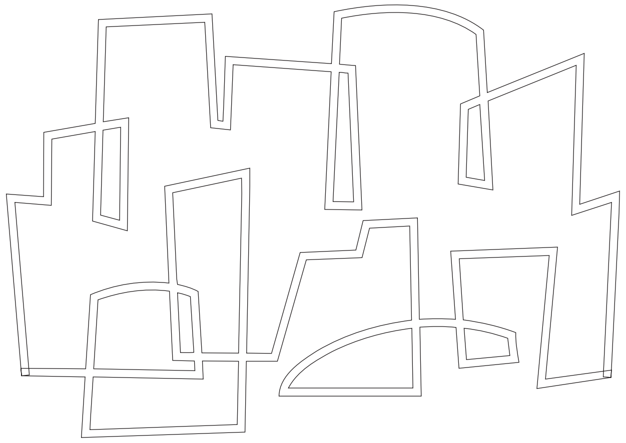
Lambda-S: an implicitly parallel λ-calculus with recursive bindings, synchronization and sid

Jan-Willem Maessen, Arvind, R.S. Nikhil, Joe Stoy

Based on paper submitted to ICFP '97

1996, November

Computation Structures Group
Memo 393



λ_S: an Implicitly Parallel λ-Calculus with Letrec, Synchronization and Side-Effects

Computation Structure Group Memo 393
November 25, 1996

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*

Submitted to ICFP '97

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.

λ_S: an Implicitly Parallel λ-Calculus with Letrec, Synchronization and Side-Effects

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*

November 25, 1996

Abstract

λ_S extends the λ-calculus with letrecs, barriers, and updateable memory cells with synchronized operations. The calculus is designed to be useful for reasoning about compiler optimizations and, thus, allows reductions anywhere, even inside λ's. Despite the presence of side-effects, the calculus retains fine-grained, implicit parallelism and non-strict functions; there is no global, sequentializing store. Barriers, for sequencing, capture a robust notion of termination. Although λ_S was developed as a foundation for the parallel functional languages pH and Id, we believe that barriers give it wider applicability– to sequential and explicitly parallel languages. In this paper we describe the λ_S-calculus and its properties which are based on a notion of observable information in a term. We also describe reduction strategies to compute the maximal observable information.

1 Introduction

The λ-calculus [9] and its variants have been invaluable in capturing the essence of languages such as Haskell [13], SML [17] and Scheme [11], providing the theoretical foundation for reasoning about and transforming programs in these languages. In this paper we describe the λ_S-calculus, another variant, intended as the formal basis for the parallel programming languages pH [2] (an extension of Haskell) and Id [18].

λ_S incorporates implicit parallelism, letrecs, synchronization and side-effects. The fine-grained parallelism evident in functional (especially non-strict) languages is exploited by default. Together with letrec, it permits programs with dynamic cyclic dependencies, a popular technique in lazy programs. The *barrier* is a control mechanism for sequencing. Side-effects are bundled with data-oriented synchronization in two mechanisms called I-structures [8] and M-structures [10]. We believe λ_S meets the following goals:

- An easy-to-understand operational semantics of pH. This includes a definition of observable values and a standard parallel reduction strategy to compute these values.
- Maximum flexibility, in order to reason about compiler optimizations. For example, it should permit reductions *inside* λ's and conditionals (because this is what compilers do) even though such reductions may not occur during execution.

The novelty of λ_S lies in the way it treats barriers and side-effects. In the FP literature, introduction of side-effects is usually in the context of a sequential operational semantics (see, for example, [12] and [16] for call-by-value languages, or state monads in Haskell [15]), and state is often modeled by a single global store with sequential operations. λ_S, on the other hand, introduces no such sequentialization, nor a separate store. The "store" is distributed throughout the term and we give small-step semantics for its manipulation. This is crucial for preserving implicit, fine-grained parallelism. Our barrier semantics captures a robust notion of "termination". We believe it is powerful enough to explain call-by-value languages and sequential languages extended with concurrency primitives, and raw (unsynchronized) side-effects. Despite the presence of side-effects, our calculus places *no* restrictions on where reductions may take place in the term.

The introduction of side-effects requires a precise definition of *sharing*. There is a large body of literature dealing with sharing, such as [20], [14], [5] and [1]; these are motivated not by side-effects but by efficiency considerations (graph reduction, optimality, *etc.*), and often do not address letrecs and parallelism.

In this paper, we build up to λ_S in three stages. §2 describes λ_C, extending the λ-calculus with letrecs, conditionals, constants and functional data structures. The technical results of λ_C have already been achieved by Ariola and Blom [4]; we use this section to introduce our (somewhat different) notation and machinery. The results are similar for the functional subset, but the Ariola and Blom semantic model does not carry over to barriers and side-effects. §3 describes λ_B, adding sequencing barriers to λ_C. §4 describes λ_S, the full calculus, adding side-effects and data-oriented synchronization. In §5 we discuss parallel reduction strategies.

In each proposition we present, we follow the convention that, unless a limitation is explicitly stated, it holds for λ_C, λ_B and λ_S. Due to space limits, we omit all proofs, deferring them to a companion technical report [7] (in preparation). Furthermore, we assume that λ_S terms are generated from a statically typed language (like pH), and thus ignore reductions precluded by static type-checking.

2 λ_C: Letrecs, Constants and Functional Data Structures

λ_C is purely functional (no side-effects). Letrecs allow sharing of computations by binding an expression to an identifier which may be used in a number of places—including recursively in its own definition. While this may be viewed as merely an optimization in a purely functional language, it is semantically essential for barriers and side-effects. Theoretically, we could work without conditionals, constants and functional data structures, but we include them to improve intuition.

The calculus will be presented by describing the syntax of terms, equivalence rules for ignoring trivial syntactic differences between terms, and reduction rules.[1] Then, we discuss equivalence between terms using a notion of printable observations. Finally, we discuss garbage collection, or erasure of irrelevant terms, which other authors often include in their reduction rules.

[1]As usual, the "$e \longrightarrow e_1$" denotes one application of a reduction rule; "$e \twoheadrightarrow e_1$" denotes a finite sequence (zero or more) of reductions. A *redex* is a term or subterm which is capable of reduction by one of the rules.

Syntax of λ_C

Expressions:			
E	$::=$	$x \mid E\ E \mid \lambda x.E$	Identifiers, applications, abstractions
	$\mid$	$\{\ S$ `in` $E\ \}$	Letrec blocks
	$\mid$	`Cond`$(E,E,E) \mid \mathrm{PF}_k(E_1,\cdots,E_k)$	Conditionals, primitive fn of arity k
	$\mid$	$\mathrm{CN}_0 \mid \mathrm{CN}_k(E_1,\cdots E_k)$	Constants, constructors of arity k
Non-initial expressions:			
	$\mid$	$\underline{\mathrm{CN}_k}(SE_1,\cdots SE_k)$	Constructor values of arity k
Statements (in blocks):			
S	$::=$	ϵ	Empty statements
	$\mid$	$x = E$	Bindings
	$\mid$	$S\ ;\ S$	Parallel statements

The syntax of λ_C is shown above. Constants CN_0 include numbers and booleans; functional data structure constructors CN_k include "`cons`"; primitives PF_k include arithmetic operators and projection functions (`Proj`$_j$) to select components of functional data structures. Non-initial expressions do not appear in user-written programs, but only as part of the reduction process. Arguments of $\underline{\mathrm{CN}_k}()$ are always *simple expressions* (SE, defined shortly: identifiers and values); with recursive blocks, this allows us to write cyclic data structures finitely. In a letrec block "{ S `in` E }", E is called its *return expression*. We use the general term "statement" for S because λ_S will later introduce components that just perform side-effects. Empty statements are a technical convenience that become useful in λ_B and λ_S. The left-hand side identifiers in the bindings of S must be pair-wise distinct. The block has recursive scoping rules—any left-hand side identifier may be referred to in any right-hand side expression, as well as in the return expression.

