LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

λ_S: an implicitly parallel λ-calculus with recursive bindings, synchronization and side effects

Computation Structures Group Memo 393-2

Oct. 26, 1998

Arvind
Jan-Willem Maessen
Rishiyur S. Nikhil
Joe E. Stoy

Appeared in Proc. Workshop on High-Level Concurrent Languages (HLCL98), proc. to appear in ENTCS. 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

λ_S: an implicitly parallel λ-calculus with recursive bindings, synchronization and side effects

Arvind [a,1], Jan-Willem Maessen [a,1], Rishiyur S. Nikhil [b] and Joseph E. Stoy [c]

[a] `[arvind, jmaessen]@lcs.mit.edu`, *MIT Laboratory for Computer Science, 545 Technology Sq., Cambridge, MA 02139, USA.*

[b] `nikhil@crl.dec.com`, *Compaq Computer Corporation Cambridge Research Laboratory*

[c] `stoy@comlab.ox.ac.uk`, *Oxford University Computing Laboratory*

Abstract

λ_S extends the λ-calculus with recursive bindings, barriers, and updatable memory cells with synchronized operations. The calculus can express both deterministic and nondeterministic computations. It 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, explicitly parallel and concurrent languages. In this paper we describe the λ_S-calculus and its properties, based on a notion of observable information in a term. We also describe reduction strategies to compute maximal observable information even in the presence of unbounded nondeterminism.

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,

[1] 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.

intended as the formal basis for the parallel programming languages pH [2] (an extension of Haskell) and its predecessor Id [18].

λ_S incorporates implicit parallelism, recursive binding, synchronization and side effects. The fine-grained parallelism inherent in functional (especially non-strict) languages is exploited by default. Using recursive bindings, this permits programs with dynamic cyclic dependencies, a popular technique in lazy programs. The barrier provides a control mechanism for sequencing. Side effects are bundled with data-oriented synchronization in two mechanisms called I-structures [8] and M-structures [10]; M-structures are used to express non-deterministic computations.

We believe λ_S meets the following goals:

- An easily understood operational semantics for 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, the calculus must permit reductions inside λ abstractions 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 side effects and barriers. In the functional programming 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 the calculus 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], [4], [6] and [1]; these are motivated not by side effects but by efficiency considerations (graph reduction, optimality, *etc.*); much of this work does not address recursive binding or parallelism.

Structure of paper: We build up to λ_S in three stages in order to emphasize the association between each calculus and its properties. §2 describes λ_C, extending the λ-calculus with recursive bindings, conditionals, constants and functional data structures. The technical results of λ_C have already been achieved by Ariola and Blom [5]; we use this section to introduce our notation and machinery. After giving the operational rules for λ_C we describe a notion of information content weaker than that of Ariola and Blom, which nonetheless allows us to establish confluence up to information content, and which can be expanded to reason about termination. §3 describes λ_B, adding sequencing barriers to λ_C. The properties of λ_C continue to hold in λ_B, but we are now able to observe termination. §4 describes λ_S, the full calculus, adding side effects and data-oriented synchronization. Side effects render the calculus nondeterministic; we show that this nondeterminism can be captured formally. In §5 we take a more pragmatic focus and describe additional rules which permit us to kernelize the calculus, simplifying its structure statically. Finally, in §6 we examine several reduction strategies for λ_S. A nondeterministic calculus requires a nondeterministic reduction strategy; strategies are characterized as standard if they will reach some permitted nondeterministic outcome, and strongly standard if they can reach any possible nondeterministic outcome.

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. We omit proofs in this paper; with the exception of confluence and standardization the proofs are straightforward.

It should be noted that we are primarily interested in devising a semantics for pH, which is a statically typed language. In all of our reduction systems it is possible for ill-typed terms to "get stuck" (*e.g.,* an application in which an integer constant appeared in the function position would simply make no further progress), or otherwise execute incorrectly. If we restricted our attention to well-typed programs these situations would never arise; but in fact the possibility of "stuck" terms does no damage to our theory or any of the propositions, and so we simply ignore the matter of typing from now on.

2 λ_C: Recursive blocks and Data Structures

λ_C is purely functional (no side effects). Recursive blocks 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 add verisimilitude.

The syntax of λ_C is shown in Figure 1. Constants CN_0 include numbers and booleans; functional data structure constructors CN_k include "`cons`"; primitives PF_k include arithmetic operators and projection functions ($\mathtt{Proj}_j$) 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 the block "{ S `in` E }", E is called its

Expressions:

$$
\begin{array}{lcll}
E & ::= & x & \text{identifiers} \\
 & | & E\ E & \text{applications} \\
 & | & \lambda x.E & \text{abstractions} \\
 & | & \{\ S\ \texttt{in}\ E\ \} & \text{recursive blocks} \\
 & | & \texttt{Cond}(E,E,E) & \text{conditionals} \\
 & | & \mathrm{PF}_k(E_1,\cdots,E_k) & \text{primitives} \\
 & | & \mathrm{CN}_0 & \text{constants} \\
 & | & \mathrm{CN}_k(E_1,\cdots E_k) & \text{constructors}
\end{array}
$$

Non-initial expressions:

$$
\begin{array}{lcll}
 & | & \underline{\mathrm{CN}_k}(SE_1,\cdots SE_k) & \text{constructor values}
\end{array}
$$

Statements (in blocks):

$$
\begin{array}{lcll}
S & ::= & \epsilon & \text{empty statements} \\
 & | & x = E & \text{bindings} \\
 & | & S\ ;\ S & \text{parallel statements}
\end{array}
$$

$$
\begin{array}{lcll}
SE & ::= & x \quad | \quad V & \text{simple expressions} \\
V & ::= & \lambda x.E \quad | \quad \mathrm{CN}_0 \quad | \quad \underline{\mathrm{CN}_k}(SE_1,\cdots SE_k) & \text{values}
\end{array}
$$

Fig. 1. **Syntax of λ_C**

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; a variable is bound at most once in any statement group. 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.

We begin our presentation of λ_C by presenting equivalence rules for ignoring trivial syntactic differences between terms, and then describe the reduction rules in some detail. Next we discuss equivalence between terms using information content. We claim that the calculus is confluent with respect to information, and is equivalent to a calculus without restrictions on instantiation and reduction.

2.1 λ_C Syntactic Equivalence Rules

The exact textual order of statements in a parallel composition does not matter. We express this using equivalences among parallel composition, as this will allow us to add barriers to the calculus. α-renaming is done in the standard way. The resulting equivalences are given in Figure 2. We use "$e[t/x]$" to denote expression e with all free occurrences of identifier x replaced by the identifier t, which is always fresh. We use e' and S' to denote appropriately renamed versions of e and S in rules where renaming is necessary.

