction of let blocks has destroyed confluence:

Proposition 1 (Ariola and Klop) *λ_{let} is not confluent.*

To see this, consider the following program:

Term0:

```
{ odd  = λn. cond(n = 0, False, even(n-1));
  even = λn. cond(n = 0, True,  odd(n-1));
in
  .. }
```

Suppose we substitute `odd` at its use in `even`. *Term0* becomes:

Term1:
```
{ odd  = λn. cond(n = 0, False, even(n-1));
  even = λn. cond(n = 0, True,  cond((n-1)=0, False, even((n-1)-1)));
in
  .. }
```
Suppose, instead, we substitute `even` at its use in `odd`. *Term0* becomes:

Term2:
```
{ odd  = λn. cond(n = 0, False, cond((n-1)=0, True, odd((n-1)-1)));
  even = λn. cond(n = 0, True,  odd(n-1));
in
  .. }
```
At this point, we can see that it will never be possible to reduce *Term1* and *Term2* to the same term. For example, in *Term1*, if we substitute `even` into the `odd` definition, the `odd` definition will still contain a call to the `even` function; this remains true no matter how may times we repeat this substitution. In `Term2`, on the other hand, the `odd` definition has a call to the `odd` function, and if we subtitute for `odd` this remains true.

This lack of confluence is true even if we permit unrestricted instantiation, i.e., if we allowed an identifier to be substituted by the expression that it is bound to even though it may not be a simple expression. Let us call this calculus λ_0. Note that this calculus no longer preserves sharing of computations.

Proposition 2 (Ariola and Klop) λ_0 *is not confluent.*

4.5 Printable Values and the Print[[]] Function

We now wish to approach the question whether the calculus defined by these rules makes sense; that is to say, whether there is a consistent notion of equality which is preserved under transformation of a term according to the rules. We accordingly wish to develop a notion of the observable information contained in a term.

Trying to distinguish terms by comparing their syntax is problematic, even in the pure λ calculus. However, the problem gets harder when let blocks are an integral part of the syntax. Consider the following two terms:

$$\{\ x = 5 \text{ in } 5\ \} \qquad \text{(M1)}$$
$$\{\ x = 3;\ y = 2 \text{ in } 5\ \} \qquad \text{(M2)}$$

M1 and M2 behave essentially the same in all contexts and thus should be equal but are syntactically very different. It is common to introduce one or more "garbage collection" rules in the calculus to attempt to remove the irrelevant clutter (see, for example,[3]. But even this is inadequate: as the (non-)confluence results of Ariola and Klop ([6]) show, there are pairs of terms which ought to be considered equal which can never reduce to compatible syntactic forms.

We therefore introduce the concept of a "printable value". This is a rudimentary notion of value—for example, in λ_{let} all λ abstractions are represented by the single symbol "λ". It is nevertheless all we need; we consider two terms as semantically distinct if they have different print-values, of course; but we also consider them to be distinct, even if they have the same print value, when there is some context that will have different print-values depending on which of these two terms is inserted.

$$
\begin{array}{lll}
PV & ::= & \bot \\
 & | & \lambda \\
 & | & \mathrm{CN}_0 \\
 & | & \underline{\mathrm{CN}_k}(\mathrm{PV}_1, \cdots, \mathrm{PV}_k)
\end{array}
$$

We also impose an *ordering* on printable values:

$$
\begin{array}{lllll}
\bot & \sqsubseteq & PV & & \\
\lambda & \sqsubseteq & \lambda & & \\
\mathrm{CN}_0 & \sqsubseteq & \mathrm{CN}_0 & & \\
\underline{\mathrm{CN}_k}(v_1, \cdots, v_j \cdots, v_k) & \sqsubseteq & \underline{\mathrm{CN}_k}(v_1, \cdots, v_j' \cdots, v_k) & \text{if } v_j \sqsubseteq v_j' & k \geq 1
\end{array}
$$

In other words, $\bot$ is strictly "less" than everything else, and the ordering on constructed terms is defined recursively based on a pair-wise ordering of corresponding arguments. There is no mutual ordering between λ-expressions, constants and constructed terms, or between constructed terms with distinct constructors.

