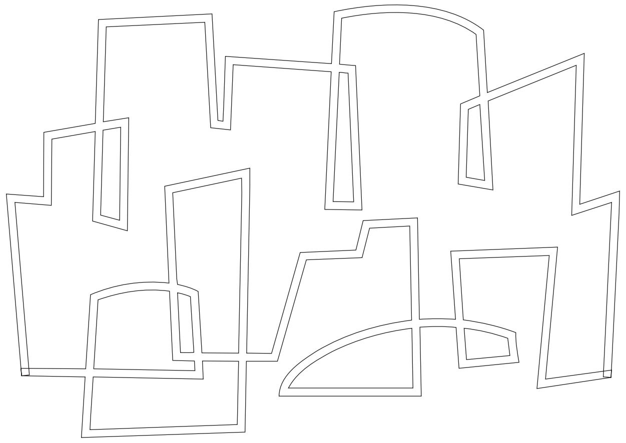
Semantics of Barriers in a Non-Strict, Implicitly-Parallel Language

Shail Aditya, Arvind, Joseph Stoy

Published at FPCA '95 Conference

1995, June

Computation Structures Group
Memo 367



Semantics of Barriers in a Non-Strict, Implicitly-Parallel Language

Computation Structures Group Memo 367
January 12, 1995

Shail Aditya
Arvind
MIT Laboratory for Computer Science
{shail,arvind}@lcs.mit.edu

Joseph E. Stoy
Oxford University Computing Laboratory
joe.stoy@comlab.ox.ac.uk

The research described in this paper was funded in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-92-J-1310.

Semantics of Barriers in a Non-Strict, Implicitly-Parallel Language

Shail Aditya Arvind
MIT Laboratory for Computer Science
{shail,arvind}@abp.lcs.mit.edu

Joseph E. Stoy
Oxford University Computing Laboratory
joe.stoy@comlab.ox.ac.uk

Abstract

Barriers in parallel languages may be used to schedule parallel activities, control memory usage and ensure proper sequentialization of side-effects. In this paper, we present operational semantics of barriers in Id and pH, which are non-strict, implicitly-parallel, functional languages extended with side-effects. The semantics are presented as a translation from a source language with barriers into a kernel language without barriers where the termination properties of an expression are represented explicitly as signals using a weak head-normal form operator ($\mathcal{W}$) and controlled *via* a strict application operator (Sap). We present two versions of the semantics — the first uses purely data-driven, eager evaluation and the second mixes eager evaluation with a demand-driven identifier reference mechanism. We compare and contrast the two for their ability to do resource management and preserve useful semantic properties.

1 Introduction

Data-parallel programming models, as embodied in *Lisp and CM-Fortran, identify well-defined computation and communication phases that are separated by barriers to ensure proper synchronization among the various parallel activities. Some parallel machines like the CM5 and Cray T3D even provide such barrier support in hardware. In this paper we study the use and implementation of barriers under a more general, multi-threaded, MIMD programming model, in which barriers need to be localized to specified subsets of parallel tasks in order to provide fine-grain control over synchronization and resource usage.

Our study is in the context of Id [13] and pH [14], which are non-strict, implicitly-parallel, functional languages extended with I-Structures [5] and M-Structures [7]. Barriers were introduced in Id to provide partial sequencing of M-Structure operations. However, their semantics were not given precisely at that stage [6]. Subsequently, pH was designed to follow the precise definition of Haskell [9] combined with the eager evaluation model and the non-functional extensions of Id. This prompted a formalization of barrier semantics for both Id and pH. The first such attempt was made in [16], both in terms of CSP [8] and (for the functional layer) denotationally. This paper presents a different formalization in terms of graph rewriting systems previously used to describe the semantics of Id [3].

The rest of the paper is organized as follows. In Section 2, we discuss some important characteristics of non-strict, eager computations. We describe the fully parallel evaluation model of Id and pH, differentiating between the *result* and the *termination* of an eager computation, and introduce the concept of control regions. In Section 3, we introduce barriers informally and present several examples of their use in controlling parallelism, managing program resources, and providing fine-grain control over scheduling of side-effects.

In Section 4, we describe the formal framework for barrier semantics by introducing a simple kernel language with parallel reduction semantics and a weak head-normal form operator ($\mathcal{W}$). In Section 5, we describe the concept of termination and propose the first semantics for barriers based on a purely eager, dataflow graph execution model. This is sufficient for both resource management and side-effect control, although it turns out to be a little fragile with respect to program transformations. Section 6 highlights these shortcomings and identifies some ways to fix them. In Section 7, we present the second semantics for barriers, attempting to overcome some of these shortcomings by mixing eager evaluation with lazy identifier referencing. Finally, Section 8 presents the summary and directions for future work.

2 Non-Strict, Eager Evaluation

The Haskell language prescribes a *lazy* evaluation strategy for non-strict computation: only those tasks are evaluated which are required to produce the result. This strategy imposes a strong sequential constraint on the overall computation, although the exact ordering of tasks is decided dynamically. On the other hand, Id and pH languages follow an *eager* evaluation strategy for the same non-strict computation: all tasks execute in parallel, restricted only by the data-dependencies among them. This strategy automatically exposes large amounts of parallelism both within and across procedures. We describe this evaluation model below.

2.1 The Fully Parallel Execution Model

A program in Id consists of an expression to be evaluated within the scope of a set of top-level function and type declarations. Each function is broken up into several threads of computation (the length of the threads is determined in part by the compiler's ability to identify strict regions and in

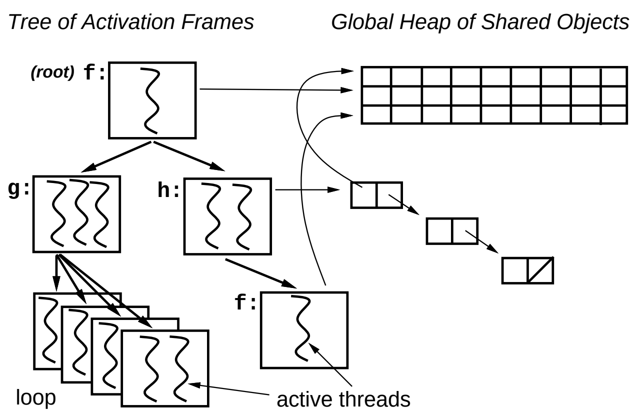


Figure 1: The Fully Parallel Execution Model of Id and pH.

part by the ability of the run-time system and the hardware to exploit the exposed parallelism efficiently).

The parallel execution model of Id and pH is shown pictorially in Figure 1. Each function application allocates an *activation frame* which records function arguments and keeps temporary, local values. Threads belonging to a function share its activation frame and may be active concurrently. The activation frame is deallocated when the function terminates. Executing threads may enable other threads by sending data or synchronization information to their associated activation frame: this characterizes the "data-driven" nature of this execution model.

The program starts by allocating a root activation frame to evaluate the top-level expression. Parallel threads containing function applications give rise to parallel child activations, while loop invocations give rise to multiple parallel iterations. At any time the overall computation is represented by a tree of activation frames as shown in Figure 1, exploiting both intra-procedural and inter-procedural parallelism.