α-renaming:

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

Equivalence properties of ";"

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

Fig. 2. **Equivalence rules for λ_C**

2.2 λ_C Reduction Rules

The reduction rules of λ_C divide roughly into two classes. The operational rules describe what we might consider the "useful work" of the calculus, such as β-reduction. Lifting rules exist purely to manipulate the block structure of a term so that opportunities for reduction can be exposed; every construct subject to reduction will have an associated lifting rule. In order to understand how these rules relate, we present the calculus a construct at a time.

2.2.1 Application

β-reduction: Any calculus which expects to capture non-strictness while preserving sharing must contain some equivalent of the β_{let} rule.

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

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, to preserve sharing and side effects we keep exactly one copy of e_2, and rely on separate instantiation rules. Note that we will retain e_2 even when it is no longer used, relying on the fact that garbage terms have no observable effect; the calculi discussed here have no garbage collection rules. β_{let} produces graph structured terms, whereas traditional β produces trees.

Expression lifting: In order to expose opportunities for useful reductions such as β_{let} we must provide lifting rules for the syntactic constructs in our language. These rules move nested block structures outside the construct, allowing (for example) internal λ abstractions to be juxtaposed with their arguments.

$$(\{S \text{ in } e\}\ e_2) \quad \longrightarrow \quad \{\ S' \text{ in } e'\ e_2\ \} \qquad\qquad (\text{LiftFun})$$

For example, the following term must be lifted in order to perform application:

$$\begin{array}{l} \{\ y\ =\ 17\ \texttt{in}\ (\lambda\ \texttt{x}\ .\ x\ +\ y)\}\ 5 \\ \quad \longrightarrow\ \ \{\ y\ =\ 17\ \texttt{in}\ (\lambda\ \texttt{x}\ .\ x\ +\ y)\ 5\} \end{array}$$

2.2.2 *Blocks*

Instantiation (Substitution): Blocks are used by λ_C to perserve sharing of unreduced expressions. During reduction we need to instantiate an identifier's use with its definition. We cannot instantiate arbitrary expressions, as this might replicate work and destroy sharing. We identify two subsets of λ_C terms which are substitutable, values (V) and identifiers (x), together known as simple expressions (SE); these can be found at the bottom of Figure 1. Given $x = a$ in a block, *where a is a simple expression*, the instantiation rules specify that we can replace uses of x by a:

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

The rule (Inst3) encodes self-instantiation, and allows us to partially unroll a cyclic structure or a recursive function; for example:

$$x = \underline{\mathtt{cons}}(1,x) \quad \longrightarrow \quad x = \underline{\mathtt{cons}}(1,\underline{\mathtt{cons}}(1,x))$$

We omit the definitions of contexts $C[\ ]$ and $SC[\ ]$ (expressions and statements, respectively, with holes into which we can plug an expression) as there are no restrictions on *where* substitutions may occur. The calculus thus permits infinite unfolding of cyclic terms; we leave it to reduction strategies to control this.

Structural rules for blocks: We must also provide lifting rules for blocks. We refer to the lifting of an already-named block as "flattening".

$$\begin{array}{lllr} x = \{\ S \text{ in } e\ \} & \longrightarrow & x = e'\ ;\ S' & \text{(Flat)} \\ \{\ S_1 \text{ in } \{\ S_2 \text{ in } e\ \}\} & \longrightarrow & \{\ S_1\ ;\ S_2' \text{ in } e'\ \} & \text{(LiftB)} \end{array}$$

As with other lifting rules, these produce new juxtapositions that trigger other reductions. For example:

$$\mathtt{f\ =\ \{}\ S_1\ \mathtt{in}\ \lambda x.e_1\ \mathtt{\};\ x\ =\ f\ a;} \quad \xrightarrow{Flat} \quad \mathtt{f\ =\ }\lambda x.e_1'\mathtt{;}\ S_1'\mathtt{;\ x\ =\ f\ a;}$$

which then allows us to instantiate `f` by $\lambda x.e_1'$ in the application.

2.2.3 *Conditionals, primitives, and data structures*

Conditionals and delta rules: As usual, we regard constants (CN_0) as already evaluated, and use the following standard rules for conditionals and primitives:

$$\begin{array}{lllr} \mathrm{PF}_k(v_1,\cdots,v_k) & \longrightarrow & w & (\delta) \\ \mathtt{Cond(True,}\ e_1\mathtt{,}\ e_2\mathtt{)} & \longrightarrow & e_1 & \text{(CondT)} \\ \mathtt{Cond(False,}\ e_1\mathtt{,}\ e_2\mathtt{)} & \longrightarrow & e_2 & \text{(CondF)} \end{array}$$

$$\texttt{Cond(}\{S \texttt{ in } e\}\texttt{, } e_1\texttt{, } e_2\texttt{)} \longrightarrow \{\ S' \text{ in } \texttt{cond(}e'\texttt{, } e_1\texttt{, } e_2\texttt{)}\} \qquad \text{(LiftCond)}$$

$$PF_k(e_1, \cdots \{\ S \texttt{ in } e\}, \cdots e_k) \longrightarrow \{\ S' \text{ in } PF_k(e_1, \cdots e', \cdots e_k)\} \qquad \text{(LiftPF)}$$

The (δ) rule given above is simply a shorthand for a family of rules. For example, when faced with the expression `+(17,6)` we reduce the addition to `23`. We consider projection operators to be primitive, but give a precise rule as they are the only primitives which operate on structured data:

$$\texttt{proj}_j\texttt{(}\underline{\mathrm{CN}_k}\texttt{(}a_1\texttt{, } \cdots\texttt{, } a_j\texttt{, } \cdots\texttt{))} \longrightarrow e_j \qquad (\delta_{\mathrm{Proj}})$$

Note also that we will not lift expressions from the arms of a conditional, nor from λ bodies. While safe in λ_C, such rules would be incorrect in λ_B and λ_S because they allow the observation of nontermination or side effects that would normally be protected by those contexts.