Syntax of Values and Simple Expressions in λ_C: During reduction we need to *instantiate* an identifier's use with its definition. We cannot instantiate arbitrary expressions, as this may replicate work and destroy sharing. We identify two subsets of λ_C, *Values* and *Identifiers*, together known as *Simple Expressions*, that are substitutable:

$$SE\ ::=\ x\ \mid\ V \qquad\qquad \text{Identifiers and values}$$
$$V\ ::=\ \lambda x.E\ \mid\ \mathrm{CN}_0\ \mid\ \underline{\mathrm{CN}_k}(SE_1,\cdots SE_k)$$

2.1 λ_C Syntactic Equivalence Rules

As in λ-calculus, bound variable names are just dummy names—their exact choice does not matter. Further, the exact textual order of statements in a parallel composition does not matter:

α-renaming:

$$\lambda x.e \equiv \lambda t.(e[t/x])$$
$$\{\ x = e\ ;\ S \text{ in } e_0\ \} \equiv \{\ t = e\ ;\ S \text{ in } e_0\ \}[t/x]$$

Equivalence properties of "`;`"

$$\epsilon\ ;\ S \equiv S$$
$$S_1\ ;\ S_2 \equiv S_2\ ;\ S_1$$
$$S_1\ ;\ (S_2\ ;\ S_3) \equiv (S_1\ ;\ S_2)\ ;\ S_3$$

α-renaming is done in the standard way; we omit details here. We use "$e[t/x]$" to denote expression e with all free occurrences of identifier x replaced by identifier t. We use e' and S' to denote renamed versions of e and S. t is always an identifier which does not otherwise occur in the left-hand term of the rule.

2.2 λ_C Reduction Rules

Function Application (β-reduction):

$$(\lambda x.e_1)\ e_2 \longrightarrow \{\ t = e_2 \text{ in } e_1[t/x]\ \} \qquad (\beta_{let})$$

There are two things to note about β_{let}. First, e_2 can be an arbitrary expression, not just a value—this makes it *non-strict*. Second, we always preserve exactly one copy of e_2—unlike traditional β, which may copy or erase e_2 depending on the number of uses of x in e_1. Together with the restricted instantiation rule below, this preserves *sharing* and side-effects, and is important for the semantics of barriers. β_{let} produces graph structured terms, whereas traditional β produces trees.

Instantiation (Substitution): Given $x = a$ in a letrec block, *where a is a simple expression*, these reduction rules specify that we can replace uses of x by a:

$$\begin{array}{lll}
\{\ x = a\ ;\ S \text{ in } C[x]\ \} & \longrightarrow\ \{\ x = a\ ;\ S \text{ in } C[a]\ \} & \text{(Inst1)}\\
x = a\ ;\ SC[x] & \longrightarrow\ x = a\ ;\ SC[a] & \text{(Inst2)}\\
 & & \\
x = a & \longrightarrow\ x = C[a] & \text{(Inst3)}\\
 & \quad\ \ \text{where } a\ =\ C[x] &
\end{array}$$

For lack of space we omit the obvious definitions of $C[\,]$ and $SC[\,]$ which are contexts (expressions and statements, respectively, into which we can plug an expression). There are no restrictions–substitutions can even occur inside λ's. The calculus thus permits infinite unfolding of cyclic terms; we leave it to reduction strategies to control this.

Block flattening:

$$x = \{\ S \text{ in } e\ \} \quad\longrightarrow\quad x = e'\ ;\ S' \qquad \text{(Flat)}$$

Expression lifting: Each rule below "lifts out" a nested expression by giving it a name (using α-renaming where necessary to avoid name conflicts). To prevent instantiation and lifting from alternating forever, lifting rules are inapplicable when e is a simple expression.

$$\begin{array}{lll}
\{\ S \text{ in } e\ \} & \longrightarrow\ \{\ S\ ;\ t = e \text{ in } t\ \} & \text{(LiftB)}\\
(e\ e_2) & \longrightarrow\ \{\ t = e \text{ in } t\ e_2\ \} & \text{(LiftAp1)}\\
(e_1\ e) & \longrightarrow\ \{\ t = e \text{ in } e_1\ t\ \} & \text{(LiftAp2)}\\
\texttt{Cond}(e, e_1, e_2) & \longrightarrow\ \{\ t = e \text{ in } \texttt{Cond}(t, e_1, e_2)\ \} & \text{(LiftCond)}\\
PF_k(\cdots,\ e,\ \cdots) & \longrightarrow\ \{\ t = e \text{ in } PF_k(\cdots, t, \cdots)\ \} & \text{(LiftPF)}
\end{array}$$

Flattening and lifting produce new juxtapositions that trigger other reductions. For example:

$$\texttt{f = \{}\ \ S_1\ \ \texttt{in}\ \ \lambda x.e_1\ \texttt{\};}\ \ \texttt{x = f a;} \qquad \overset{\text{(Flat)}}{\longrightarrow} \qquad \texttt{f =}\ \lambda x.e_1'\ \ \texttt{;}\ \ S_1'\texttt{;}\ \ \texttt{x = f a;}$$

which then allows us to instantiate `f` by $\lambda x.e_1'$ in the application. Note that we do not lift expressions from λ bodies, nor from the arms of a conditional. While safe in λ_C, such rules would be incorrect in λ_B and λ_S because they would "expose" (make observable) potential non-terminations or side-effects that would normally be protected by those contexts.

Constants, conditionals, delta rules: As usual, we regard constants (CN_0) as already evaluated, and:

$$\begin{array}{llll}
\texttt{Cond(True,}\ e_1\texttt{,}\ e_2\texttt{)} & \longrightarrow\ e_1 & \text{(CondT)} & \\
\texttt{Cond(False,}\ e_1\texttt{,}\ e_2\texttt{)} & \longrightarrow\ e_2 & \text{(CondF)} & \\
\text{PF}_k(v_1,\cdots,v_k) & \longrightarrow\ w & (\delta) & e.g.,\ \texttt{+}(\underline{n},\underline{m}) \longrightarrow \underline{n+m}\\
\texttt{proj}_j\texttt{(}\underline{\text{CN}_k}\texttt{(}e_1\texttt{,}\ \cdots\texttt{,}\ e_j\texttt{,}\ \cdots\texttt{))} & \longrightarrow\ e_j & \text{(Proj)} &
\end{array}$$

Constructors: Constructor terms in the initial program are reduced to constructor values:

$$\mathrm{CN}_k(\cdots, e_j, \cdots) \quad\longrightarrow\quad \{ \cdots ; t_j \texttt{ = } e_j ; \cdots \texttt{ in } \underline{\mathrm{CN}_k}(\cdots, t_j, \cdots) \} \qquad \text{(Cons)}$$

Here, e_j need not be a simple expression. The resulting "$\underline{\mathrm{CN}_k}$()" term has identifier arguments and, thus, there are no lifting rules for constructors.