In contrast, under lazy evaluation computation is spawned in parallel only if it is already known to contribute towards the final result. Otherwise, every potentially concurrent task is suspended in a *thunk* immediately upon creation. A "demand-driven" evaluation of the suspended thunks exposes only a small part of the parallel execution tree at any given time. Of course, some of the thunks may never get evaluated under demand-driven evaluation.

All threads participating in the parallel computation share a globally addressable heap. Data structures allocated on the heap may continue to exist even after the function that allocated them has terminated. Such data structures either have to be explicitly deallocated or are garbage-collected when no more references to them remain.

2.2 Result *vs.* Termination

An important aspect of non-strict, eager evaluation is that the production of the result of a computation and its termination may not always coincide. For instance, the resulting `Cons`-cell in the following example[1] may be allocated and returned without waiting for the termination of the computation that fills it. I-Structure synchronization on the `Cons`-cell automatically blocks any attempt to read its contents until it is filled.

Example 1:
```
def enlist f x = (f x) : nil;
```

Under lazy evaluation, a computation proceeds only as far as necessary to produce the requested result, so we never have to worry about the termination of the computation aside from the production of its result. On the other hand, under eager evaluation, several activities may be spawned eagerly that do not immediately contribute towards the result. Therefore, it is semantically important to separate the production of the result of a computation from its overall termination. From now on, we will use the term *Value Semantics* to refer to the result of a computation, and *Termination Semantics* to identify its termination.[2] Semantically, a given computation may give rise to one of the following possible outcomes [1]:

1. **Result with proper Termination** — The computation produces a result and terminates.
2. **Result with improper Termination** — The computation produces a result but does not terminate. The non-termination could be either due to an infinite computation or due to a deadlocked computation.
3. **No Result or Termination** — The computation does not produce a result and does not terminate. Again, the non-termination could be due to infinite computation or deadlock.[3]

For example, the function `f1` below produces an infinite list of `x`'s as the result and terminates properly, `f2` produces the same result but does not terminate because of infinite computation, while `f3` does not produce a result and deadlocks because the `+` operator is strict in its arguments.

Example 2:
```
def f1 x = { a = x:a in a };

def f2 x = { a = x:(f2 x) in a };

def f3 x = { a = a+x in a };
```

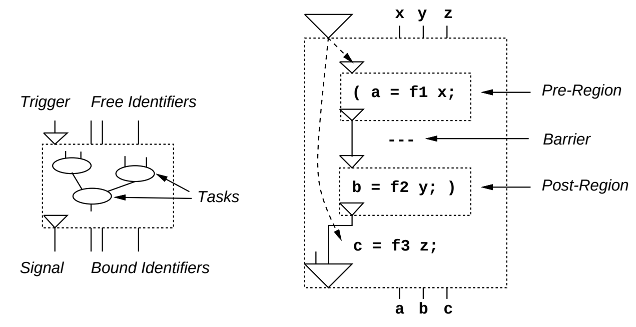
2.3 Control Regions

Another important aspect of eager evaluation is the notion of *control regions*. In a sequential language, the control "flows" through a chain of primitive tasks each of which has a well defined initiation and termination point. Under the fully parallel model of Id, a group of tasks may be initiated in parallel, constituting a *control region* as shown in Figure 2 (a). A control region is informally defined as a set of parallel tasks that always execute together (respecting data-dependencies), and their combined termination may be used as a synchronization event to initiate other control regions. A control region containing a set of parallel tasks requires the values of its free identifiers as input and produces one or

[1] All our examples use the Id syntax [13]. In Id, functions are introduced using the *def* keyword and colon (:) is the infix list constructor.

[2] These correspond to what were called non-strict and strict semantics, respectively, in [16].

[3] The fourth possibility, of proper termination but without producing a result, is not considered to be meaningful and is lumped with the third outcome. Some graph-reduction based semantic definitions [3] do, however, distinguish between these two cases.



(a): A Control Region. (b): Sequencing Regions with Barriers.

Figure 2: Control Regions and Barriers.

more values as output. It also takes in a trigger input and signals the termination of all the tasks contained within it.

As an example, the body of an Id function constitutes a control region which is initiated when the function is applied to all its arguments. The termination of the entire body is determined by the joint termination of each of its sub-computations. Control regions are always properly nested within one another and therefore form a "tree" of parallel activities rather than a sequence. The activation tree in Figure 1 shows the dynamic nesting of procedure control regions. Similarly, each of the branches of a conditional expression forms a control region that is statically nested inside the enclosing region. In this case, only one of the branches is initiated when the predicate is resolved and the termination of the entire expression is determined by the termination of that branch.

3 Barriers under Eager Evaluation

3.1 Barrier Specification

Barriers provide a mechanism to detect the termination of a set of parallel activities enclosed within a control region. A barrier is specified in Id using three or more dashes (`---`) creating two sub-regions within a given control region — one above the barrier called the *pre-region* and the other below the barrier called the *post-region* (see Figure 2 (b)). Intuitively, no computation within the post-region is allowed to proceed until all computations within the pre-region have terminated.[4] The termination of control regions must also respect their nesting, making the barriers *hyper-strict*, *i.e.*, the entire computation subtree spawned from the tasks within a pre-region must terminate before any computation in the post-region is allowed to proceed.

Id also provides the ability to delimit the scope of a control region to an arbitrary subset of the parallel tasks at hand. Unlike data-parallel computation where all barriers are global, barriers in Id may be localized to a specified set of parallel activities by grouping those activities in parentheses. This is also illustrated in the example shown in Figure 2 (b). The activities `(f1 x)`, and `(f3 z)` are initiated in parallel because the post-region of the barrier excludes the activity `(f3 z)` using parentheses. The activity `(f2 y)` is initiated upon termination of `(f1 x)` as prescribed by the barrier. The entire block terminates when all the activities within all sub-regions inside it have terminated. The power and usefulness of barriers in Id stem from the fact that the granularity of the regions being sequentialized is completely under user control.

Figure 2 illustrates another important point: the partitioning of parallel computation in Id into control regions is completely static and textual. The user always has a clear picture of which computations will evaluate and under what circumstances. This is possible only under eager evaluation. Under lazy evaluation, even if some computations are explicitly sequentialized or scheduled using barriers, it is not clear whether they will ever evaluate until something demands their value: there is no natural notion of control region as a group of concurrent activities.

3.2 The Uses of Barriers

In this section we describe the various uses of barriers under the parallel execution model discussed in Section 2.

Controlling Parallelism

As shown in Figure 1, the eager evaluation strategy unfolds the parallel computation into a tree of activation frames. In some "embarrassingly parallel" computations, the potential parallelism is limited not by the data-dependencies but purely by the size of the available frame or heap memory in the machine. An unrestricted unfolding of the activation tree may cause the machine to run out of memory very quickly. In such cases, it is important to "throttle" the parallelism to match the available resources. Barriers allow the programmer (or the compiler) to control the parallelism by sequentializing some parallel activities. For instance, a single barrier in the example below reduces the exponential resource requirement of the `fib` function to a linear requirement proportional to the size of the problem. Of course, it also reduces the parallelism accordingly.