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

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

Here, e_j need not be a simple expression. The resulting "$\underline{\mathrm{CN}_k}()$" term has identifier arguments; 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, one which 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 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 recursive blocks make syntactic comparisons futile:

Proposition 2.1 (Non-confluence) (Ariola and Klop [7]) *λ_C is not confluent.*

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

$$\{\ \texttt{x = }\underline{\texttt{Cons}}\texttt{(1, y); y = }\underline{\texttt{Cons}}\texttt{(2, x) in x}\ \}$$

[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.

$$
\begin{array}{lcl}
T_P & ::= & \Omega \quad | \quad \lambda \quad | \quad \mathrm{CN}_0 \quad | \quad \mathrm{CN}_k(T_{P1}, \cdots T_{Pk})
\end{array}
$$

$$
\begin{array}{lcll}
\multicolumn{4}{l}{\mathrm{Info} : E \to T_P} \\
\mathrm{Info}[\![\{\ S \ \texttt{in}\ E\ \}]\!] & = & \mathrm{Info}[\![E]\!] & \\
\mathrm{Info}[\![\lambda x.E\ ]\!] & = & \lambda & \\
\mathrm{Info}[\![\mathrm{CN}_0\ ]\!] & = & \mathrm{CN}_0 & \\
\mathrm{Info}[\![\underline{\mathrm{CN}_k}(x_1, \cdots x_k)]\!] & = & \mathrm{CN}_k(\mathrm{Info}[\![x_1]\!], \cdots \mathrm{Info}[\![x_k]\!]) & \\
\mathrm{Info}[\![E]\!] & = & \Omega & \textit{otherwise}
\end{array}
$$

$$
\begin{array}{lcll}
\Omega & \sqsubseteq & t & \\
t & \sqsubseteq & t & \\
\mathrm{CN}_k(t_1, \cdots, t_j \cdots, t_k) & \sqsubseteq & \mathrm{CN}_k(t_1, \cdots, t'_j \cdots, t_k) & \text{if } t_j \sqsubseteq t'_j
\end{array}
$$

Fig. 3. **Information content for terms of λ_C**

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 information content" or Info of a term. We then discuss observations of a term after either instantiation, yielding the printed form Print, or after progressive reduction, yielding Print*.

2.3.1 *Info: instantaneous observable information in a term*

To handle cyclic terms, we define printing in several stages: the Info function captures the immediate structure of the term, and Print unfolds cyclic or partially-instantiated terms. Info$[\![e]\!]$ generates trees T_P, preserving constants and constructors. Applications, conditionals and primitives are replaced by the symbol "Ω"; λ abstractions are replaced by the symbol "λ" (the body is ignored). The syntax of trees T_P and the rules for computing Info can be found in Figure 3. We also give the ordering used to compare trees T_P; the ordering implies that information content increases monotonically with reduction:

Proposition 2.2 (Info Monotonicity) *If* $e \longrightarrow\!\!\!\!\!\rightarrow e_1$ *then* $\mathrm{Info}[\![e]\!] \sqsubseteq \mathrm{Info}[\![e_1]\!]$.

λ_C is not confluent, but two reduction paths will never head off to different results:

Proposition 2.3 (Confluence with respect to Info) *If $e \twoheadrightarrow e_1$ and $e \twoheadrightarrow e_2$, then there exists e_3 such that $e_1 \twoheadrightarrow e_3$ and* $\text{Info}[\![e_2]\!] \sqsubseteq \text{Info}[\![e_3]\!]$.

Thus, while the information content of any pair of reduction paths cannot necessarily be brought together, the information can always be made to converge to an ascending chain.

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

Definition 2.4 (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 2.5 (Fundamental theorem of graph reduction in λ_C) (Ariola and Blom [5]) *If $e \twoheadrightarrow e_1$ in λ_0, then $\exists e_2$ such that $e \twoheadrightarrow e_2$ in λ_C, and* $\text{Info}[\![e_1]\!] \sqsubseteq \text{Info}[\![e_2]\!]$.

2.3.2 Print: comparing cyclic terms

In an actual language implementation, we may consider a computation to have terminated even if it has a cyclic structure which can be unwound using instantiation (otherwise cyclic terms would serve no useful purpose in our calculus). We must therefore be careful about how termination is defined. To do so, we define a notion called Print which can be used to "unfold" a cyclic term. Conceptually, we are interested in the information content of a completely instantiated version of the provided term. Cyclic terms allow infinite instantiation, so we express Print as the downward-closed set of information reachable by instantiation:

Definition 2.6 (Print) $\text{Print}[\![e]\!] = \{t \mid e \overset{Inst}{\twoheadrightarrow} e_1, t \sqsubseteq \text{Info}[\![e_1]\!]\}$.

These downward-closed sets are ordered by containment ($\subseteq$). Note that because the ordering on T_P is bounded below by Ω, this construction is the usual ideal completion by downward closure, and acyclic terms will have a finite Print with a unique upper bound.

Using Print we can distinguish infinitely generated terms from cyclic terms. Consider the following examples:

```
{ x = Cons(1,x) in x }          { f = λn. Cons(1, f n)  in f 0 }
```

The example on the left is stable—its Print value is the (downward closure of the) infinite list of ones. The example on the right generates a slightly longer list of ones at each iteration, but will never reach a stable value. Thus, we call a term stable when its Print no longer changes under reduction:

Definition 2.7 (Stable term) A term e is *stable* if when $e \longrightarrow\!\!\!\!\!\rightarrow e_1$, $\text{Print}[\![e]\!] = \text{Print}[\![e_1]\!]$.

It is undecidable in general whether a term is stable. We can, however, identify a syntactically testable subset of terms which are certainly stable:

Definition 2.8 (Terminated term) 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 term contains no applications, conditionals, *etc.*, unless they are inside a λ-abstraction.

Proposition 2.9 (Stability of terminated terms) *If $e \in ET$, then e is stable.*

We expect stable terms not in ET to contain some ongoing computation which will not affect their result. We can construct such a computation using a function which loops:

```
loop = λx.loop x;
```

The result of looping can simply be ignored, yielding a stable term with infinite computation:

```
{  y = loop 0
  in 5 }
```

2.3.3 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 reducts. We obtain $\text{Print}^*[\![e]\!]$ by looking at the outcomes of reduction sequences which start at e and produce maximal information.

Definition 2.10 (Reduction Sequences)
$\text{RS}(e) = \{s | s_0 = e, s_{i-1} \longrightarrow s_i, 0 < i < |s|\}$.

We are interested in reduction sequences which make progress toward some outcome.

Definition 2.11 (Progressive Reduction Sequences)
$\text{PRS}(e) =$
$\quad \{s | s \in \text{RS}(e), \exists i. \forall j > i, t. s_j \longrightarrow\!\!\!\!\!\rightarrow t \Rightarrow \exists k > j. \text{Print}[\![t]\!] \subseteq \text{Print}[\![s_k]\!]\}$.