2.3 Comparing terms for information content

Having presented the calculus for λ_C, we must now ask if there is a non-trivial notion of the "meaning" of a term that is preserved under reduction. It is traditional here to discuss the "normal form" of a term and reduction strategies to achieve it.[2] Unfortunately, because of letrec blocks, direct syntactic characterizations (such as normal forms) are problematic. First, terms may contain irrelevant clutter (since β_{let} does not discard argument expressions). But even if we are tempted to use "garbage collection" rules to erase clutter, the following result suggests that letrec blocks make syntactic comparisons futile:

Proposition 1 (Non-confluence (Ariola and Klop [6])) *λ_C is not confluent.*

To see this, consider this term, which represents an infinite list of alternating 1's and 2's:

```
{  x  =  Cons(1, y) ; y  =  Cons(2, x)    in    x  }
```

Let us take two paths, by substituting for x and for y, respectively. The resulting terms are:

```
{  x  =  Cons(1, y) ; y  =  Cons(2, Cons(1, y))    in    x  }          (M1)
{  x  =  Cons(1, Cons(2, x)) ; y  =  Cons(2, x)    in    x  }          (M2)
```

From this point on, in M_1 the y definition will always contain y, whereas in M_2 the y definition will always contain x, *i.e.,* we can never bring these two terms together syntactically again. However, M_1 and M_2 will behave identically in all contexts.

Thus, instead of syntactic comparisions, we compare observable behavior. We first discuss the "instantaneous print" or Print⟦⟧ of a term. Then we discuss the "full print" or Print*⟦⟧ of a term, *i.e.,* what can be observed about a term after reduction.

2.3.1 Print⟦⟧: instantaneous observable information in a term

Print⟦e⟧ transforms a term into the syntax E_P. It preserves constants and constructors; replaces applications, conditionals and nested letrecs by the symbol "$\perp$"; replaces any λ-expression with the symbol "λ" (and ignores the λ-body), and so on.

To handle cyclic terms, we define printing in two stages: a Print⟦⟧ function captures the graph structure of the term, and an Unfold() function flattens graphs into trees that can be compared. Print⟦⟧ produces terms with the following syntax:

$$\begin{array}{lcllllll}
E_P & ::= & V_P & \mid & \{ S_P \text{ in } SE_P \} & & & \\
S_P & ::= & \epsilon & \mid & x = V_P & \mid & S_P \,;\, S_P & \\
SE_P & ::= & x & \mid & V_P & & & \\
V_P & ::= & \perp & \mid & \lambda \quad \mid \quad \mathrm{CN}_0 & \mid & \mathrm{CN}_k(SE_{P_1}, \cdots SE_{P_k}) &
\end{array}$$

[2]It is also possible to map terms to a separate space of denotational meanings (see, *e.g.,* [19]), but for λ_B and λ_S this is quite complicated and (for λ_S) rather indirect.

Print$\llbracket\rrbracket$ simply discards λ bodies, applications, conditionals, nested blocks, *etc.*:

$$\begin{array}{lcll}
\mathrm{Print}\llbracket\{\ S\ \mathtt{in}\ E\ \}\rrbracket & = & \{\ \mathrm{Print}_S\llbracket S\rrbracket\ \mathrm{in}\ \mathrm{Print}_{SE}\llbracket E\rrbracket\ \} & \\
\mathrm{Print}\llbracket E\rrbracket & = & \mathrm{Print}_V\llbracket E\rrbracket & \textit{otherwise} \\
 & & & \\
\mathrm{Print}_S\llbracket \epsilon\rrbracket & = & \epsilon & \\
\mathrm{Print}_S\llbracket x = E\rrbracket & = & x = \mathrm{Print}_V\llbracket E\rrbracket & \\
\mathrm{Print}_S\llbracket S_1;\ S_2\rrbracket & = & \mathrm{Print}_S\llbracket S_1\rrbracket\ ;\ \mathrm{Print}_S\llbracket S_2\rrbracket & \\
 & & & \\
\mathrm{Print}_V\llbracket \lambda x.E\ \rrbracket & = & \lambda & \\
\mathrm{Print}_V\llbracket \mathrm{CN}_0\ \rrbracket & = & \mathrm{CN}_0 & \\
\mathrm{Print}_V\llbracket \underline{\mathrm{CN}_k}(\cdots,\ e_j,\ \cdots)\rrbracket & = & \mathrm{CN}_k(\cdots,\ \mathrm{Print}_{SE}\llbracket e_j\rrbracket,\ \cdots) & \\
\mathrm{Print}_V\llbracket E\rrbracket & = & \bot & \textit{otherwise} \\
 & & & \\
\mathrm{Print}_{SE}\llbracket x\rrbracket & = & x & \\
\mathrm{Print}_{SE}\llbracket E\rrbracket & = & \mathrm{Print}_V\llbracket E\rrbracket &
\end{array}$$

Unfold$_n$() does a finite unfolding of outputs of Print$\llbracket\rrbracket$, producing trees with the following syntax:

$$T_P \quad ::= \quad \bot \quad | \quad \lambda \quad | \quad \Omega \quad | \quad \mathrm{CN}_0 \quad | \quad \mathrm{CN}_k(T_{P1},\ \cdots\ T_{Pk})$$

It prints values, and follows bindings (including arguments of data structures) by keeping an environment of bindings. However, it follows bindings through the environment only up to a finite depth n, bottoming out by printing Ω. Ω is equivalent to $\bot$, but we use it as a technical convenience to distinguish partially-unfolded printable terms from unevaluated printable terms.

$$\begin{array}{lcll}
\mathrm{Unfold}_n(\{S_P\ \mathrm{in}\ SE_P\}) & = & \mathrm{Unfold}'_n(SE_P)\ (\mathrm{Env}\ S_P\ \{\forall x.x \mapsto \bot\}) & \text{generate nodes} \\
\mathrm{Unfold}_n(V_P) & = & V_P & \\
 & & & \\
\mathrm{Unfold}'_0(x)\ \rho & = & \Omega & \\
\mathrm{Unfold}'_n(x)\ \rho & = & \mathrm{Unfold}'_{n-1}(\rho(x))\ \rho & \text{one unfolding} \\
\mathrm{Unfold}'_n(\mathrm{CN}_k(\cdots,\ e_j,\ \cdots))\ \rho & = & \mathrm{CN}_k(\cdots,\ \mathrm{Unfold}'_n(e_j)\ \rho,\ \cdots) & \\
\mathrm{Unfold}'_n(V_P)\ \rho & = & V_P & \textit{otherwise} \\
 & & & \\
\mathrm{Env}(\epsilon)\ \rho & = & \rho & \\
\mathrm{Env}(x = V_P)\ \rho & = & \rho\{x \mapsto V_P\} & \\
\mathrm{Env}(S_{P1}; S_{P2})\ \rho & = & \mathrm{Env}(S_{P1})\ (\mathrm{Env}(S_{P2})\ \rho) &
\end{array}$$