Example 3:
```
def fib n = if n < 2 then n
            else { x = fib (n-1);
                   ---
                   y = fib (n-2);
                  in x+y };
```

Controlling Frame Storage

Sometimes a group of activities that are initiated simultaneously may not exhibit a lot of parallelism because of pre-existing data-dependencies among them. But under eager evaluation, all the initiated activities can grab memory resources and hold on to them for a long time without doing any useful work. Barriers help in scheduling the activities in the appropriate partial order so that the overall resource requirement is reduced. This behavior is illustrated by the following example.

Example 4:
```
def fold1 f z nil = z
  | fold1 f z (x:xs) = fold1 f (f z x) xs;

def fold2 f z nil = z
  | fold2 f z (x:xs) = { y = f z x;
                         ---
```

[4] Actually, all we care about is that the *effect* (computed values and side-effects) of the post-region is not visible from outside until the pre-region has completely terminated.

```
                              in fold2 f y xs };
```

Assuming that f is a strict function, both fold1 and fold2 shown above fold f on a given list of elements from left to right. Because of eager evaluation, the fold1 function rapidly expands into a sequence of frames each of which spawns a child frame to compute the application (f z x). Thus it could use up to $O(n)$ frames for an n-element list even if we optimized fold1 for tail-calls. On the other hand, the fold2 function waits for each child application to finish before moving on to the next list element, and so it uses at most two frames during the entire computation. Note that unlike the case of the fib function in Example 3, no parallelism is lost by inserting this barrier since the computation is already sequential.

Controlling Heap Storage

Barriers are also useful for scheduling the automatic (or programmer-specified) deallocation of heap resources once they are no longer in use. This strategy may be used to manage the heap without paying any extra overhead for garbage collection, as shown in the following example. While traversing a large data structure we keep a local notebook which is released at the end of the traversal.

Example 5:

```
def compute graph =
 { note = make_notebook ();
   result = traverse graph note;
   ---
   @release note;
  in result };
```

Sequentializing Side-Effects

Barriers may be used in parallel M-Structure computations for scheduling side-effects in the desired partial order. Under eager evaluation, all side-effects present within a block are initiated in parallel. Therefore, a barrier may be required to ensure proper interleaving of M-take (read-and-lock) and M-put (write-and-unlock) operations. For instance, a barrier is necessary in the following example to ensure that the mutable array slot is taken before the new value is put back, because there is no explicit data-dependency to control the order otherwise:[5]

Example 6:

```
def replace a i v =
  { x = a![i];
    ---
    a![i] = v;
   in x };
```

Barriers provide the ability to detect the combined termination of a group of parallel update operations without sequentializing them completely. A classic example of this sort is the construction of a histogram:

Example 7:

```
def histogram xs n =
  { a = marray (1,n);
    { for i <- 1 to n do
        a![i] = 0; };
    _ = accum a xs;
    ---
   in a };

def accum a nil = ()
 | accum a (x:xs) =
  { i = hash x (bounds a);
    a![i] = a![i]+1;
    _ = accum a xs; };
```

[5]In Id, *M-take* and *M-put* operations are represented using a slightly modified array indexing syntax a![i]. Indexed expressions on the right-hand-side of a binding are *take* operations while those on the left-hand-side are *put* operations.

The accum function is allowed to unfold automatically, accumulating the frequencies in parallel into the histogram array a using M-Structure operations. No sequentialization is imposed over the accumulations, but the barrier in the function histogram guarantees that all the accumulations have terminated before the histogram array is returned. Note that the barrier ensures that all the computations above the barrier have terminated before returning the array, although actually we are interested only in the termination of all side-effects on the array itself.

Controlling Access to Shared External State

Barriers are also useful in managing activities that share a common external resource or state. Shared peripheral devices, I/O streams, and protected kernel data-structures may constitute some of the external state that user programs may need to control or gain access to. Most functional languages allow only implicit control of such computation *via* monads, continuations, abstract datatypes, or state transformers [10, 12, 15]. On the other hand, in Id such objects are represented explicitly using M-Structures and access to them is controlled using locks and barriers that provide mutual exclusion and explicit sequentialization as necessary. For instance, currently there exist two I/O libraries in Id, sequential I/O [17] and threaded I/O [13], both of which crucially rely on barriers for proper sequentialization.

3.3 Syntactic Properties of Barriers

The definition of barriers in terms of *pre* and *post* regions suggests that the following syntactic properties for barriers should hold under any reasonable semantics:

Property 1 (Eliminating Redundant Barriers) *Given a statement group S^{par}, we have*

$$\begin{pmatrix}\texttt{---}\\ S^{par}\end{pmatrix} \equiv \begin{pmatrix}S^{par}\\ \texttt{---}\end{pmatrix} \equiv S^{par}$$

Property 2 (Associativity of Barriers) *Given statement groups S_1^{par}, S_2^{par}, and S_3^{par}, we have*

$$\left(\begin{pmatrix}S_1^{par}\\ \texttt{---}\\ S_2^{par}\end{pmatrix}\\ \texttt{---}\\ S_3^{par}\right) \equiv \left(S_1^{par}\\ \texttt{---}\\ \begin{pmatrix}S_2^{par}\\ \texttt{---}\\ S_3^{par}\end{pmatrix}\right)$$

4 A Kernel Language

In this section we provide a framework for discussing the semantics of barriers. We present a small parallel language and its rewrite rule semantics.

c	$\in$	Constant
$F, t, x, y, z \ldots$	$\in$	Identifier
$SE, X, Y, Z, \ldots$	$\in$	Simple Expression
E	$\in$	Expression
S	$\in$	Statement
PF^n	$\in$	Primitive Fn. with n arguments
Constant	::=	*Integer* | *Float* | *Boolean*
SE	::=	Identifier | Constant
PF	::=	$+ \mid - \mid \cdots \mid$ `hd` | `tl` | `Cons` | $\cdots$
		| `Alloc` | `I-fetch` | `M-take` | $\cdots$
E	::=	$SE \mid PF^n(X_1, \ldots, X_n)$
		| $\mathtt{Cond}\,(X, E_1, E_2) \mid \lambda x_1 \cdots x_n.\, E$
		| $\mathtt{Ap}(F, X) \mid \mathtt{Fap}^n(F^n, X_1, \ldots, X_n)$
		| $\mathtt{Sap}^n(F^n, X_1, \ldots, X_n)$
		| $\mathtt{Pap}^i(F^n, \underline{n-i}, X_1, \ldots, X_i)$ | Block
Block	::=	$\{\ S^{par}\ \mathtt{in}\ (X_1, \ldots, X_n)\ \}$
S^{par}	::=	$S_1; \ldots; S_n \mid \epsilon$
S^{seq}	::=	$(S_1^{par}\ \text{---}\ S_2^{par})$
S	::=	Binding | Command | S^{seq}
Binding	::=	$x_1, \ldots, x_n = E$
Command	::=	$\mathtt{I\text{-}store}(X, I, Z) \mid \mathtt{M\text{-}put}(X, I, Z)$

Figure 3: The Syntax of Kernel Expression Language.

4.1 Language Syntax

Figure 3 shows the grammar of our kernel expression language. The grammar ensures that every sub-expression of a computation is named with an identifier. This is extremely desirable in order to be able to express and preserve sharing constraints among data-structures.