As we will see in §4.2, this definition will need to be extended to reason about nondeterminism.

Print* contains the Print limits of all the progressive reduction sequences:

Definition 2.12 (Print*)
$\text{Print*}[\![e]\!] = \{\bigcup_i \text{Print}[\![s_i]\!] | s \in PRS(e)\}$.

There are subtleties to our definition of progress. First, we must ensure that our deterministic calculus only leads to a single outcome. This can be difficult:

```
{  f = λn. 5
   in f (loop 0) }
```

There are reduction sequences for this term which never complete the evaluation of `f`, focusing instead on `loop`. In spite of the fact that $\text{Print}[\![s_i]\!]$ will be Ω for every term in that sequence, we can always reach the outcome 5; thus, such a sequence has failed to make progress.

There are, however, reduction sequences which produce infinite results during infinite evaluation. Any reduction sequence for the following term has this property:

```
{ f = λn. Cons(1, f (n+1)) in f 0 }
```

Such sequences continually generate more and more information about a term, and we thus consider them to be making progress.

Using the definition of Print*, we can show that when the calculus is confluent, the outcome of reduction is unique:

Proposition 2.13 (Print* preservation in λ_C) *In λ_C, if $e \longrightarrow\!\!\!\!\!\rightarrow e_1$ then* $\text{Print*}[\![e]\!] = \text{Print*}[\![e_1]\!]$. $\text{Print*}[\![e]\!]$ *is a singleton set.*

3 λ_B: adding Barriers to λ_C

We now add a control mechanism called the barrier, which sequences statements in a block. λ_B remains purely functional (no side effects). Barriers get their practical motivation from controlling side effects (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 incrementally.[3] The new syntax is:

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

Intuitively, S_1 (the pre-region) must terminate before S_2 (the post-region) is visible to the surrounding context (no instantiation of terms in S_2 will

[3] In a previous paper [3] barrier semantics were given by translation into a barrier-free calculus. The present paper gives a direct semantics which is different from the semantics captured by the older work; several pathological cases have been cleaned up in the evolution to the present system.

be possible). The sequencing involves only these two subterms. Our static recursive scoping rule for 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:

$$((H\ ;\ S_1)\ \texttt{---}\ S_2) \quad\equiv\quad H\ ;\ (S_1\ \texttt{---}\ S_2) \qquad \text{(BAR equivalence)}$$

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

$$\epsilon\ \texttt{---}\ S \quad\longrightarrow\quad S \qquad\qquad \text{(BAR discharge)}$$

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

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

A terminated term in λ_B contains no barriers unless they are inside a λ or conditional. Thus, our definition of a terminated term does not change.

Barriers make it possible to observe non-termination, even when a nonterminating value is never used. When placed before a barrier, the left-hand term below will allow the barrier to discharge, whereas the right-hand term will not:

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

All λ_C propositions hold in λ_B. It is comparatively simple to show the associativity of barriers; we therefore do not include it as an equivalence in its own right:

Proposition 3.1 (Barrier associativity) *In all contexts,*
$S_1\texttt{---}(S_2\texttt{---}S_3) = (S_1\texttt{---}S_2)\texttt{---}S_3$.

Similarly, we expect barriers to be more restrictive than parallel grouping:

Proposition 3.2 (Barriers reduce results) *Every reduction in $C[S_1\texttt{---}S_2]$ can be modeled by a reduction in $C[S_1;\ S_2]$,* i.e.,
$\forall x \in \mathit{Print}^*[\![C[S_1\texttt{---}S_2]]\!].\exists y \in \mathit{Print}^*[\![C[S_1;\ S_2]]\!].x \subseteq y$.

The postregion of a barrier will have no effect on the computation until the barrier is discharged. Indeed, we can postpone any reduction in the postregion as long as reductions remain in the preregion.

Proposition 3.3 (Postregion postponement)
If $C_1[S_1\texttt{---}S_2] \twoheadrightarrow C_3[S_3\texttt{---}S_4]$ where the barrier is the same in both terms, then there exists a context C_2 such that
$C_1[S_1\texttt{---}S_2] \twoheadrightarrow C_2[S_3\texttt{---}S_2] \twoheadrightarrow C_3[S_3\texttt{---}S_4]$.

Expressions:

$$
\begin{array}{lcll}
E & ::= & x \mid E\ E \mid \lambda x.E \mid \{\ S \texttt{ in } E\ \} \mid \texttt{Cond}(E,E,E) & \\
 & \mid & \mathrm{PF}_k(E_1,\cdots,E_k) \mid \mathrm{CN}_0 \mid \mathrm{CN}_k(E_1,\cdots E_k) & \\
 & \mid & \texttt{allocate()} & \text{cell allocation}
\end{array}
$$

Non-initial expressions:

$$
\begin{array}{lcll}
 & \mid & \underline{\mathrm{CN}_k}(SE_1,\cdots SE_k) & \\
 & \mid & \texttt{O}_j & \text{cell identifiers}
\end{array}
$$

Statements (in blocks):

$$
\begin{array}{lcll}
S & ::= & \epsilon \mid x = E \mid S\ ;\ S \mid S \texttt{ --- } S & \\
 & \mid & \texttt{Store}(E,E) & \text{store}
\end{array}
$$

Non-initial statements:

$$
\begin{array}{lcll}
 & \mid & \texttt{allocator} & \text{storage allocator} \\
 & \mid & \texttt{empty}(\texttt{O}_j) \mid \texttt{error}(\texttt{O}_j) \mid \texttt{full}(\texttt{O}_j,V) & \text{cell states}
\end{array}
$$

Primitives and constants and constructors:

$$
\begin{array}{lcll}
\mathrm{PF}_1 & ::= & \texttt{negate} \mid \texttt{not} \mid \texttt{Proj}_1 \mid \texttt{Proj}_2 \mid \cdots & \\
 & \mid & \texttt{I-Fetch}(E) \mid \texttt{M-Fetch}(E) & \text{fetch operations} \\
 & & & \\
V & ::= & \lambda x.E \mid \mathrm{CN}_0 \mid \underline{\mathrm{CN}_k}(x_1,\cdots x_k) & \\
 & \mid & \texttt{O}_j & \text{cell identifiers} \\
H & ::= & x = V \mid H\ ;\ H & \\
 & \mid & \texttt{allocator} & \text{storage allocator} \\
 & \mid & \texttt{empty}(\texttt{O}_j) \mid \texttt{error}(\texttt{O}_j) \mid \texttt{full}(\texttt{O}_j,V) & \text{cell states}
\end{array}
$$

Fig. 4. **Syntax of λ_S**

Until a barrier has discharged, the contents of its postregion do not matter:

Proposition 3.4 (Postregion irrelevance) *If* $C_1[S_1\texttt{---}S_2] \twoheadrightarrow C_2[S_3\texttt{---}S_2]$ *where the barrier is the same in both terms, then for every statement S such that* $\mathrm{BV[S]} = \mathrm{BV[S_2]}$ *there is a reduction* $C_1[S_1\texttt{---}S] \twoheadrightarrow C_2[S_3\texttt{---}S]$.