Because Print$\llbracket\rrbracket$ does not unfold cyclic terms, it always terminates. When we Print$\llbracket\rrbracket$ these terms representing an infinite list of ones:

```
{  x  =  Cons(1, x)                     in   x  }
{  x  =  Cons(1, y);  y  =  Cons(1, x)  in   x  }
```

we get E_P terms that look the same as the original E terms. When we then Unfold$_n$() them, we obtain a list of n ones terminated by Ω:

```
Cons(1, Cons(1, Cons(1, ... Ω)...)
```

In order to compare T_P trees, we define the following ordering:

$$\begin{array}{lcll}
\Omega & = & \bot & \\
\bot & \sqsubseteq & T_P & \\
\lambda & \sqsubseteq & \lambda & \\
\mathrm{CN}_0 & \sqsubseteq & \mathrm{CN}_0 & \\
\mathrm{CN}_k(v_1,\ \cdots,\ v_j\ \cdots,\ v_k) & \sqsubseteq & \mathrm{CN}_k(v_1,\ \cdots,\ v'_j\ \cdots,\ v_k) & \text{if } v_j \sqsubseteq v'_j
\end{array}$$

We now have a way to compare E_P terms. A term E_1 "contains less information" than another term E_2 if each finite unfolding of E_1 is less than some finite unfolding of E_2; and, equivalence of two E_P terms can be derived in the usual way from this ordering:

$$\begin{array}{lcl} E_{P1} \sqsubseteq E_{P2} & \Leftrightarrow & \forall n.\exists m.\text{Unfold}_n(E_{P1}) \sqsubseteq \text{Unfold}_m(E_{P2}) \\ E_{P1} = E_{P2} & \Leftrightarrow & E_{P1} \sqsubseteq E_{P2} \quad \text{and} \quad E_{P2} \sqsubseteq E_{P1} \end{array}$$

Proposition 2 (Print monotonicity) *If* $e \twoheadrightarrow e_1$ *then* $\text{Print}[\![e]\!] \sqsubseteq \text{Print}[\![e_1]\!]$

λ_C is not confluent, but two reduction paths can always be reunited so their print values match:

Proposition 3 (Print confluence in λ_C) *If $e \twoheadrightarrow e_1$ and $e \twoheadrightarrow e_2$, then there exist e_1' and e_2' such that $e_1 \twoheadrightarrow e_1'$ and $e_2 \twoheadrightarrow e_2'$, and Print$[\![e_1']\!]$ = Print$[\![e_2']\!]$*

The following theorem is very important: it assures us that we have not lost any computational power by enforcing sharing using letrec blocks and restricted substitution.

Definition (λ_0): Let λ_0 be the calculus without any restriction on instantiation, *i.e.*, an identifier may be instantiated by an arbitrary expression, not just a simple expression.

Proposition 4 (Fundamental theorem of graph reduction in λ_C (Ariola and Blom [4]))

If $e \twoheadrightarrow e_1$ in λ_0, then $\exists e_2$ such that $e \twoheadrightarrow e_2$ in λ_C, and Print$[\![e_1]\!] \sqsubseteq$ Print$[\![e_2]\!]$.

2.3.2 Print*: all potentially observable information in a term

Print*$[\![]\!]$ is a set of printed terms representing the maximal printable information that can be obtained from a term or any of its reductions. We obtain Print*$[\![e]\!]$ by looking at the outcome of every possible reduction sequence starting at e. A reduction sequence of a term e is a partial function from natural numbers $\mathbf{N}$ to E. It is partially defined up to some n (with s_0 being e itself), and it is a reduction sequence in that for any two adjacent defined elements s_{j-1}, s_j, the former reduces in one step to the latter. We write $s_j\uparrow$ or $s_j\downarrow$ when s is defined or undefined, respectively, at j.

Definition (Reduction Sequence):

$$RS(e) = \{s \mid s_0 = e \quad \wedge \quad \forall j. s_j\downarrow \Rightarrow s_{j+1}\downarrow \quad \wedge \quad \forall j > 0. s_j\uparrow \Rightarrow s_{j-1} \longrightarrow s_j\}$$

The Complete Reduction Sequences for an expression e are all reduction sequences starting at e which are either infinte, or whose final term contains no redexes:

Definition (Complete Reduction Sequences):

$$CRS(e) = \{s \in RS(e) \mid \forall j > 0. s_j\uparrow \wedge s_{j+1}\downarrow \Rightarrow \nexists e'. s_j \longrightarrow e')\}$$

If we Print$[\![]\!]$ each term in a sequence, we get an ascending chain of E_P terms (monotonicity). Intuitively, we want the least upper bound of this chain as the "final" print value. However, since chains may be infinite and E_P terms only form a *preorder*, we use the domain $E_P{}^* \supset E_P$ which is isomorphic to the ideal completion of E_P (and where the isomorphism preserves E_P), in which we can take least upper bounds:

Definition (Print*$[\![]\!]$): $\quad \text{Print}^*[\![e]\!] = \{ \bigsqcup_{j \in \omega, s_j\uparrow} \text{Print}[\![s_j]\!] \mid s \in CRS(e) \}$

Proposition 5 (Print* determinacy in λ_C) *In λ_C, Print*$[\![e]\!]$ has precisely one element.*

Proposition 6 (Print* preservation in λ_C) *In λ_C, if $e \twoheadrightarrow e_1$ then* $\text{Print}^*[\![e]\!] = \text{Print}^*[\![e_1]\!]$.

2.3.3 Termination

A stable term is one whose Print$[\![]\!]$ no longer changes under further reduction:

Definition (stable terms): A term e is *stable* if $\text{Print}^*[\![e]\!] = \{ \text{Print}[\![e]\!] \}$.

It is undecidable in general whether a term is stable; this motivates the identification of the following classes of stable terms.

Definition (terminated terms): *Terminated* terms are described by the following syntax:

$$\begin{array}{lcl} ET & ::= & V \quad | \quad \{ H \text{ in } SE \} \\ H & ::= & x = V \quad | \quad H \ ; \ H \end{array}$$

A terminated expression contains no applications, conditionals, *etc.*, unless they are inside a λ-abstraction or arm of a conditional.

Proposition 7 (Stability of terminated terms) *If $e \in ET$, then e is stable.*

Consider the following two terms:

$$\texttt{x = } \bot \qquad \textit{and} \qquad \{ \ \texttt{x = Cons(1,x)} \quad \texttt{in} \quad \texttt{x} \ \}$$

When we Print$[\![]\!]$ and unfold them, the former gives $\bot$, whereas the latter gives $\texttt{Cons(1, Cons(}\cdots\texttt{,}\Omega\texttt{))}$. It is for this reason alone that we make a distinction between $\bot$ and Ω, and this motivates the following:

Definition (ground-printing terms): In a *ground-printing* term e, $\text{Unfold}_n(\text{Print}[\![e]\!])$ does not contain $\bot$ for every n larger than some m.

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 *never* terminate.

2.4 Garbage Collection of irrelevant terms in λ_C

Our β_{let} rule does not discard argument terms, even if the formal parameter is unused. Thus, terms may contain irrelevant clutter. Our calculus is free of GC rules, and we can study GC as a separate topic.