Aside from the usual arithmetic primitives, the strict primitives `hd` and `tl` are used to manipulate lists. `Cons` is the only non-strict primitive function: it returns the allocated `Cons`-cell before it is completely filled. The primitive function `Alloc` is used to allocate either I-Structure or M-Structure memory block of size n. The type-checker ensures that this memory is used consistently.

There are several forms of application expressions in Figure 3. `Ap` is the general apply operation, $\mathtt{Fap}^n$ is a full-arity application of a known function to n arguments, $\mathtt{Sap}^n$ denotes strict application, and $\mathtt{Pap}^i$ denotes a partial application of a function to i arguments accumulated within the closure.[6] The latter three forms are generated during compiler optimizations or as intermediate results during rewriting. The language also provides nested λ-expressions that may contain free identifiers, and a block construct that controls lexical scoping and enables precise sharing of sub-expression values.

The kernel language allows several identifiers to be bound at once in a multiple-value binding. The number of multiple-values being returned from the right-hand-side expression must match the number of identifiers being bound on the left-hand-side. Finally, a group of statements separated by semi-colons forms a parallel control region. Two parallel control regions may be sequentialized using a barrier.

4.2 Weak Head-Normal Forms

Typically, a non-strict program computes *Weak Head-Normal Forms* as its answer. An expression is said to be in normal form if it cannot be simplified any further, while an expression is said to be in weak head-normal form as soon as its top-level structure becomes "stable". Under eager evaluation, an expression has to be in weak head-normal form before it can be passed around as an argument, stored into data-structures, or returned as a result from a function. Thus weak head-normal forms provide an exact characterization of the run-time *values* under eager evaluation.

[6] We will omit the arity superscripts when they are obvious.

WHNF	$\in$	Weak Head-Normal Form	
Signal	$\in$	$\{\bot, 0\}$	
$\mathcal{W}$	::	Expression $\rightarrow$ Signal	
WHNF	::=	Constant | $\mathtt{Cons}(X, Y) \mid \mathtt{Alloc}(\underline{n})$	
		| $\lambda x_1 \cdots x_n.\, E$	
		| $\mathtt{Pap}(F^n, \underline{n-i}, X_1, \ldots, X_i)$	
$\mathcal{W}(c)$	=	0	
$\mathcal{W}(x)$	=	$\mathcal{W}(E)$	Where x is bound to E
$\mathcal{W}(\mathtt{Cons}(X, Y))$	=	0	
$\mathcal{W}(\mathtt{Alloc}(E))$	=	$\mathcal{W}(E)$	
$\mathcal{W}(\lambda x_1 \cdots x_n.\, E)$	=	0	
$\mathcal{W}(\mathtt{Pap}(F^n, \underline{n-i}, X_1, \ldots, X_i))$	=	0	
$\mathcal{W}(E)$	=	$\bot$	Otherwise

Figure 4: Definition of Weak Head-Normal Forms (Values).

The weak head-normal forms or values recognized in our language are shown in Figure 4. Constants like integers, floats, booleans and underlined expressions (*e.g.*, $\underline{n-i}$) constitute literal values. An allocated data-structure such as a `Cons`-cell or an `Alloc` memory block is also considered to be a value even if its components are not yet filled. Note that the size of the memory block must be known for the allocation to take place. λ-expressions are considered to be in weak head-normal form because we do not reduce inside the body unless it is fully applied. In addition, partial applications (`Pap`) are also considered to be in weak head-normal form because they also represent functions, albeit produced *via* currying.

It is not possible to test algorithmically within the language whether a given expression is in weak head-normal form or not without destroying the confluence property. An expression may not be in weak head-normal form at one point, but may become so after some evaluation. However, we can add to the language a monotonic primitive function $\mathcal{W}(E)$ defined over the range $\{\bot, 0\}$ without destroying confluence. This function produces the signal 0 if the given expression is in weak head-normal form and produces $\bot$ otherwise (see Figure 4). We use this primitive extensively in defining both the value semantics of the kernel language discussed below and the termination semantics of barriers described in Section 5.

4.3 Rewriting Semantics

Figure 5 shows the rewrite rules that apply to kernel language terms without barriers. We make extensive use of intermediate identifiers in order to preserve sharing of sub-expressions. This system therefore represents a Graph Rewriting System [3].

We use the $\mathcal{W}$ operator to express additional conditions under which a particular rewrite rule applies. For example, the **List** rules (`hd` and `tl`) fire only when the `Cons`-cell is available and the component being selected becomes a

δ Rules:

$$+(\underline{m},\underline{n}) \longrightarrow \underline{+(m,n)}$$
$$\cdots$$

Cond Rules:

$$\texttt{Cond True}\ (E_1,E_2) \longrightarrow E_1$$
$$\texttt{Cond False}\ (E_1,E_2) \longrightarrow E_2$$

List Rules:

$$\frac{X = \texttt{Cons}(Y,Z) \qquad \mathcal{W}(Y)=0}{\texttt{hd}(X) \longrightarrow Y}$$

$$\frac{X = \texttt{Cons}(Y,Z) \qquad \mathcal{W}(Z)=0}{\texttt{tl}(X) \longrightarrow Z}$$

Apply Rules:

$$\frac{F = \lambda X_1 \cdots X_n.\ E}{\texttt{Ap}(F,X) \longrightarrow \texttt{Pap}(F,\underline{n-1},X)}$$

$$\frac{F = \texttt{Pap}(F',\underline{n},X_1,\ldots,X_i) \qquad n>1}{\texttt{Ap}(F,X) \longrightarrow \texttt{Pap}(F',\underline{n-1},X_1,\ldots,X_i,X)}$$

$$\frac{F = \texttt{Pap}(F',1,X_1,\ldots,X_i)}{\texttt{Ap}(F,X) \longrightarrow \texttt{Fap}(F',X_1,\ldots,X_i,X)}$$

$$\frac{F = \lambda X_1 \cdots X_n.\ E}{\texttt{Fap}(F,Y_1,\ldots,Y_n) \longrightarrow (\mathcal{RB}[\![E]\!])[Y_1/X_1,\ldots,Y_n/X_n]}$$

Strict Apply Rule:

$$\frac{\mathcal{W}(X_1)+\cdots+\mathcal{W}(X_n)=0}{\texttt{Sap}(F,X_1,\ldots,X_n) \longrightarrow \texttt{Fap}(F,X_1,\ldots,X_n)}$$

I-Structure Rules:

$$\frac{\texttt{I-store}(X,\underline{i},Z) \qquad \mathcal{W}(X)=0 \quad \mathcal{W}(Z)=0}{\texttt{I-fetch}(X,\underline{i}) \longrightarrow Z}$$

$$\frac{\texttt{I-store}(X,\underline{i},Z) \qquad \mathcal{W}(X)=0 \quad \mathcal{W}(Z)=0 \quad \mathcal{W}(Z')=0}{\texttt{I-store}(X,\underline{i},Z') \longrightarrow \top}$$

M-Structure Rules:

$$\frac{\mathcal{W}(X)=0 \quad \mathcal{W}(Z)=0}{\texttt{M-put}(X,\underline{i},Z);\ Y = \texttt{M-take}(X,\underline{i}) \longrightarrow Y = Z}$$

$$\frac{\mathcal{W}(X)=0 \quad \mathcal{W}(Z)=0 \quad \mathcal{W}(Z')=0}{\texttt{M-put}(X,\underline{i},Z);\ \texttt{M-put}(X,\underline{i},Z') \longrightarrow \top}$$

Figure 5: Rewrite Rules for the Kernel Language.

value. The latter condition reflects the data-driven nature of eager evaluation. In contrast, under lazy evaluation, selecting a component from a `Cons`-cell would immediately return a *thunk* which is subsequently evaluated to a value only if necessary. From the point of view of denotational semantics, these two evaluation strategies are equivalent, but they yield very different dynamic resource utilization and termination characteristics.