In particular, we can replace the postregion with one which binds all its variables to bottom. Note that this proposition could be strengthened to allow S to define other variables not visible in the context and not defined by S_1.

4 λ_S: adding synchronizing data structures

λ_S is the full calculus, obtained by adding updatable memory cells that are dynamically allocated. Each cell can be in a full or empty state; an operation on an empty cell will be blocked until the cell becomes full. Our formulation preserves small-step semantics (local rules) for fine-grain parallelism: there is no global "store", and cell contents are embedded in the term.

The syntax of λ_S is shown in Figure 4. `allocate()` produces a new `empty` cell named by a cell identifier $\texttt{O}_j$. `Store`, `I-Fetch` and `M-Fetch` are cell operations that depend on and affect the cell state. Just before execution, we introduce a "`allocator`" term to represent the pool of memory managed by the storage allocator:

$$e \quad \longrightarrow \quad \{\ \texttt{allocator in}\ e\ \} \qquad\qquad (e \text{ is the entire program})$$

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 to the class of values. We extend our definition of quiescent terms H to include statements indicating the state of the heap, referring to the whole class H as "heap terms". The syntactic equivalence rules (α-renaming, semicolons, barriers) remain unchanged.

4.1 λ_S Reduction Rules

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

$$\texttt{allocator;}\ x\ \texttt{= allocate()} \quad \longrightarrow \quad \texttt{allocator; empty(O}_j\texttt{);}\ x\ \texttt{= O}_j$$
$$\text{where } \texttt{O}_j \text{ is a new object identifier} \qquad \text{(Alloc)}$$

The `allocator` term remains unchanged; it merely enables allocation.[4] Because there are no rules to move the `allocator` term into λ abstractions, 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 `allocator` terms at the start. It can even be relaxed to an unlimited degree of parallelism by adding the reduction rule:

$$\texttt{allocator} \longrightarrow \texttt{allocator ; allocator}$$

(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:

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

The argument terms must be a cell identifier and a simple expression, not

---

[4] We require an oracle to generate our object indices for us. Note, however, that it would be straightforward to encode the index j in the `allocator` term, in which case the term *would* change upon allocation. Indeed, we can use a variation of Gödel numbering to allow the pool to be split. We ignore such machinery as it gratuitously complicates the presentation. It is, if anything, simpler than similar concerns that can be raised about renaming in λ calculi.

arbitrary expressions, *i.e.,* a cell can only store a simple expression. 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:

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

I-Fetch leaves the cell unchanged, allowing further reads; M-Fetch empties the cell, requiring another `Store` before further reads. This captures the "I-structure" and "M-structure" synchronization behavior in pH. These rules make λ_S nondeterministic.

New lifting rules:

$$
\begin{array}{lcll}
\texttt{Store(}\{S\ \texttt{in}\ e\}\texttt{,}e_2\texttt{)} & \longrightarrow & S'\ \texttt{;}\ \texttt{Store(}e'\texttt{,}e_2\texttt{)} & \text{(LiftM1)} \\
\texttt{Store(}e_1\texttt{,}\{S\ \texttt{in}\ e\}\texttt{)} & \longrightarrow & S'\ \texttt{;}\ \texttt{Store(}e_1\texttt{,}e'\texttt{)} & \text{(LiftM2)}
\end{array}
$$

4.2 Properties of λ_S

The definitions of Info and Print remain unchanged. Info monotonicity (Prop. 2.2) continues to hold, but Info confluence (Prop. 2.3) does not, as multiple outcomes can be obtained from a single term due to nondeterminism.

To better explain properties of the nondeterministic calculus, we will make extensive use of the function `choose`:

```
choose = λx. {  m = allocate();
                 Store0(m, True);
                 ( x = M-Fetch1(m)
                   ---
                   Store1(m, False));
                 ( y = M-Fetch2(m)
                   ---
                   Store2(m, True));
                in y }
```

When `choose` is invoked, the two `M-Fetch` operations may execute in either order: if $\texttt{M-Fetch}_1$ executes first, the result of the block will be `False`; if $\texttt{M-Fetch}_2$ executes first, the result will be `True`. The `choose` function thus nondeterministically produces either `True` or `False`. No deterministic strategy can produce both outcomes.

Introducing non-determinism has some worrisome implications for Print*. Consider the following term:

```
{  f = λn. Cond(choose 100, f n, 5);
  in f 0 }
```

If the outcome `True` is produced consistently, computation will never terminate. At the moment, we reject such a reduction sequence as failing to make progress; however, it can fail to achieve the outcome 5 only after repeated non-deterministic choice, and not because some part of the term is being gratuitously ignored. This distinction is important when we discuss reduction strategies in §6.

In order to capture such nontermination, we would like to consider reduction sequences which always contain unresolved non-determinism. Unfortunately, because nondeterminism is implicit in the manipulation of memory cells we must instead settle for a conservative approximation:

Definition 4.1 (Potential unbounded nondeterminism) A reduction sequence s is said to contain potential unbounded nondeterminism if $\forall i. \exists O_n, j, k. i < j < k \wedge s_j \stackrel{Store}{\longrightarrow} s_{j+1} \wedge (s_k \stackrel{Store}{\longrightarrow} s_{k+1} \vee s_k \stackrel{StoreErr}{\longrightarrow} s_{k+1})$, where the Store or StoreErr reductions all refer to the same location O_n.

Only a single outstanding store to a location is actually necessary for computation based on that location to be nondeterministic if the location already contains a value. However, note that a single instance of nondeterminism can easily be resolved; in order for unbounded nondeterminism to occur, new opportunities for nondeterministic choice must forever be created. In order to create such opportunities, we will need to constantly create or reuse locations in the reduction. If new locations are created, two writes to them will be required to realize nondeterminism; if an old location is reused, it will be written multiple times.