Print⟦⟧ is really implemented by the Print′⟦⟧ function that carries along with it a second argument, the *environment*, that remembers all identifier bindings in the surrounding scope. This environment argument is a collection of bindings $x = v$.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathrm{Print}[\![e]\!] = \mathrm{Print}'[\![e]\!] \ \mathrm{empty\_environment}} \\
\multicolumn{3}{l}{\mathrm{where}} \\
\quad \mathrm{Print}'[\![x]\!] \ \mathrm{env} & = & \mathrm{Print}'[\![v]\!] \ \mathrm{env} \qquad \text{if } [\![x = v]\!] \in \mathrm{env} \\
\quad \mathrm{Print}'[\![\lambda x.e]\!] \ \mathrm{env} & = & \lambda \\
\quad \mathrm{Print}'[\![\{\ S \text{ in } a\ \}]\!] \ \mathrm{env} & = & \mathrm{Print}'[\![a]\!] \ \mathrm{env}' \\
 & \multicolumn{2}{l}{\text{where } a \in SE \quad \text{and} \quad \mathrm{env}' = \mathrm{Extend}[\![S]\!] \ \mathrm{env}} \\
\quad \mathrm{Print}'[\![\mathrm{CN}_0]\!] \ \mathrm{env} & = & \mathrm{CN}_0 \\
\quad \mathrm{Print}'[\![\underline{\mathrm{CN}_k}(a_1, \cdots, a_k)\ ]\!] \ \mathrm{env} & = & \underline{\mathrm{CN}_k}(\ \mathrm{Print}'[\![a_1]\!] \ \mathrm{env}, \qquad k \geq 1 \\
 & & \qquad \vdots \\
 & & \qquad \mathrm{Print}'[\![a_k]\!] \ \mathrm{env}) \\
\quad \textit{otherwise} & = & \bot
\end{array}
$$

Print′⟦⟧ uses the auxiliary Extend⟦⟧ function which augments an environment with all the bindings in a block's statement provided that the identifiers are bound to values (we assume that α renaming has been used to prevent name clashes among bindings in the environment).

$$
\begin{array}{llll}
\mathrm{Extend}\ [\![\ x = v\ ]\!] \ \mathrm{env} & = & \mathrm{env} + [\![x = v]\!] & \\
\mathrm{Extend}\ [\![\ (S_1\ ;\ S_2)\ ]\!] \ \mathrm{env} & = & \mathrm{Extend}\ [\![\ S_1\ ]\!] \ (\mathrm{Extend}\ [\![\ S_2\ ]\!] \ \mathrm{env}) & \\
\mathrm{Extend}\ [\![\ \cdots\ ]\!] \ \mathrm{env} & = & \mathrm{env} & \textit{otherwise}
\end{array}
$$

Note that being able to print a non-$\bot$ value from a term does not mean that the term has terminated, or even that it will ever terminate. For example, given this term:

```
{     loop = λx.loop x;
      bot  = loop (λy.y);
 in
      (λz.z)  }
```

we can print its value (λ), but the term as a whole will never terminate.

The following two properties of printing are about "monotonicity". The first captures the idea that as we continually reduce a term, its information content continually increases, i.e., printing a value following one or more reductions will only produce a "more defined" value.

Proposition 3 *If* $e \twoheadrightarrow e_1$ *then* $\text{Print}[\![e]\!] \sqsubseteq \text{Print}[\![e_1]\!]$

The second expresses the notion of equality that we wanted: if one expression reduces to another then they are print-equivalent.

Proposition 4 *If* $e \twoheadrightarrow e_1$ *then* $\mathit{Print}^*[\![e]\!] = \mathit{Print}^*[\![e_1]\!]$

Note, as a corollary, that if e is *convertible* to e_1 (that is, by applying rules in either direction), then $\text{Print}^*[\![e]\!] = \text{Print}^*[\![e_1]\!]$.

4.6 The Power of the λ_{let} Calculus

Our instantiation rules permit instantiation of an identifier only when it is bound to a simple expression. The following theorm by Ariola et al. [5] assures us that we have not lost any computational power due to this restriction on substitution:

Proposition 5 *(Fundamental theorem of graph reduction): If* $e \twoheadrightarrow e_1$ *in* λ_0*, then* $\exists e_2$ *such that* $e \twoheadrightarrow e_2$ *in* λ_{let}*, and* $\mathit{Print}[\![e_1]\!] \sqsubseteq \mathit{Print}[\![e_2]\!]$.