Definition (garbage collection, or GC): GC is a transformation from E to E that erases part of a term,[3] such that for any context C, $\text{Print}^*[\![C[e]]\!] = \text{Print}^*[\![C[\text{GC}[\![e]\!]]]\!]$

The following is a proposed garbage collection process:[4]

Definition (garbage collection rule GC_0):

$$\{ \ S_G \ ; \ S \text{ in } e \ \} \longrightarrow \{ \ S \text{ in } e \ \} \qquad \text{where} \qquad \forall x \in \text{FV}(S) \cup \text{FV}(e)\text{: } x \notin \text{BV}(S_G)$$

In other words, if S and e do not use any identifier defined by S_G, then S_G can safely be erased. GC_0 can be applied anywhere (including inside λ bodies).

Proposition 8 (GC of unreachable terms (in λ_C)) *In λ_C, GC_0 is a correct rule.*

[3]The Print$[\![]\!]$ function, of course, erases irrelevant differences, but it transforms E to E_P and does not preserve deep structure (*e.g.,* λ bodies).

[4]FV(e) are the free variables of e. BV(s) are the top-level bound variables of statement s, *i.e.,* the left-hand sides of all bindings comprising s. FV(s) are the free variables of statement s, *i.e.,* the free variables of the right-hand sides of all bindings comprising s, minus the variables that s itself defines (*i.e.,* minus BV(s)).

3 λ_B: adding Barriers to λ_C

We now add a control mechanism called the *barrier*, which sequences statements in a letrec block. λ_B remains purely functional (no side-effects). Barriers get their practical motivation from controlling side-effects which are discussed later, but their semantic implications are evident in a purely functional setting. Since λ_B differs from λ_C in only a small way, we present it here informally.[5] The new syntax is:

$$S \quad ::= \quad S_1 \texttt{ --- } S_2 \qquad\qquad \text{Barriers}$$

S_1 (the *pre-region*) must terminate before S_2 (the *post-region*) is visible to the rest of the term. The sequencing involves only these two subterms. Our static recursive scoping rule for letrec bindings remains the same—in this respect, there is no difference between barriers and semicolons. To α-renaming and semicolon equivalences, we add a new syntactic equivalence:

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

(Recall, from our discussion on terminated terms, that: $H ::= x = V \mid H \ ; \ H$). This allows value bindings to escape from pre-regions. The escaped bindings may enable other instantiations in the surrounding context. When all such bindings escape, the barrier can be discharged:

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

It follows that the barrier can be discharged as soon as the pre-region has terminated:

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

In Print⟦⟧, we add a clause that omits statements from post-regions:

$$\text{Print}_S[\![\ S_1 \texttt{ --- } S_2\ ]\!] \quad = \quad \text{Print}_S[\![S_1]\!]$$

A terminated term in λ_B contains no barriers unless it is inside a λ or conditional (just as it contains no application unless it is inside such sub-terms).

All λ_C propositions hold in λ_B except for Prop. 8 (correctness of GC$_0$). The post-region of a barrier has no effect on the computation until the barrier is discharged:

Proposition 9 (Opacity of post-regions) *If* $C[S_1 \texttt{ --- } S_2] \twoheadrightarrow C'[S_1' \texttt{ --- } S_2']$ *then* $\exists\ C''$ *such that* $C[S_1 \texttt{---} S_2] \twoheadrightarrow C''[S_1' \texttt{---} S_2] \twoheadrightarrow C'[S_1' \texttt{---} S_2']$

i.e., it is always possible to concentrate on S_1 and ignore S_2 until the barrier is discharged.

Proposition 10 (Barrier associativity) *In all contexts,* $S_1 \texttt{---} (S_2 \texttt{---} S_3) \equiv (S_1 \texttt{---} S_2) \texttt{---} S_3$

The sole difference between λ_C and λ_B is this:

Proposition 11 (Barriers constrain results) $\mathit{Print}^*[\![C[S_1 \texttt{---} S_2]]\!] \sqsubseteq \mathit{Print}^*[\![C[S_1 ; S_2]]\!]$

where the $P_1 \sqsubseteq P_2$ means that for every p_1 in P_1, there is a p_2 in P_2 such that $p_1 \sqsubseteq p_2$. Barriers essentially make it possible to observe non-termination. When placed before a barrier, the left-hand term below will allow the barrier to discharge, whereas the right-hand term will not:

[5]However, it should be noted that this presentation and the semantics of barriers themselves differ from indirect semantics given previously [3].

Expressions:

E	$::=$	$x \mid E\ E \mid \lambda x.E \mid \{\ S$ `in` $E\ \} \mid$ `Cond`(E,E,E)	(as before)
	$\mid$	$\mathrm{PF}_k(E_1,\cdots,E_k) \mid \mathrm{CN}_0 \mid \mathrm{CN}_k(E_1,\cdots E_k)$	(as before)
	$\mid$	`allocate()`	Cell allocation
	$\mid$	`I-Fetch`(E) $\mid$ `M-Fetch`(E)	I- and M-structure fetch

Non-initial expressions:

	$\mid$	$\underline{\mathrm{CN}_k}(SE_1,\cdots SE_k)$	(as before)
	$\mid$	$\mathtt{O}_j$	Cell identifiers

Statements (in blocks):

S	$::=$	$\epsilon \mid x = E \mid S$ `;` $S \mid S$ `---` S	(as before)
	$\mid$	`S-Store(`E`,`E`)`	Synchronized store

Non-initial statements:

	$\mid$	`heap`	Heap allocator
	$\mid$	`empty(`$\mathtt{O}_j$`)` $\mid$ `error(`$\mathtt{O}_j$`)` $\mid$ `full(`$\mathtt{O}_j$`,`V`)`	Empty, error, full cells

Figure 1: **Syntax of λ_S**

```
5                           { x = ⊥ in 5 }
```

Garbage collection of irrelevant terms in λ_B: Because GC_0 can affect termination, which is now observable, GC_0 is no longer correct for λ_B. We constrain the GC rule so that only unreachable bindings which have terminated are erased.

Definition (garbage collection rule $\mathbf{GC}_v$):

$\{\ B\ ;\ S \text{ in } e\ \} \longrightarrow \{\ S \text{ in } e\ \}$ where $\forall x \in \mathrm{FV}(S) \cup \mathrm{FV}(e)$: $x \notin \mathrm{BV}(B)$
and B is defined by the syntax: $B ::= x = V \quad \mid \quad B\ ;\ B$

Proposition 12 (GC of unreachable terminated terms) *GC_v is a correct rule.*

4 λ_S: adding side-effects and data-oriented synchronization

λ_S is the full calculus, obtained by adding updateable memory cells that are dynamically allocated in a *heap*.[6] Each cell can be in a *full* or *empty* state, which affects the behavior of cell operations. Our formulation preserves small-step semantics (local rules) for fine-grain parallelism; there is no global "store".