It is worth noting that a term may have unbounded nondeterminism, yet still have a finite Print*; the earlier example in this section has unbounded nondeterminism, yet its Print* is $\{\{\Omega\}, \{\Omega, 5\}\}$.

Resolving non-determinism implies we are making progress in computation; therefore the definition of PRS(e) (Def. 2.11) is extended to include reduction sequences which have potentially unbounded nondeterminism:

Definition 4.2 (Progressive Reduction Sequences)
$$\begin{aligned}\mathrm{PRS}(e) = \\ &\{s | s \in \mathrm{RS}(e), (\exists i. \forall j > i, t. s_j \longrightarrow\!\!\!\!\!\rightarrow t \Rightarrow \exists k > j. \mathrm{Print}[\![t]\!] \subseteq \mathrm{Print}[\![s_k]\!]) \quad \vee \\ &\qquad s \text{ contains unbounded nondeterminism}\}.\end{aligned}$$

Print* preservation (Prop. 2.13) no longer holds; performing an `M-Fetch` may preclude certain reductions that would have been possible by doing that `M-Fetch` later. Because there may be multiple possible outcomes of reduction, Print* preservation (Prop. 2.13) must be weakened:

Proposition 4.3 (Print* contraction)
If $e \longrightarrow\!\!\!\!\!\rightarrow e_1$ *then* $\mathrm{Print}^*[\![e_1]\!] \subseteq \mathrm{Print}^*[\![e]\!]$.

All other propositions continue to hold (Stability of terminated terms, Prop. 2.9, and the barrier propositions, Props. 3.1–3.4).

Because I-structure locations are write-once, only a single value can be read from them; thus, in the absence of multiple stores, they are deterministic. We get a hint of this in the following seemingly equivalent examples:

```
{  m = allocate();                  {  m = allocate();
   Store(m,1);                         Store(m,1);
   y = I-Fetch(m);                     y = M-Fetch(m);
   z = I-Fetch(m);                     Store(m,y);
  in y+z };                            z = M-Fetch(m);
                                      in y+z };
```

Though these examples seem innocuously different, the left-hand example always produces 2 while the right-hand example may produce 2 or result in a multiple store error.

Definition 4.4 (I-structure subset) λ_I is λ_S without the `M-Fetch` syntax and rules.

This subset is important because cells with `Store` and `I-Fetch` operations express the same idea as "communication variables", (*c.f.* "logic variables", "single-assignment variables", *etc.*). If a reduction sequence terminates without a store error, then the printed result is unique:

Proposition 4.5 (Termination is unique in λ_I) *If* $e \longrightarrow\!\!\!\!\!\rightarrow e_1$ *and* $e_1 \in ET$ *then* $\text{Print}^*[\![e]\!] = \{\text{Print}[\![\mathrm{e}_1]\!]\}$.

Furthermore, all reduction sequences produce a store error if any one of them does:

Proposition 4.6 (Store error confluence in λ_I) *If* $e \longrightarrow\!\!\!\!\!\rightarrow C[\text{error}(\mathtt{O_i})]$ *and* $e \longrightarrow\!\!\!\!\!\rightarrow e_1$ *then* $e_1 \longrightarrow\!\!\!\!\!\rightarrow C_1[\text{error}(\mathtt{O_j})]$.

Thus, the nondeterminism in λ_I is benign: if we consider all programs containing any `error()` to be equivalent, then λ_I produces deterministic results (this is the view taken in the language pH).

5 Kernelization

The λ_S calculus serves two functions. First, it provides detailed semantics for the pH programming language; this has been the focus of this paper thus far. Second, it provides a way to describe and reason about the program transformations performed by a compiler for pH. In the course of compilation many such transformations must occur even between pH source code and a λ_S-like intermediate form: pattern matching syntax must be transformed into conditionals, list comprehensions must be desugared, and so forth. Many of these transformations simplify the program by reducing the complexity of the language being compiled (desugaring falls into this category).

It is useful to consider additional reduction rules that model certain important transformations performed by the compiler. In this section we describe two such reduction rules, which preserve the properties of the calculi described thus far. We then explain how the addition of these rules permits us to kernelize the language and subsequently work with fewer rules and a simpler syntax. By reducing the complexity of the language, we reduce the complexity of its compiler; our kernelization results are thus immediately useful for implementation.

5.1 New reduction rules

Naming Rule: We need to give names to subexpressions in order to discuss most compiler optimizations. Our calculus does not allow naming to be derived from other reductions, in contrast to the calculus given by Ariola and Blom [5] where it is a consequence of garbage collection and instantiation. We thus add a separate rule as follows:

$$\begin{array}{lll} e & \longrightarrow & \{\ t = e \text{ in } t\ \} \quad \text{(Name)} \\ \text{provided} & e \notin \mathit{SE} & \\ & e \text{ is not a block } \{\ S \text{ in } e_1\ \} & \\ & e \text{ does not have a name, } \textit{i.e.,} \text{ is not bound by a statment } x = e & \end{array}$$

The rule itself is very simple: an expression may be replaced by a block which names that expression and returns it as a result:

$$17 + 12 \quad \longrightarrow \quad \{\ t = 17 + 12 \text{ in } t\ \}$$

The first side condition preserves the monotonicity of the calculus. Consider what happens when we name a constant:

$$5 \quad \longrightarrow \quad \{\ t = 5 \text{ in } t\ \}$$

The original constant in this example has the information content 5, whereas after naming its information content is Ω. Non-monotonic information content means we can no longer view our terms as making progress towards some final result—we would somehow have to allow for "regressive" naming steps.

It is unnecessary to name an expression which is already named; similarly, a block should be lifted rather than named. Thus, we impose the second and third side conditions. These rule out naming the subexpressions on the right-hand side of the (Name) rule itself, and thus prevent naming from being applied *ad infinitum*.

Argument Lifting: In addition to naming, we add a lifting rule for the argument of an application, the only "exposed" part of a term which does not

Kernel Terms:

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

Fig. 5. **Syntax of kernel terms**

already have one:

$$(e_1\ \{S \text{ in } e\}) \quad \longrightarrow \quad \{\ S' \text{ in } e_1\ e'\ \} \quad \text{(LiftArg)}$$

Because β_{let} implicitly names the argument to an application during reduction, this rule is redundant. However, no evaluation rule depends upon the exact structure of the argument to an application, and thus adding it will cause no harm.