Note that all the **Apply** rules except the `Sap` rule are *non-strict*: they fire even if the applied arguments have not yet become a value. In case of `Sap`, we use the $\mathcal{W}$ operator to wait explicitly for the arrival of all the arguments before proceeding with the application. In other cases, partial applications are created whenever a function is applied to a number of arguments that is less than its arity. Once all the arguments have been syntactically applied, the function body is instantiated, replacing λ-bound parameters by actual arguments. The operation $\mathcal{RB}[\![E]\!]$ simply creates a new copy of the λ-body by renaming all its bound identifiers to new identifiers.

Block Flattening Rule:

$$\begin{array}{lll}
\{\ x = \{\ y_1 = E_1'; & & \{\ x = y; \\
\quad\quad\quad \cdots & & \quad y_1 = E_1'; \\
\quad\quad\quad y_m = E_m'; & & \quad \cdots \\
\quad\quad\quad \texttt{in}\ y\ \}; & \longrightarrow & \quad y_m = E_m'; \\
\quad x_1 = E_1; & & \quad x_1 = E_1; \\
\quad \cdots & & \quad \cdots \\
\quad x_n = E_n; & & \quad x_n = E_n; \\
\quad \texttt{in}\ z\ \} & & \quad \texttt{in}\ z\ \}
\end{array}$$

Substitution Rules:

$$\{\ x_1 = E_i; \cdots ; x = c; \cdots ; x_n = E_n;\ \texttt{in}\ z\ \} \longrightarrow \qquad (c \neq \odot)$$
$$\{\ x_1 = E_i[c/x]; \cdots ; x_n = E_n[c/x];\ \texttt{in}\ z[c/x]\ \}$$

$$\{\ x_1 = E_i; \cdots ; x = y; \cdots ; x_n = E_n;\ \texttt{in}\ z\ \} \longrightarrow \qquad (x \neq y)$$
$$\{\ x_1 = E_i[y/x]; \cdots ; x_n = E_n[y/x];\ \texttt{in}\ z[y/x]\ \}$$

Degenerate Cycle Rule:

$$\{\ x_1 = E_i; \cdots ; x = x; \cdots ; x_n = E_n;\ \texttt{in}\ z\ \} \longrightarrow$$
$$\{\ x_1 = E_i[\odot/x]; \cdots ; x = \odot; \cdots ; x_n = E_n[\odot/x];\ \texttt{in}\ z[\odot/x]\ \}$$

Blowup Rule:

$$\{\ x_1 = E_i; \cdots ; x = \top; \cdots ; x_n = E_n;\ \texttt{in}\ z\ \} \longrightarrow \top$$

Figure 6: Canonicalization Rules.

The rules for I-Structures and M-Structures implement their respective protocols. The rules for I-Structures show that all `I-fetch` operations to a known location must wait until an `I-store` operation succeeds in storing a value to the same location. Two `I-store` operations to the same location take the program into an inconsistent state ($\top$). The M-Structure rules are similar, except that each `M-put` operation non-deterministically matches up with one `M-take` operation within the same control region. Two simultaneous `M-put` operations take the program into an inconsistent state.

Figure 5 shows only the reduction rules that perform useful computations. In order to apply these rules properly, program terms are brought into a canonical form after each reduction step, using the canonicalization rules shown in Figure 6. Canonicalization rules include block flattening, identifier and constant substitution, elimination of redundant bindings, and propagation of $\top$. Nested blocks are flattened by merging their sets of bindings. This cannot cause any name clashes since we always rename all bound identifiers of a λ-expression on application. Bindings of the form $x = y$ or $x = c$ cause substitution of $[y/x]$ or $[c/x]$ respectively within their entire lexical scope.[7] Only identifiers or constants are allowed to be substituted, in order to preserve precise sharing of sub-expressions among terms. After substitution, all such simple bindings are eliminated from the program term, leaving it in a unique canonical form.

It is also possible to encounter degenerate bindings of the form $x = x$ during reduction and substitution, because of

[7]Blocks (delimited by { }) determine the lexical scope of a binding. Lexical scope defines the *visibility* of a binding and is unrelated to its control region which defines the *control* of its initiation and termination.

circular dependencies present within the program. In this case, the semantic meaning of x is bottom ($\bot$), because such bindings represent deadlocked computation. Such bindings are canonicalized by substituting a special constant $\odot$ for x ($\mathcal{W}(\odot) = \bot$), while retaining the binding $x = x$ as $x = \odot$. Finally, the blowup rule shows that once an inconsistent state ($\top$) is created, the whole program becomes inconsistent.

5 Semantics of Barriers

We define the semantics of barriers by translating a kernel program with barriers into another kernel program without barriers where the termination of each control region is represented explicitly as a signal. To achieve this, we must precisely define the meaning of termination for each construct in our kernel language and provide a way of combining it in order to produce a joint termination signal.

5.1 What is Termination?

Let us consider the termination semantics of a simple parallel block. We translate it into another block as follows:

$$\begin{array}{l} \{\, x_1 = E_1; \\ \quad \cdots \\ \quad x_n = E_n; \\ \text{in } x \,\} \end{array} \Longrightarrow \begin{array}{lcl} \{\, x_1, s_1 & = & \mathbf{TE}[\![E_1]\!]; \\ \quad s_1' & = & s_1 + \mathcal{W}(x_1); \\ \quad \cdots & & \\ \quad x_n, s_n & = & \mathbf{TE}[\![E_n]\!]; \\ \quad s_n' & = & s_n + \mathcal{W}(x_n); \\ \text{in } x, s_1' + \cdots + s_n' \,\} & & \end{array}$$

An expression E is translated as $\mathbf{TE}[\![E]\!]$ which dynamically returns a value along with an explicit termination signal defined over the range $\{\bot, 0\}$. We use the weak head-normal form operator ($\mathcal{W}$) to wait for the arrival of the dynamic value and combine it with the signal using the strict arithmetic operator $+$. The resulting signal is again defined over the range $\{\bot, 0\}$. All signals in a given block are similarly combined into a single signal for the whole block.

Since every sub-expression is explicitly named in the kernel language, the signal produced in the above manner ensures the termination of the actual computation taking place within the block. But what about references to free identifiers of the block? Some of these references are used within the actual block computation and hence are automatically included in its termination signal, but others are not. For example, free identifiers present in a non-strict data constructor or the free identifiers of a conditional branch that is not taken do not directly contribute towards the result of the computation. Should we nonetheless explicitly include all such references as part of the overall termination of the block?

There are two choices before us at this point, leading to two different versions of the termination semantics. Under "data-driven" semantics, we account for all the syntactically free identifiers of a block as part of its computation, and hence always include them in the block's overall termination. Under "demand-driven" semantics, only those identifiers are included in a block's termination which are demanded by the actual computation taking place within the block. Correspondingly, we obtain two different translations for barriers that are described below and in Section 7 respectively.