4.7 Reduction Strategies and Standard Reduction

When we actually sit down and perform reductions on a program, we would like to make sure we get at least as much information as one could get from *any* other choice of reductions:

> **Definition:** A *standard reduction strategy* σ ensures that, if $e \twoheadrightarrow e_1$ as a result of some sequence of reduction rules, then $\exists\ e_2$ such that $e \overset{\sigma}{\twoheadrightarrow} e_2$ according to the rules of strategy σ, and $\text{Print}[\![e_1]\!] \sqsubseteq \text{Print}[\![e_2]\!]$.

Briefly, we can devise a parallel standard reduction strategy for λ_{let} in three stages. First, we manipulate the program to put it into kernel form—a canonical form which enables us to ignore several of the lifting rules during the remainder of reduction. We then choose redexes in a fair manner, making sure that we never reduce inside the body of a λ or in the branches of a conditional. Finally, we stop when we run out of candidate redexes in permitted contexts.

5 λ_B: Adding Barriers to λ_{let}

In this section, we describe λ_B, which extends λ_{let} with a *local sequencing mechanism* called a *barrier*, which sequences statements in a let block. λ_B remains a purely functional language (i.e., it has no side-effects). The barrier's sole effect is that a program may be *less defined* than the corresponding program that has the barriers erased, i.e., the presence of barriers may prevent termination, or may even prevent printing a non-$\perp$ value. The barrier gets its practical motivation from controlling side-effects like M-structures; however, all the semantic subtlety of barriers can be studied in this simpler, purely functional setting.

In Sections 5.1, 5.2, and 5.3, we present the λ_B *calculus*, i.e., the syntax of λ_B terms and the reduction rules. In Sect. 5.4, we briefly define the notion of printing for λ_B. Finally, in Sect. 5.5 we provide some commentary on λ_B.

5.1 Syntax

λ_B extends the syntax of λ_{let} with a new type statement called a Barrier:

$$\begin{array}{lcll} S & ::= & \cdots & \text{as before} \\ & | & S \texttt{ --- } S & \text{Barriers (sequential)} \end{array}$$

The Barrier represents sequencing: the statement before the barrier is executed to completion before the statement after the barrier; we refer to these statements as the *pre-region* and the *post-region* of the barrier, respectively. The barrier is a local sequencing construct, and only involves the two statements that are its subterms. A program can contain any number of barriers.

Our previous static recursive scoping rule for identifiers remains the same—in this respect, there is no difference between barriers and semicolons.

5.1.1 Values and Simple Expressions

Values and Simple Expressions remain the same in λ_B as in λ_{let}.

Statements which bind names to actual values are treated specially by the barrier rules. We refer to them as "value statments", H:

$$H \quad ::= \quad x = V \quad | \quad H \; ; \; H$$

5.1.2 Contexts and Renaming

Expression contexts remain the same in λ_B as in λ_{let}. Statement contexts for expresssions extend those in λ_{let} with new clauses for barriers:

$$\begin{array}{lcl} SC[\,] & ::= & \ldots \\ & | & SC[\,] \texttt{ --- } S \quad | \quad S \texttt{ --- } SC[\,] \end{array}$$

5.2 λ_B Equivalence Rules

The α-renaming and semicolon equivalence rules remain the same in λ_B as in λ_{let}. We add a new rule for barriers:

$$(H \; ; \; S_1) \texttt{ --- } S_2 \qquad \equiv \qquad H \; ; \; (S_1 \texttt{ --- } S_2)$$

This rule allows value bindings to escape from the pre-regions of barriers. The escaped bindings may enable other instantiations in the surrounding context. When all such bindings can escape, the barrier can discharge—see Sect. 5.3.1 below.

5.3 λ_B Reduction Rules

The existing reduction rules remain the same. We add a new group of rules for barriers.

5.3.1 Barrier Rules for λ_B

The following rules say that a barrier can be "discharged" (eliminated) once the pre- or post-region is empty (the first rule is the most common in practice).

$$\begin{array}{lcll} \epsilon \texttt{ --- } S & \longrightarrow & S & \text{(BAR1)} \\ S \texttt{ --- } \epsilon & \longrightarrow & S & \text{(BAR2)} \end{array}$$

The following proposition regarding barriers states that once all the statements in the pre-region of a barrier have terminated, the barrier can be discharged immediately:

Proposition 6 *In any context, the following reduction is always correct:*

$$(H \;\texttt{---}\; S) \longrightarrow (H \;;\; S)$$

5.4 Printable Values, and the Print⟦⟧ Function

Printable values, and their ordering, remain the same in λ_B as in λ_{let}.