It must be emphasized that neither argument lifting nor naming compromises the expressive power of our calculus. All the properties for all the calculi discussed so far continue to hold when the above rules are added. Indeed, the calculi are equivalent:

Proposition 5.1 (Name and LiftArg add no power) *If* $e_1 \twoheadrightarrow e_2$ *in* λ_S *with (Name) and (LiftArg), then* $e_1 \twoheadrightarrow e_3$ *in* λ_S *where* $\mathrm{Info}[\![e_2]\!] \sqsubseteq \mathrm{Info}[\![e_3]\!]$.

5.2 Kernel Language

An initial term is kernelized by repeated application of naming and lifting:

Definition 5.2 (Kernelization)
1. Apply the (Name) rule until a Name-normal form is reached.
2. Apply the lifting rules until no further lift reductions are possible.

Kernelization gives rise to a kernel langauge, whose grammar is given in Figure 5 (compare with Fig. 4).

Kernelization has some nice (and easily proved) properties:

Proposition 5.3 (Kernelization confluence) *Kernelization is confluent up to syntactic equivalence (α-renaming, properties of "`;`" and "`---`") and is strongly normalizing (always terminates).*

Proposition 5.4 (Kernelization is static) *A kernel expression remains a kernel expression after further reduction.*

Proposition 5.5 (Expressivity of kernel language) *If* e_k *is the kernelized version of* e, *then* $\mathrm{Print}^*[\![e]\!] = \mathrm{Print}^*[\![e_k]\!]$.

Once we have kernelized, the kernel syntax dictates that no further lifting or naming can ever occur. We thus refer to lifting and naming as static rules or kernelization rules; they can be applied during kernelization and never used again. The remaining reduction rules, by contrast, are called dynamic rules: β_{let}, Inst2, Flat, CondT, CondF, δ, Cons, BAR discharge, Alloc, Store, StoreErr, I-Fetch, M-Fetch.

5.3 Restricting reduction to outer contexts

We can also restrict the calculus by performing reductions only where computations are "exposed" (avoiding λ-bodies, arms of conditionals, and post-regions), and by avoiding the repeated unfolding of cyclic data structures. Restricting evaluation simply means limiting the contexts[5] where an identifier may be instantiated and where reductions may occur:

Limited instantiation contexts:

$$
\begin{array}{lcl}
C[\,] & ::= & [\,] \quad | \quad C[\,]\ E \quad | \quad \texttt{Cond}(C[\,],E,E) \quad | \quad \mathrm{PF}_k(\cdots,C[\,],\cdots) \\
SC[\,] & ::= & x = C[\,] \quad | \quad SC[\,]\ ;\ S \quad | \quad SC[\,]\ \texttt{---}\ S \\
 & | & \texttt{Store}(C[\,],\ E) \quad | \quad \texttt{Store}(E,\ C[\,])
\end{array}
$$

Limited reduction contexts:

$$
\begin{array}{lcl}
C[\,] & ::= & [\,] \quad | \ \{\ SC[\,]\ \texttt{in}\ E\ \} \quad | \ \{\ S\ \texttt{in}\ C[\,]\ \} \\
 & | & C[\,]\ E \quad | \quad \texttt{Cond}(C[\,],E,E) \quad | \quad \mathrm{PF}_k(\cdots,C[\,],\cdots) \\
SC[\,] & ::= & [\,] \quad | \quad x = [\,] \quad | \quad SC[\,]\ ;\ S \quad | \quad SC[\,]\ \texttt{---}\ S \\
 & | & \texttt{Store}(C[\,],\ E) \quad | \quad \texttt{Store}(E,\ C[\,])
\end{array}
$$

Note that these restricted contexts become simpler still if we consider only the kernel language. Either way, restricting the calculus in this way has no effect on its power:

Proposition 5.6 (Expressivity of restricted instantiation) $\mathrm{Print}^*[\![e]\!] = \mathrm{Print}^*_{\mathrm{restricted}}[\![e]\!]$, *where the latter is defined using only restricted reduction contexts.*

6 Reduction Strategies for λ_S

A reduction strategy is a method for choosing among many available redexes in a calculus; we can characterize a strategy σ as a reflexive subrelation of reduction ($\twoheadrightarrow$). Traditionally, a standard reduction strategy σ produces a term with at least as much information as is obtainable from any other strategy, so that when $e \twoheadrightarrow e_1$ in the calculus, there is an e_2 such that $e \overset{\sigma}{\twoheadrightarrow} e_2$ and $\mathrm{Print}[\![e_1]\!] \subseteq \mathrm{Print}[\![e_2]\!]$. This notion needs elaboration in the presence of

[5] 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.

nondeterminism.

Consider the following example:

```
{  f = λn. Cond(choose 100, 17, 5);
    x = loop 0
   in f x }
```

This has only two outcomes, 17 and 5. In particular, $\{\Omega\}$ is not in its Print*. No deterministic strategy can produce both of these outcomes; we settle for a definition of standard reduction which produces at least one of these outcomes but not $\{\Omega\}$. We need the following definitions to identify precisely the outcomes according to a strategy σ:

Definition 6.1 (Reduction sequence in σ)
$\mathrm{RS}_\sigma(e) = \{s | s \in \mathrm{RS}(e), \forall i. e \xrightarrow{\sigma}\!\!\!\!\twoheadrightarrow s_i\}$

Definition 6.2 (Complete reduction sequence in σ)
$\mathrm{CRS}_\sigma(e) = \{s | s \in \mathrm{RS}_\sigma(e),\ s \text{ is not a proper subsequence of } s' \in \mathrm{RS}_\sigma(e)\}$

Note that RS_σ must contain $\mathrm{Print}[\![e]\!]$; therefore RS_σ and thus CRS_σ are nonempty.

Definition 6.3 ($\mathbf{Print}^*_\sigma$) $\mathrm{Print}^*_\sigma[\![e]\!] = \{\bigcup_i \mathrm{Print}[\![s_i]\!] | s \in \mathrm{CRS}_\sigma(e)\}$

We define standardization as follows:

Definition 6.4 (Standardization) A reduction strategy σ is standard when for all e, $\mathrm{CRS}_\sigma(e) \subseteq \mathrm{PRS}(e)$.

We refer to a strategy as strongly standard if reduction according to its constraints can produce any of the outcomes allowed by the calculus—that is, none is ruled out *a priori* by the strategy.

Definition 6.5 (Strong standardization) A reduction strategy σ is strongly standard when $\mathrm{Print}^*[\![e]\!] = \mathrm{Print}^*_\sigma[\![e]\!]$ for all e.