5.2 Data-Driven Barrier Translation

The data-driven identifier referencing scheme uses *Dynamic Dataflow Graphs* as its compilation and execution model [4]. We may view a Kernel Id control region as a dataflow graph, where each node represents a primitive operation and arcs represent data-dependencies among them (see Figure 2 (a)). During execution, *tokens* or *messages* carry run-time data values along the arcs to their appropriate destinations. The values of all the free identifiers of a control region are explicitly sent as input tokens to its dataflow graph, and it computes the values of its bound identifiers as output tokens. Note that in this data-driven model, the values are either scalars or pointers to data objects residing in the heap — unevaluated expressions (*thunks*) are never treated as values. Similarly, selecting components from heap objects directly produces data values instead of thunks that would in lazy evaluation remain unevaluated until their value is demanded.

Figure 7 shows the rules for transforming Kernel Id language constructs such that their termination events become explicit. We provide two sets of translation rules. $\mathbf{TE}[\![]\!]$ translates a kernel language expression E into a new composite expression that computes the termination signal of the original expression E in addition to its value. $\mathbf{TS}[\![]\!]$ translates a kernel language statement S into a parallel statement group S' and a signal identifier s that is bound to a signal expression representing the termination of S. Additional syntactic properties of the given expression or the statement may also be used, such as the free identifiers (FV) or the bound identifiers (BV) of an expression or a statement.

In all the rules shown in Figure 7, we maintain the invariant that the weak head-normal forms of all the free identifiers statically present within a construct are implicitly or explicitly collected into its dynamic signal. This is reflected in the rules for identifiers, primitive function applications, conditionals, λ-expressions, and user-defined function applications. For strict primitives such as $+$ this may be redundant, but it keeps the semantics intuitive and *compositional*, *i.e.*, termination of a program term (expression or statement) depends solely on the termination of its sub-terms.

The translation rule for function application shows that we not only wait for the arrival of the operator and the operand, but also the dynamic termination signal from the function application itself. The rule for λ-expressions correspondingly translates each function body to return a signal upon its termination. This body signal waits for the arrival of all the arguments of the function, thereby ensuring that unused or non-strict arguments are also collected before signaling the termination of the function application.

It may seem redundant to collect the weak head-normal form of a function argument at the application site when we already collect it within the body of the function. However, collection at the application site is necessary because any of these sites could be a partial application. In that case, the arrival of the argument contributes towards building the partial application closure rather than a executing a function body. Building a closure is a resource-consuming computation and therefore its termination must be tracked independently.

Finally, the translation of a barrier encloses its post-region in a λ-expression which is applied to the accumulated termination signal of its pre-region using the strict application primitive `Sap`. This prevents the initiation of the com-

Expressions

$$
\begin{array}{lcl}
\mathbf{TE} & :: & \text{Expression} \rightarrow \text{Expression} \times \text{Signal Expression} \\
\\
\mathbf{TE}[\![c]\!] & = & c, \quad 0 \\
\mathbf{TE}[\![x]\!] & = & x, \quad \mathcal{W}(x) \\
\mathbf{TE}[\![PF^n(X_1, \ldots, X_n)]\!] & = & PF^n(X_1, \ldots, X_n), \quad \mathcal{W}(X_1) + \cdots + \mathcal{W}(X_n) \\
\mathbf{TE}[\![\texttt{Cond}\,(X, E_1, E_2)]\!] & = & \{\ y, s = \texttt{Cond}\,(X, \mathbf{TE}[\![E_1]\!], \mathbf{TE}[\![E_2]\!]); \\
 & & \ \texttt{in}\ y, s + \mathcal{W}(z_1) + \cdots + \mathcal{W}(z_n)\ \} \\
\text{where} \quad z_1, \ldots, z_n = FV(E_1) \cup FV(E_2) & & \\
\mathbf{TE}[\![\lambda x_1 \cdots x_n.\ E]\!] & = & \lambda x_1 \cdots x_n.\ \{\ y, s = \mathbf{TE}[\![E]\!] \\
 & & \qquad\qquad \texttt{in}\ y, s + \mathcal{W}(x_1) + \cdots + \mathcal{W}(x_n)\ \}, \quad \mathcal{W}(z_1) + \cdots + \mathcal{W}(z_n) \\
\text{where} \quad z_1, \ldots, z_n = FV(\lambda x_1 \cdots x_n.\ E) & & \\
\mathbf{TE}[\![\texttt{Ap}(F, X)]\!] & = & \{\ y, s = \texttt{Ap}(F, X)\ \texttt{in}\ y, s + \mathcal{W}(F) + \mathcal{W}(X)\ \} \\
\mathbf{TE}[\![\{\ S^{par}\ \texttt{in}\ (X_1, \ldots, X_n)\ \}]\!] & = & \{\ S'^{par}\ \texttt{in}\ (X_1, \ldots, X_n, s)\ \} \\
\text{where} \quad S'^{par}, s = \mathbf{TS}[\![S^{par}]\!] & &
\end{array}
$$

Statements

$$
\begin{array}{lcl}
\mathbf{TS} & :: & \text{Statement} \rightarrow \text{Statement} \times \text{Signal Identifier} \\
\\
\mathbf{TS}[\![x_1, \ldots, x_n = E]\!] & = & (x_1, \ldots, x_n, s_1 = \mathbf{TE}[\![E]\!]; \\
 & & s = s_1 + \mathcal{W}(x_1) + \cdots + \mathcal{W}(x_n)), \quad s \\
\mathbf{TS}[\![\epsilon]\!] & = & (s = 0), \quad s \\
\mathbf{TS}[\![S_1; \ldots; S_n]\!] & = & (S'_1; \ldots; S'_n; s = s_1 + \cdots + s_n), \quad s \\
\text{where} \quad S'_i, s_i = \mathbf{TS}[\![S_i]\!] \quad 1 \leq i \leq n & & \\
\mathbf{TS}[\![S_1^{par}\ \texttt{---}\ S_2^{par}]\!] & = & (S_1'^{par}; \\
 & & F = \lambda s.\{\ S_2'^{par}\ \texttt{in}\ y_1, \ldots, y_m, s_2\ \}; \\
 & & y_1, \ldots, y_m, s'_2 = \texttt{Sap}(F, s_1)), \quad s'_2 \\
\text{where} \quad S_i'^{par}, s_i \ = \ \mathbf{TS}[\![S^{par}]\!] \quad 1 \leq i \leq 2 & & \\
\qquad\quad\ y_1, \ldots, y_m \ = \ BV(S_2^{par}) & & \\
\mathbf{TS}[\![\texttt{I-store}(X_0, X_1, X_2)]\!] & = & (\texttt{I-store}(X_0, X_1, X_2); \\
 & & s = \mathcal{W}(X_0) + \mathcal{W}(X_1) + \mathcal{W}(X_2)), \quad s \\
\mathbf{TS}[\![\texttt{M-store}(X_0, X_1, X_2)]\!] & = & (\texttt{M-store}(X_0, X_1, X_2); \\
 & & s = \mathcal{W}(X_0) + \mathcal{W}(X_1) + \mathcal{W}(X_2)), \quad s
\end{array}
$$

Figure 7: Barrier Translation Rules based on Data-Driven Eager Evaluation.

putation below the barrier until after the computation above the barrier has terminated.