The syntax of λ_S is shown in Figure 1. `allocate()` produces a new, `empty` cell named by a *cell identifier* $\mathtt{O}_j$. `S-Store`, `I-Fetch` and `M-Fetch` are cell operations that depend on and affect the cell state. Just before execution, we introduce a "`heap`" term as follows:

$e \quad \longrightarrow \quad \{$ `heap` in $e\ \}$ $\qquad$ (e is the entire program)

This is not a reduction rule; it just prepares the orginal program for execution.

Syntax of Values and Simple Expressions in λ_S: we add cell identifiers:

V	$::=$	$\lambda x.E \mid \mathrm{CN}_0 \mid \underline{\mathrm{CN}_k}(x_1,\cdots x_k)$	(as before)
	$\mid$	$\mathtt{O}_j$	Cell identifiers

[6]Functional data structures in λ_C could also have been explained with an explicit heap.

Heap Terms in λ_S: as before, heap terms are "quiescent" (including value statements):

$$\begin{array}{lcll}
H & ::= & x = V \mid H \texttt{ ; } H & \text{(as before)} \\
 & \mid & \texttt{heap} & \text{Heap allocator} \\
 & \mid & \texttt{empty(O}_j\texttt{)} \mid \texttt{error(O}_j\texttt{)} \mid \texttt{full(O}_j\texttt{,}V\texttt{)} & \text{Empty, error, full cells}
\end{array}$$

The syntactic equivalence rules (α-renaming, semicolons, barriers) remain unchanged.

4.1 λ_S Reduction Rules

New lifting rules: Here, t represents a variable which does not otherwise occur, and $e \notin SE$:

$$\begin{array}{lcll}
\texttt{S-Store(}e\texttt{,}e_2\texttt{)} & \longrightarrow & \texttt{(} t \texttt{ = } e \texttt{ ; S-Store(}t\texttt{,}e_2\texttt{))} & \text{(LiftS-Store1)} \\
\texttt{S-Store(}e_1\texttt{,}e\texttt{)} & \longrightarrow & \texttt{(} t \texttt{ = } e \texttt{ ; S-Store(}e_1\texttt{,}t\texttt{))} & \text{(LiftS-Store2)}
\end{array}$$

Cell operations: `allocate()` creates a new memory cell, named by a new cell identifier:

$$\begin{array}{lcll}
\texttt{(heap ; } x \texttt{ = allocate())} & \longrightarrow & \texttt{(heap ; empty(O}_j\texttt{); } x \texttt{ = O}_j \texttt{)} & \text{(Alloc)} \\
 & & \text{where } \texttt{O}_j \text{ is a new cell identifier} &
\end{array}$$

The `heap` term remains unchanged; it merely enables allocation. Because there are no rules to move the `heap` term into λ's, into conditionals or below barriers, allocation cannot occur in such regions; allocation can only occur in "exposed" computations. This rule also sequentializes: only one allocation can occur at a time. This can be trivially relaxed to a fixed degree of parallelism p by introducing p `heap` terms at the start. It can even be relaxed to an unlimited degree of parallelism by adding the reduction rule:

$$\texttt{heap} \longrightarrow \texttt{heap ; heap}$$

(we merely have to ensure that our standard reduction strategy does not get stuck in this rule.) We can store a value into an empty cell, but not into a full cell:

$$\begin{array}{lcll}
\texttt{(empty(O}_j\texttt{); S-Store(O}_j\texttt{, } v\texttt{))} & \longrightarrow & \texttt{full(O}_j\texttt{, } v\texttt{)} & \text{(S-Store)} \\
\texttt{(full(O}_j\texttt{, } v\texttt{); S-Store(O}_j\texttt{, } w\texttt{))} & \longrightarrow & \texttt{error(O}_j\texttt{)} & \text{(S-StoreErr)}
\end{array}$$

The argument terms must be a cell identifier and a value, not arbitrary expressions, *i.e.,* a cell can only store a value. The I-Fetch and M-Fetch rules say that a value can be extracted from a full cell. However, they differ in the resulting state of the cell:

$$\begin{array}{lcll}
\texttt{(full(O}_j\texttt{, } v\texttt{); } x \texttt{ = I-Fetch(O}_j\texttt{))} & \longrightarrow & \texttt{(full(O}_j\texttt{, } v\texttt{); } x \texttt{ = } v\texttt{)} & \text{(I-Fetch)} \\
\texttt{(full(O}_j\texttt{, } v\texttt{); } x \texttt{ = M-Fetch(O}_j\texttt{))} & \longrightarrow & \texttt{(empty(O}_j\texttt{); } x \texttt{ = } v\texttt{)} & \text{(M-Fetch)}
\end{array}$$

I-Fetch leaves the cell unchanged, allowing further reads. M-Fetch empties the cell, requiring another `S-Store` before further reads. This captures the "I-structure" and "M-structure" synchronization behavior in pH. These rules make λ_S *non-deterministic.* Consider these two program fragments:

$$\begin{array}{lcl}
\texttt{empty(O}_1\texttt{) ;} & \qquad & \texttt{empty(O}_1\texttt{) ;} \\
\texttt{S-Store(O}_1\texttt{, } v\texttt{) ;} & & \texttt{S-Store(O}_2\texttt{, } m\texttt{) ;} \\
\texttt{S-Store(O}_1\texttt{, } w\texttt{) ;} & & \texttt{S-Store(O}_2\texttt{, } n\texttt{) ;} \\
\texttt{x = I-Fetch(O}_1\texttt{)} & & \texttt{y = M-Fetch(O}_2\texttt{)}
\end{array}$$

Since ";" permits various orderings, we can produce any of the following outcomes (respectively):

$$\begin{array}{llcll}
\twoheadrightarrow & \texttt{Error(O}_1\texttt{); x = I-Fetch(O}_1\texttt{)} & \qquad & \twoheadrightarrow & \texttt{Error(O}_2\texttt{); y = M-Fetch(O}_1\texttt{)} \\
\text{or} & \texttt{Error(O}_1\texttt{); x = } v & & \text{or} & \texttt{Full(O}_2\texttt{, } m\texttt{); y = } n \\
\text{or} & \texttt{Error(O}_1\texttt{); x = } w & & \text{or} & \texttt{Full(O}_2\texttt{, } n\texttt{); y = } m
\end{array}$$

4.2 Properties of λ_S

Print⟦⟧ and Print*⟦⟧ remain unchanged. Print*⟦⟧ can now contain more than one element. Print monotonicity (Prop. 2) continues to hold, but Print confluence (Prop. 3), Print* determinacy (Prop. 5), and Print* preservation (Prop. 6) no longer hold; performing an `M-Fetch` may preclude certain reductions that would have been possible by doing that `M-Fetch` later. The net effect is that the set of reachable values becomes smaller. Thus, Print* preservation (Prop. 6) must be weakened:

Proposition 13 (Print* contraction) *If* $e \longrightarrow\!\!\!\!\!\rightarrow e_1$ *then* $\text{Print*}[\![e_1]\!] \subseteq \text{Print*}[\![e]\!]$

All other propositions continue to hold: Stability of terminated terms (Prop. 7); Opacity of post-regions (Prop. 9); Barrier associativity (Prop. 10); Barriers constrain results (Prop. 11), and GC$_v$ of unreachable terminated terms (Prop. 12). A more aggressive GC rule could also eliminate `S-Store`'s and full cells for which it can be shown that no matching `I-` or `M-Fetch` is possible.