This definition is correct for deterministic calculi; however, because our definition of potential unbounded nondeterminism is approximate, we will often believe that a term has more possible outcomes than it actually does. In order to properly define strong standardization for a non-deterministic calculus, we must therefore devise a way to spot nondeterminism precisely. We are not aware of such a characterization, and it remains to be seen if it can even exist.

Even a standard strategy is non-trivial: if we throw in an (irrelevant) statement the strategy must not get stuck, as we saw in `loop` in the earlier example. For this reason simply restricting the reduction context does not lead to a standard strategy. For a strategy to be standard, *all* of its reductions must make progress. Such a strategy will avoid a non-terminating path if there is a terminating path that can be reached purely by deterministic reduction;

non-deterministic choice may still make infinite reduction inevitable, as seen in §4.2.

6.1 The σ_n family of strategies

In this section we describe a family of parallel nondeterministic strategies. Without loss of generality, we restrict our attention to the kernelized language. We limit instantiation and reduction to the outer contexts described in the previous section; allowing reduction elsewhere can only jeopardize standardization. In addition, 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.

Fairness: Having restricted the reduction context, we must ensure that no redex in a restricted context is ignored indefinitely. We avoid this by ensuring 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:

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

Definition 6.6 (σ_n reduction strategies for λ_S)
While the term is not a terminated term:

(i) Decorate all λ abstractions with the positive integer $n > 0$.

(ii) 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. Still, for any n, each invocation of this step will terminate. The choices in this step represent a kind of nondeterminism. For λ_C and λ_B, this nondeterminism is benign—there is still a unique outcome, and this strategy will find it, even if $n = 1$. In λ_S, however, any particular choice of n, no matter how large, can force the program's nondeterminism to be resolved too early, thereby ruling out some outcomes. We use the following variant of our choose function for reasoning about strategy:

```
delay = λd. λv. Cond(n=0, v, delay(d-1,v));
  biasedChoose = λd. {  m = allocate();
                        Store0(m, True);
                        ( x = M-Fetch1(delay(d,m))
                          ---
                          Store1(m, False));
                        ( y = M-Fetch2(m)
                          ---
                          Store2(m, True));
```

```
                    in y }
```

Now `delay` must be instantiated d times before $\texttt{M-Fetch}_1$ executes. Again, if $\texttt{M-Fetch}_1$ executes before $\texttt{M-Fetch}_2$, the result is `False`, but if $\texttt{M-Fetch}_2$ executes first, the result is `True`. However, σ_n forces us to execute $\texttt{M-Fetch}_2$ first when $n < d$, so the only possible outcome is `True`.

Biasing choice in this way has a major effect. In the following example, σ_{99} will not terminate, and will produce an infinite list of ones:

```
{  f = λn. Cond(biasedChoose 100, Cons(1, f (n+1)), Nil)
   in f 0 }
```

Similarly, the following example, which can produce any integer as an answer, will neither terminate or produce an answer under σ_{99}. This is the justification for considering unbounded nondeterminism to be productive even when it fails to yield increasing or stable results:

```
{  f = λn. Cond(biasedChoose 100, f (n+1), n)
   in f 0 }
```

We characterize the behavior of the σ_n as follows:

Proposition 6.7 (σ_n is standard for λ_S) *The strategy σ_n is standard for λ_S.*

Proposition 6.8 (σ_n is strongly standard for deterministic calculi) *The strategy $\sigma_n, n \geq 1$, is strongly standard for both λ_C and λ_B.*

Finally, we noted earlier that while λ_I is not confluent, it does exhibit confluence in the absence of store errors. We thus expect similar behavior from any strategy:

Proposition 6.9 (σ_n standardization for λ_I) *σ_n is standard for λ_I; in addition, it is strongly standard for terms in which no store errors can occur.*

6.2 Other strategies

Consider a variant of σ_n, where the strategy non-deterministically chooses the n used to annotate the λ's at each step. By choosing a sufficiently large n at each step we ensure that every possible outcome can be reached by some choice of reductions within the strategy, and we conjecture that this strategy is strongly standard for λ_S. In order to show this we will need a precise characterization of non-determinism and a useful definition of strong standardization.

We have seen examples of programs in λ_S that do not terminate even after producing the answer. For λ_C, the purely functional subset of λ_S, we can devise a minor variant of σ_n which, like a *lazy* reduction strategy, will terminate after all the answers have been produced. A lazy reduction strategy focuses on “outermost” redexes, always picking one that is needed to produce the answer. Ariola and Blom [5] give such a strategy, and show that it is standard for λ_C. Our variant, evaluates only reachable subterms of the expression—that is, those which bind variables appearing in the result or in other reachable subterms. We are free to evaluate as many reachable subterms as we like in parallel, modulo the fairness restrictions imposed by σ_n. The final term may still contain redexes, but further reductions will not affect the printed value (and so termination at this point makes sense). Indeed, this strategy finds answers equivalent to those found by lazy reduction or σ_n.

Finally, there is hope even for the very simple strategy σ_∞, which places no fairness constraints on the instantiation of abstractions but performs reductions only in restricted contexts:

Proposition 6.10 (σ_∞ is standard for terminating terms) *If* $e \longrightarrow\!\!\!\!\!\rightarrow e_1$ *and* $e_1 \in ET$*, then* $e \overset{\sigma_\infty}{\longrightarrow\!\!\!\!\!\rightarrow} e_2$ *and* $\mathrm{Print}[\![\mathrm{e_1}]\!] = \mathrm{Print}[\![\mathrm{e_2}]\!]$.

7 Conclusion

In this paper we have described λ_S, a fine-grained parallel extension of the λ-calculus with recursive blocks, barriers (for sequencing), and synchronized side effects. 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. Side-effects in the form of M-structures allow expression of non-determinism and concurrency primitives. Barrier sequencing also makes raw (unsynchronized) side effects manageable locally without compromising parallelism globally.

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 standard reduction strategies to compute the maximal information in a term even in the presence of unbounded non-dterminism. A simple variation of our strategy subsumes traditional lazy evaluation.

By permitting reductions everywhere, even inside λ-bodies, conditional branches, and below sequential barriers, the calculus provides the flexibility to reason about program transformations and compiler optimizations.

We believe that λ_S 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 or ML extended with futures, and Haskell extended with monads and concurrency primitives. Its simplicity and flexibility should 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 Arvind. Properties of a first-order functional language with sharing. *Theoretical Computer Science*, 146(1–2):69–108, 24 July 1995.

[5] 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.

[6] 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.

[7] 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.

[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.