It is easy to see that the data-driven termination semantics presented above provides precise control over both resource management and scheduling of side-effects, as discussed in Section 3. The signal generation and composition rules shown in Figure 7 ensure that the production of a termination signal from a control region implies that all its input values have arrived and that the entire computation within that region including side-effects has terminated. Therefore, any resources used during this computation may be safely released.

6 Placement of Barrier Translation within a Compiler

The barrier translation presented above is a source to source transformation of a Kernel Id program. A natural question is when should this translation be performed within the compiler. Should we translate the barriers before or after other program transformations such as λ-lifting, pattern-matching etc. are performed? Moreover, many reduction rules shown in Figure 5 may be used at compile-time as compiler optimizations [2]. We may even want to postpone the barrier translation until after the optimization phase because the translation increases the complexity of the program. Does it matter (semantically) when this translation is performed?

6.1 Canonicalization *vs.* Barrier Translation

Unfortunately, the barrier translation shown in Figure 7 is very fragile even with respect to the simple canonicalization procedure described in Section 4.3. The termination semantics of a program may be seriously altered when canonicalization is performed before the barrier translation as opposed to after it. Consider the example given below.

Example 8:
```
{ ( x = a;
    ---
    a = 1; )
  in a }
```

Without canonicalization, this program deadlocks under the barrier translation because the value of `a` is required to discharge the barrier. But the binding (`x = a`) can be eliminated *via* canonicalization which would make the above program terminate normally.

If we do not perform canonicalization at all places before the barrier translation, we may introduce spurious free identifiers within a control region, as shown below.

Example 9:
```
def f x = { y = a in x };
( b = f 1;
  ---
  ... )
```

Inlining the function `f` into the pre-region of the barrier introduces the redundant binding (`y = a`) within that region. Suppose that this region is not canonicalized before barrier translation, while the function `f` is canonicalized. This would make the inlining appear incorrect, because the pre-region would collect the free identifier `a` into its termination signal while the function `f` would not.

6.2 Compiler Optimizations *vs.* Barrier Translation

The Role of Free Identifiers

A similar discrepancy may be seen in the termination characteristics of a program when traditional compiler optimizations are performed before or after the barrier translation. For instance, consider the following program fragment:

Example 10:
```
def f x = 1;
( y = f a;
  ---
  ... )
```

If the user (or the compiler) decided to inline the function `f` before performing barrier translation, it would eliminate the free identifier `a` from the pre-region of the barrier. The computation below the barrier would no longer wait for the value of the identifier `a` to arrive, as it would have if the inlining had been performed after barrier translation.

Most other reduction rules shown in Figure 5 would also yield a different termination behavior if used at compile-time before performing barrier translation. For instance, applying `Cond` rules would eliminate the free identifiers of one of the branches that would otherwise be collected into the signal of the translated conditional expression (see Figure 7). Optimizing general applications `Ap` to partial applications `Pap` would introduce new free identifiers into a control region that would not be present if the optimization had been performed after barrier translation.

Another program transformation that changes the set of free identifiers of a control region is λ-lifting [11]. λ-lifting converts free identifiers of a function into additional parameters and then lifts them to the top-level as combinators. All occurrences of the lifted function identifier are replaced by partial applications of the transformed function to its previous free identifiers. This transformation would eliminate free identifiers from the region containing the lifted λ-expression, and introduce free identifiers into regions where the lifted function identifier occurs. Again, this would alter the termination semantics of those regions.

The Role of Bound Identifiers

The translation of a λ-expression in Figure 7 shows that the signal of the λ-body collects the weak head-normal forms of all incoming bound identifiers of the function. This is essentially a resource management requirement — if an incoming argument is not used within the body of the function then it must be collected into its termination signal. This ensures that no dangling argument values are left behind to be received after the production of the termination signal has allowed the function's activation frame to be deallocated.

This data-driven compilation scheme creates some subtle problems for transformations involving functions that do not use some of their arguments. Consider the following example:

Example 11:
```
def f x y = y;                     def f x y = y;
{ f' = f b;                        { f' = f b;
 ( a = f' 1;          Inlining      ( a = 1;
   ---                  -->           ---
   b = a; );                          b = a; );
 in a }                             in a }
```

It is easy to detect that the application (`f' 1`) before the barrier is a full-arity application of the function `f`. If the compiler decided to inline this function, then the resulting code after canonicalization shown on the right would be able to discharge the barrier immediately and the program would terminate normally. If the compiler decided not to inline, then the data-driven implementation of the function `f` would wait for its first argument `b` at run-time before releasing the termination signal which in turn is supposed to discharge the barrier and allow the identifier `b` to become a value. Therefore, the program would deadlock instead.

The above discrepancy arises because the data-driven implementation deliberately waits for the unused bound identifiers of a function even though at compile-time we eliminate such bindings *via* canonicalization. On the other hand, this problem would not arise if the compiler was careful not to inline functions whose partial applications spanned multiple control regions. Whether this condition severely restricts the applicability of the inlining optimization is a topic for further investigation.

6.3 Currying Equivalence *vs.* Barrier Translation

Functional languages such as Haskell (with lazy evaluation) or functional Id without barriers (with eager evaluation) satisfy several nice denotational properties based on λ-calculus that may be used for equational reasoning among syntactic terms. For example, the following equation expresses the *currying* equivalence:

$$\lambda xy.\ e \equiv \lambda x.\ \lambda y.\ e$$

When barriers are added to the language, we would still like to preserve as many denotational equalities as possible because they provide a useful intuitive model to the user and to the compiler for reasoning about equivalence of program terms. Unfortunately, as far as the currying equivalence goes, the termination semantics shown in Figure 7 differentiates between the two cases. This is illustrated by the following example:

Example 12:
```
                              def f x = g;
def f x y = y;                def g y = y;
{ f' = f b;                   { f' = f b;
 ( a = f' 1;         vs.       ( a = f' 1;
   ---                           ---
   b = a; );                     b = a; );
 in a }                        in a }
```

The left-hand-side shows a function `f` with two arguments that is curried over two applications. `f'` gets bound to a partial application of the function `f` which behaves like the identity function in value semantics but has a slightly more complex behavior for termination semantics. The function application (`f' 1`) above the barrier does not produce a signal until it also receives the first argument of the function `f`. This is because the barrier translation rules mandate that functions should collect their unused arguments into signals.

But that can never happen, because the actual value for that argument `b` cannot be produced until the application itself terminates. So the program deadlocks.

The right-hand-side shows an uncurried version of the same function. Now `f'` gets bound to a pure identity function that does not contain any free identifiers within its closure. The application above the barrier therefore successfully terminates, producing the value for `b` which allows the binding for `f'` to terminate. Thus the program successfully produces a value and terminates.

6.4 Making Dataflow Barriers more Robust

Judging by the above discussion, it seems that the simplest way to guarantee that the source program semantics are kept intact is to perform barrier translation before any other program transformation is performed. This in itself is not problematic, and can be easily done within the compiler, although it shows that the barrier translation scheme presented in Figure 7 is not sufficiently robust. Below we outline some possible ways to improve its robustness.