4.3 I-Structure Subset of λ_S

Definition (I-structure subset): λ_I is λ_S without the `M-Fetch` syntax and rules.

This subset is important because cells with `S-Store` and `I-Fetch` operations express the same idea as "logic variables", "single-assignment variables", "communication variables", *etc.* found in logic and constraint programming languages. If a reduction sequence terminates without a store error, then the print-result is unique:

Proposition 14 (Termination is unique (in λ_I)) *If* $e \longrightarrow\!\!\!\!\!\rightarrow e'$ *and* e' *is terminated and error-free,*
then $\text{Print*}[\![e]\!] = \{\ \text{Print}[\![e']\!]\ \}$

and all reduction sequences produce a store error on a location $\mathtt{O}_j$ if any one of them does:

Proposition 15 (Store error confluence (in λ_I)) *If* $e \longrightarrow\!\!\!\!\!\rightarrow e_1$ *and* $e \longrightarrow\!\!\!\!\!\rightarrow e_2$,
and e_1 *is* $C[\mathtt{error(O}_j\mathtt{)}]$, *then* $e_2 \longrightarrow\!\!\!\!\!\rightarrow e_2'$, *where* e_2' *is* $C'[\mathtt{error(O}_j\mathtt{)}]$

Thus, although λ_I is non-deterministic (as demonstrated in the left column of the example in Section 4.1), it is more benign: if we consider all programs containing any `error()` as equivalent, then λ_I can be considered to produce deterministic results (this is exactly the view taken in the language pH).

5 Reduction Strategies for λ_S

A *reduction strategy* is a method to choose amongst many available redexes. Traditionally, a *standard* reduction strategy σ produces a term with at least as much information as is obtainable from any other strategy, *i.e.,* if $e \longrightarrow\!\!\!\!\!\rightarrow e_1$ in the calculus, then there is an e_2 such that $e \overset{\sigma}{\longrightarrow\!\!\!\!\!\rightarrow} e_2$ according to the standard reduction strategy σ, and $\text{Print}[\![e_1]\!] \sqsubseteq \text{Print}[\![e_2]\!]$. This notion needs elaboration in the presence of non-determinism. Consider the following example (let `x` be an empty cell):

```
{  S-Store(x,1);
   (  y1 = M-Fetch(x)  ---  S-Store(x,2) );          (F1)
   y2 = M-Fetch(x)                                   (F2)
in
   (y1, y2) }
```

The two `M-Fetch` operations may execute in any order. If F_1 executes first, the result of the block will be (1,2). If F_2 executes first, then F_1 waits on an empty cell, and the result is ($\perp$,1). No *deterministic* strategy can produce both outcomes. This motivates the following two variants of standard reduction. Let $\text{Print}^*_\sigma[\![]\!]$ be a version of $\text{Print}^*[\![]\!]$ that uses only reductions permitted by the strategy.

Definition (Strong and Weak standard reduction): A reduction strategy σ is *strongly standard* when $\forall e.\ \text{Print}^*_\sigma[\![e]\!] = \text{Print}^*[\![e]\!]$ and *weakly standard* when $\forall e.\ \text{Print}^*_\sigma[\![e]\!] \subseteq \text{Print}^*[\![e]\!]$.

Any deterministic strategy can only hope to be weakly standard. But even such a strategy is non-trivial. If we threw in the (irrelevant) statement "`y3=`$\perp_y$" in the above example, it must not get stuck in $\perp_y$ and produce ($\perp$,$\perp$); it must still produce either (1,2) or ($\perp$,1).

In this section, we develop a weakly standard *parallel* strategy. We first define a *kernel* language, a subset of E (assuring ourselves that we have not thereby lost expressive power). Then, we limit instantiation and reduction to the outermost parts of the term. This enforces the barrier synchronization discipline and prevents infinite recursion; it also harmlessly postpones some operations which can safely be left until they appear in an exposed context. Finally, we ensure that we choose *fairly* among available instantiations, to avoid getting stuck in an infinite sequence of reductions that are irrelevant to the observable part of a term.

The kernel language is defined by the following (compare with Fig. 1):

Kernel Terms:

$$\begin{array}{lcl}
E & ::= & x \mid SE\ SE \mid \lambda x.E \mid \{\ S\ \texttt{in}\ SE\ \} \\
 & \mid & \texttt{Cond}(SE,E,E) \mid \text{PF}_k(SE_1,\cdots,SE_k) \mid \text{CN}_0 \mid \underline{\text{CN}_k}(x_1,\cdots x_k) \\
 & \mid & \texttt{allocate()} \mid \texttt{O}_j \mid \texttt{I-Fetch}(SE) \mid \texttt{M-Fetch}(SE) \\
S & ::= & \epsilon \mid x = E \mid S\ \texttt{;}\ S \mid S\ \texttt{---}\ S \\
 & \mid & \texttt{heap} \mid \texttt{S-Store}(SE,SE) \mid \texttt{empty}(\texttt{O}_j) \mid \texttt{error}(\texttt{O}_j) \mid \texttt{full}(\texttt{O}_j,V)
\end{array}$$

An initial term can be *kernelized* by repeated application of the "lifting" rules: LiftB, LiftAp1, LiftAp2, LiftCond, LiftPF, LiftS-Store1, LiftS-Store2, and Cons.

Proposition 16 (Kernelization confluence) *kernelization is confluent up to syntactic equivalence (α-renaming, properties of "`;`" and "`---`") and is strongly normalizing (always terminates).*

Proposition 17 (Kernelization is static) *A kernel expression remains a kernel expression after further reduction.*

Proposition 18 (Expressivity of kernel language) *If e_k is the kernelized version of e, then*
$\text{Print}^*[\![e]\!] = \text{Print}^*[\![e_k]\!]$.

The remaining reduction rules are called *dynamic* rules: β_{let}, Inst2, Flat, CondT, CondF, δ, Proj, BAR1, Alloc, S-Store, S-StoreErr, I-Fetch, M-Fetch.

Limited contexts for instantiation and reduction: We want to perform reductions only where computations are "exposed" (avoiding λ-bodies, arms of conditionals, and post-regions) and we want to prevent useless unfolding of cyclic data structures. We thus limit the contexts[7] where an identifier may be instantiated and where reductions may occur:

Limited instantiation contexts:

$C[\,]$ ::= $[\,]$ $\mid [\,]\, SE$ $\mid$ `Cond`$([\,],E,E)$ $\mid \mathrm{PF}_k(\cdots,[\,],\cdots)$

$SC[\,]$ ::= $x = C[\,]$ $\mid SC[\,]$; S $\mid SC[\,]$ `---` S $\mid$ `S-Store`$(C[\,], E)$ $\mid$ `S-Store`$(E, C[\,])$

Limited reduction contexts:

$C[\,]$ ::= $[\,]$ $\mid \{\ SC[\,]$ `in` $SE\ \}$

$SC[\,]$ ::= $[\,]$ $\mid x = [\,]$ $\mid SC[\,]$; $S \mid SC[\,]$ `---` S

Fairness: The only remaining way to get stuck in an irrelevant sequence of reductions is by repeatedly applying recursive λ-expressions. We avoid this by insuring that we instantiate only a finite amount before turning our attention to other reductions. We decorate a λ-expression with a positive integer, decrementing it each time we instantiate it. Eventually, the decoration reaches zero, disallowing further instantiation:

$x = a\ ;\ SC[x]$	$\longrightarrow$	$x = a\ ;\ SC[a]$	$a \neq \lambda^n y.e$	(Inst2a)
$x = \lambda^n y.e\ ;\ SC[x]$	$\longrightarrow$	$x = \lambda^{n-m} y.e\ ;\ SC[\lambda^m y.e]$	$n \geq m > 0$	(Inst2b)

Definition (σ_n, the weak standard reduction strategies for λ_S)

1. Compute the kernel form of the initial term.
2. While the term is not a terminated term:
   - 2.1 Decorate all λ abstractions with the positive integer $n > 0$.
   - 2.2 Apply the dynamic rules (with limited contexts, *etc.*), in any order, even in parallel, until no longer possible.