In the Print⟦⟧ function, we add one clause to the Extend⟦⟧ helper function for barrier terms:

$$\text{Extend } [\![\; (S_1 \;\texttt{---}\; S_2) \;]\!] \text{ env } \quad = \quad \text{Extend } [\![\; S_1 \;]\!] \text{ env}$$

Note that it omits identifiers from the post-region.

All the Print and Print* propositions of λ_{let} (in Sect. 4.5) remain unchanged.

5.4.1 Stable Terms and Terminated Terms

As we continually reduce a term, we may reach a stage where, even though we may continue reducing it (perhaps forever), the result of applying Print⟦⟧ to the term will no longer change. We say that the term has reached a stable state, and we capture this formally in the following definition:

> **Definition:** A term e is *stable* if $\text{Print*}[\![e]\!] = \{ \; a \mid a \sqsubseteq \text{Print}[\![e]\!] \; \}$

When reduction stops the term may still contain "computations" (such as applications) that are unable to make progress simply because of a deadlock condition. We would like to define the category of "terminated terms" that have not stopped for this reason, i.e., terms that have stopped cleanly and properly:

> **Definition:** *Terminated terms* are described by the following syntax:
> $$ET \quad ::= \quad V \quad \mid \quad \{ \; H \text{ in } SE \; \}$$

A terminated expression contains no applications, unless they are inside λ abstractions. Note, however, the body of any λ expression can be an arbitrary expression. Terminated terms are stable:

Proposition 7 *If $e \in ET$, then e is stable.*

Further reductions may still be possible within λ-bodies, but this cannot affect what is printed.

Another interesting class of terms are those terms whose printed values contain no $\perp$.

> **Definition:** A *ground-printing term* is a term e such that $\text{Print}[\![e]\!]$ terminates, and the printed value does not contain $\perp$.

The printed value value may be an arbitrarily deep data structure, and this condition requires that none of the leaves are $\perp$. Note that a ground-printing term is always stable, but it may not have terminated; this is possible because Print⟦⟧ does not explore "irrelevant" computations, which may not have terminated and which, indeed, may even loop forever.

5.5 Discussion of λ_B

The interaction between our β_G rule for function application and the barrier rules is subtle. Perhaps the most surprising consequence is that:

$$\lambda x.e \quad \neq \quad \{\ t = e \text{ in } \lambda x.t\ \} \qquad\qquad \text{even if } e \text{ is only a free variable } (\neq x).$$

The inequality is not surprising in eager languages such as SML and Scheme—if e were a non-terminating computation, the left-hand side would terminate and the right-hand side would not. However, if e was merely a free variable (say, z), then the two sides *would* be equal in SML and Scheme, but they are still not equal in λ_B.

Consider the following term (where $\perp$ is any non-terminating term):

```
( K = λx.1 ;
  b = ⊥ ;
  ( x1 = K b
    ---
    S ) )
```

Even though `K` discards its argument, the barrier will not discharge, as demonstrated by the following sequence of reductions (for comparison, we also show what would happen with the traditional β rule):

```
       ( K = λx.(λy.y) ;
         b = ⊥ ;
         ( x1 = K b
           ---
           S ) )

⟶      ( K = λx.(λy.y) ;
         b = ⊥ ;
         ( x1 = (λx.(λy.y)) b
           ---
           S ) )

⟶     [using βG]                            [using traditional β]

       ( K = λx.(λy.y) ;                    ( K = λx.(λy.y) ;
         b = ⊥ ;                              b = ⊥ ;
         ( x1 = { x’ = b in (λy.y) }          ( x1 = (λy.y)
           ---                                  ---
           S ) )                                S ) )

⟶      ( K = λx.(λy.y) ;                    ( K = λx.(λy.y) ;
         b = ⊥ ;                              b = ⊥ ;
         ( x1 = (λy.y) ;                      ( x1 = (λy.y) ;
           x’ = b                               S ) )
           ---
           S ) )

⟶      ( K = λx.(λy.y) ;                    ( K = λx.(λy.y) ;
         b = ⊥ ;                              b = ⊥ ;
         x1 = (λy.y) ;                        x1 = (λy.y) ;
         ( x’ = b                             S )
           ---
           S ) )
```