This is a family of strategies, parameterized by n and by the choices in Step 2.2. Still, for any n, each invocation of this step will always terminate. The choices in this step represent a kind of non-determinism. For λ_C and λ_B, this non-determinism is benign—there is still a unique outcome, and this strategy will find it, even if $n = 1$ (and so it is strongly standard). However in λ_S any particular choice of n, no matter how large, can force the program's non-determinism to be resolved too early, thereby ruling out some outcomes. Consider this variant on our previous example:

```
{  f =  λj. if (j > 1) then f (j-1)
             else {  (  y = M-Fetch(x) --- S-Store(x,2))   (F1)
                  in
                     y }
   S-Store(x,1);
   y1 = f m;
   y2 = M-Fetch(x)                                         (F2)
in
   (y1, y2) }
```

Now `f` must be instantiated m times before F_1 executes. Again, if F_1 executes before F_2, the result is (1,2), but if F_2 executes first, the result is ($\perp$,1). However, if $n < m$, it forces us to execute F_2 first, so the only possible outcome is ($\perp$,1).

Other interesting reduction strategies: Closely related is the strategy σ which non-deterministically chooses the n used to annotate the λ's at each step. By choosing a sufficiently large

[7]Technically we need to specify, separately, contexts for plugging in an E given an E, for plugging in an S given an E, for plugging in an E given an S, and so on. We overload things, hoping the meaning is always clear.

n at each step we insure that every possible outcome can be reached by some choice of reductions within the strategy. Thus, σ is strongly standard.

Suppose we ignore fairness; call this strategy σ_0. This would have the same termination behavior as σ_n, but may print less of an answer than σ_n. Consider this program:

```
{   K = λx.λy.x ;    loop = λx.loop x     in     K v (loop a) }
```

Neither σ_n nor σ_0 will terminate; however, under σ_n we will eventually print (the value of) `v`, whereas under σ_0 we may print $\perp$ forever by never reducing the `K` application. For programs that *do* have a terminal form, σ_0 is adequate—fairness is not necessary:

Proposition 19 (σ_0 is standard for terminating terms)
If $e \twoheadrightarrow e_1$ and $e_1 \in ET$, then $e \overset{\sigma_0}{\twoheadrightarrow} e_2$ and $Print[\![e_1]\!] \sqsubseteq Print[\![e_2]\!]$

Recall that λ_C (purely functional) *is* deterministic. A *lazy* reduction strategy focuses on "outermost" redexes, always picking one that is needed to produce the answer. Ariola *et al.* [4] show such a strategy, and show that it is standard for λ_C. We can give a parallel strategy σ_{plazy} identical to σ_n, except that we continue evaluation only while the term is not ground-printing (defined in Sec. 2.3.3). The final term may still contain redexes, but further reductions will not affect the printed value (and so termination here makes sense). Indeed, σ_{plazy} finds answers equivalent to those found by lazy reduction.

6 Conclusion

In this paper, we have described λ_S, a fine-grained parallel extension of the λ-calculus with letrec, barriers (for sequencing), and synchronized side-effects. The calculus avoids any encoding into the reduction rules of considerations inspired by reduction strategy. This makes the calculus more flexible and, therefore, more useful for reasoning about program transformations.

Based on the notion of observable information, we have discussed a number of important properties of λ_S and its important sub-calculi, λ_C, λ_B and λ_I. We have described parallel strong and weak standard reduction strategies to compute the maximum information in a term. A simple variation of our strategy subsumes traditional lazy evaluation.

Our barrier semantics capture a robust notion of termination: in (S_1 `---` S_2), we wait for all of the *dynamic* computation in S_1. We believe that this gives λ_S wide applicability, and will allow us to model a spectrum of languages: fine-grained implicitly parallel languages like pH and Id; totally sequential languages (by inserting barriers everywhere); and points in between, such as Scheme extended with futures. Barrier sequencing also makes raw (unsynchronized) side effects manageable locally without compromising parallelism globally.

We believe that λ_S is a simple, flexible, and powerful calculus that will facilitate the study of realistic parallel languages with side-effects.

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] S. Aditya, Arvind, L. Augustsson, J.-W. Maessen, and R. S. Nikhil. Semantics of pH: A Parallel Dialect of Haskell. In *Proc. Haskell Wkshp. (FPCA 95), La Jolla CA, USA*, June 1995.

[3] S. Aditya, Arvind, and J. Stoy. Semantics of barriers in a non-strict, implicitly-parallel language. In *Functional Programming and Computer Architecture*, 1995.

[4] Z. M. Ariola and S. Blom. Cyclic lambda calculi. Technical Report CIS-TR-96-13, Dept. of Computer and Information Sciences, Univ. of Oregon, Eugene OR, USA, 1996.

[5] 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.

[6] Z. M. Ariola and J. W. Klop. Cyclic lambda graph rewriting. In *Proc. Ninth Symp. on Logic in Computer Science (LICS'94), Paris, France*, pages 416–425, 1994.

[7] Arvind, J.-W. Maessen, R. S. Nikhil, and J. E. Stoy. λ_S: An implicitly parallel λ-calculus with letrec, synchronization and side-effects. Technical report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge MA 02139, USA, in preparation. (Full version with proofs).

[8] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Trans. on Programming Languages and Systems*, 11(4):598–632, October 1989.

[9] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North Holland, Amsterdam, 1981.

[10] P. Barth, R. S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-strict, Functional Language with State. In *Springer Verlag LNCS 523 (Proc. Functional Programming and Computer Architecture, Cambridge MA, USA)*, pages 538–568, 1991.

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

[12] M. Felleisen and D. P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69:243–287, 1989.

[13] P. 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.

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

[15] J. Launchbury and S. L. Peyton Jones. Lazy Functional State Threads. In *Proc. ACM SIGPLAN '94 Conf. on Programming Language Design and Implementation, Orlando FL, USA*, pages 24–35, June 22-24 1994.

[16] I. A. Mason and C. L. Talcott. Equivalence in Functional Languages with Effects. *J. of Functional Programming*, 1(3):287–327, July 1991.

[17] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge MA, USA, 990.

[18] R. S. 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.

[19] J. E. Stoy. The Semantics of Id. In *A Classical Mind: Essays in Honor of C.A.R.Hoare (A.W.Roscoe, ed.)*, pages 379–404. Prentice Hall, New York, 1994.

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