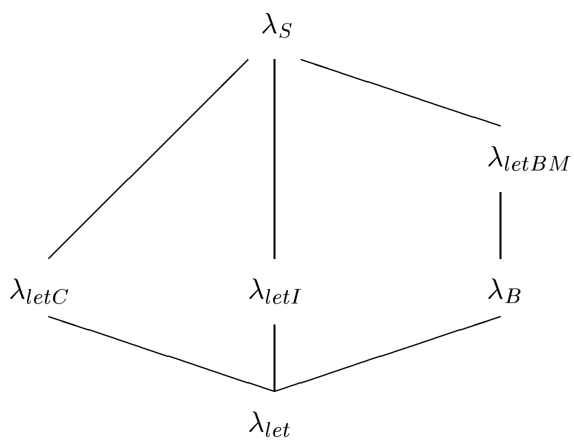

Figure 1: A taxonomy of calculi used to study pH semantics

The last term (on the left) is stuck: since `b` never becomes a value, the barrier never discharges. This behaviour is fundamentally related to our β_G rule which does not discard the argument `b` even though the function itself does not use the argument. In contrast, using the traditional β rule (on the right), the barrier does discharge.

The addition of barriers allows us to observe the *termination* of any term in the calculus. For example, the following two terms are indistinguishable in λ_{let}:

$\{\ x = 5 \text{ in } 5\ \}$ and $\{\ x = \bot \text{ in } 5\ \}$

The following context will distinguish them in λ_B:

$\{\ (\ y = [\,]$

$z = 3\ )$
in
z $\}$

It should be clear from this example that equality in λ_{let} does not imply equality in λ_B.

6 Conclusion

λ_B is just one example of a useful extension of pure λ_{let} which enables us to study issues of termination in the absence of other complications. pH, the parallel dialect of Haskell that we are designing, also has I-structures[7] and M-structures, which add implicitly synchronized mutable storage to the language. In Figure 1 we show various extensions we have formulated and studied, culminating in λ_S, which is intended to capture all the essential semantic properties of pH itself. λ_S is also playing the desired role in the pH compiler: each rule for code generation and optimization is written in terms of λ_S constructs.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *J. of Functional Programming*, 1(4):375–416, 1991.

[2] Shail Aditya, Arvind, Lennart Augustsson, Jan-Willem Maessen, and Rishiyur S. Nikhil. Semantics of pH: A Parallel Dialect of Haskell. In *Proc. Haskell Wkshp. (FPCA 95), La Jolla CA, USA*, June 1995.

[3] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. In *Proc. ACM Conf. on Principles of Programming Languages*, pages 233–246, 1995.

[4] Zena M. Ariola and Arvind. A Syntactic Approach to Program Transformations. In *Proc. Symp. on Partial Evaluation and Semantics Based Program Manipulation, Yale University, New Haven CT, USA*, June 1991. Also CSG Memo 322, MIT Lab for Computer Science.

[5] Zena M. Ariola and Stefan Blom. Lambda calculus plus letrec: graphs as terms and terms as graphs. Technical Report DRAFT, Dept. of Computer and Information Sciences, Univ. of Oregon, Eugene OR, USA, October 1996.

[6] Zena M. Ariola and J. W. Klop. Lambda calculus with explicit recursion. Technical Report CIS-TR-96-04, Dept. of Computer and Information Sciences, Univ. of Oregon, Eugene OR, USA, 1996.

[7] Arvind, Rishiyur Sivaswami Nikhil, and Keshav Kumar Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Trans. on Programming Languages and Systems*, 11(4):598–632, October 1989.

[8] William Clinger and Jonathan Rees (eds.). Revised^4 Report on the Algorithmic Language Scheme. Technical report, MIT AI Laboratory, November 2 1991.

[9] Paul Hudak *et.al.* Report on the Programming Language Haskell, A Non-strict, Purely Functional Language, Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[10] John Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. ACM Conf. on Principles of Programming Languages*, pages 144–154, 1993.

[11] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge MA, USA, 990.

[12] Rishiyur Sivaswami Nikhil. Id (Version 90.1) Language Reference Manual. Technical Report CSG Memo 284-2, MIT Lab for Computer Science, 545 Technology Square, Cambridge MA 02139, USA, July 15 1991.

[13] Christopher P. Wadsworth. Semantics and Pragmatics of the Lambda-calculus, 1971. D.Phil. thesis, University of Oxford.