One simple way to ensure that elimination or introduction of free identifiers into the pre-region of a barrier has no effect on its termination semantics is to make a permanent record of its free identifiers before any canonicalization or compiler transformations are attempted. This set would be recorded at the barrier itself by inspecting the original source program.[8] These identifiers would then be used to control the discharge of the barrier at the time of translation irrespective of the actual set of free identifiers of the pre-region obtained after canonicalization or program transformations. Note that this scheme would correctly handle Example 8 and Example 10 by permanently associating the free identifier `a` with the region above the barrier and collecting it within its termination signal during barrier translation.

The problem introduced by partial canonicalization may also be easily overcome, by ensuring that we always canonicalize the whole program before performing the barrier translation, thus ensuring that redundant bindings are consistently eliminated from everywhere within the program. This avoids the problem shown in Example 9 and makes canonicalization a robust operation.

Thus the overall barrier translation mechanism within the compiler becomes the following:

1. Record the free identifiers of every pre-region of a barrier in the source program.
2. Perform program transformations: λ-lifting, inlining, etc.
3. Canonicalize the program eliminating all redundant bindings.
4. Translate the barriers in a manner similar to Figure 7 using the weak head-normal forms of previously saved free identifiers for each pre-region.

The problem of preserving currying equivalence is a little more difficult to overcome. This show that the data-driven termination semantics are fundamentally at odds with desirable semantic properties enjoyed by the value semantics. In the next section we explore a different approach to this problem, mixing eager evaluation with a demand-driven identifier referencing mechanism.

[8] Identifier substitutions would still need to be performed on this set of free identifiers.

7 Mixed Eager and Lazy Evaluation

In this section we present an alternate translation for barriers based on a demand-driven identifier referencing mechanism within the context of non-strict, eager evaluation. This translation overcomes some of the semantic shortcomings of the data-driven barrier translation presented above and is also sufficient for resource management under an environment-based implementation. We will show the translation rules and discuss their properties, comparing them with the data-driven translation.

The concept of a control region remains unchanged under this mixed model of evaluation. We spawn all tasks within a control region eagerly, but each computation fetches the values of its free (input) identifiers as needed rather than being explicitly provided with them. Similarly, the values of the bound (output) identifiers computed within a region are not automatically sent to the computations that may need them. Instead, these values are stored in a known place from where other computations may obtain them as needed. Under this execution model, the trigger and the signal of a control region (see Figure 2 (a)) take on a real meaning since these are the only dependency arcs that actually enter and leave the control region. All data-dependency arcs become implicit in storing and fetching computed values.

7.1 Barrier Translation based on Mixed Evaluation

Figure 8 shows the new barrier translation rules under the mixed execution model. We no longer collect weak head-normal forms of the free identifiers of every sub-expression into its signal. Nevertheless, we still need to collect the weak head-normal forms of the bound identifiers of a computation into its signal, identifying the termination of that computation. Similarly, the weak head-normal forms of the identifiers participating in side-effect operations `I-store` and `M-store` signify the completion of those operations. Within expressions, identifiers are expected to be fetched on demand and therefore do not need any separate termination signal.

Another interesting difference between this translation and that of Figure 7 is the rule for λ-expressions. Note that the rule presented in Figure 7 collects all λ-bound identifiers into its termination signal in order to capture the identifiers that are left unused. On the other hand, the rule presented in Figure 8 does not collect any such identifiers. This is because unused identifiers are never fetched under mixed evaluation. This property allows the currying equivalence to be preserved as shown in Example 12. Under the new translation rule, the signal from the body of the function `f` on the left-hand-side would no longer depend on its unused bound identifier `x`; so the left-hand-side would also produce a value and terminate properly.

7.2 Properties of Mixed Barrier Semantics

The translation shown in Figure 8 is automatically robust with respect to compiler optimizations and program transformations, since termination of a computation depends only upon the values of identifiers that are actually required during that computation. No artificial data-dependencies may be introduced or eliminated by program transformations. It also satisfies the currying equivalence, and is therefore a semantically cleaner alternative to purely eager, data-driven barrier translation.

$$
\begin{array}{lcl}
 & & \textsc{Expressions} \\
\mathbf{TE} & :: & \text{Expression} \to \text{Expression} \times \text{Signal Expression} \\
 & & \\
\mathbf{TE}[\![c]\!] & = & c, \quad 0 \\
\mathbf{TE}[\![x]\!] & = & x, \quad 0 \\
\mathbf{TE}[\![PF^n(X_1,\ldots,X_n)]\!] & = & PF^n(X_1,\ldots,X_n), \quad 0 \\
\mathbf{TE}[\![\mathtt{Cond}\,(X,E_1,E_2)]\!] & = & \mathtt{Cond}\,(X, \mathbf{TE}[\![E_1]\!], \mathbf{TE}[\![E_2]\!]) \\
\mathbf{TE}[\![\lambda x_1 \cdots x_n.\ E]\!] & = & \lambda x_1 \cdots x_n.\ \mathbf{TE}[\![E]\!], \quad 0 \\
\mathbf{TE}[\![\mathtt{Ap}(F,X)]\!] & = & \mathtt{Ap}(F,X) \\
\mathbf{TE}[\![\{\ S^{par}\ \mathtt{in}\ (X_1,\ldots,X_n)\ \}]\!] & = & \{\ S'^{par}\ \mathtt{in}\ (X_1,\ldots,X_n,s)\ \} \\
\text{where}\quad S'^{par}, s = \mathbf{TS}[\![S^{par}]\!] & & \\
 & & \\
 & & \textsc{Statements} \\
\mathbf{TS}[\![]\!] & :: & \text{Statement} \to \text{Statement} \times \text{Signal Identifier} \\
 & & \\
\mathbf{TS}[\![x_1,\ldots,x_n = E]\!] & = & (x_1,\ldots,x_n,s_1 = \mathbf{TE}[\![E]\!]; \\
 & & s = s_1 + \mathcal{W}(x_1) + \cdots + \mathcal{W}(x_n)), \quad s \\
\mathbf{TS}[\![\epsilon]\!] & = & (s = 0), \quad s \\
\mathbf{TS}[\![S_1;\ldots;S_n]\!] & = & (S_1';\ldots;S_n'; s = s_1 + \cdots + s_n), \quad s \\
\text{where}\quad S_i', s_i = \mathbf{TS}[\![S_i]\!] \quad 1 \le i \le n & & \\
\mathbf{TS}[\![S_1^{par}\ \text{---}\ S_2^{par}]\!] & = & (S_1'^{par}; \\
 & & F = \lambda s.\{\ S_2'^{par}\ \mathtt{in}\ y_1,\ldots,y_m,s_2\ \}; \\
 & & y_1,\ldots,y_m,s_2' = \mathtt{Sap}(F,s_1)), \quad s_2' \\
\text{where}\quad S_i'^{par}, s_i \ = \ \mathbf{TS}[\![S^{par}]\!] \quad 1 \le i \le 2 & & \\
\qquad\quad y_1,\ldots,y_m \ = \ BV(S_2^{par}) & & \\
\mathbf{TS}[\![\mathtt{I\text{-}store}(X_0,X_1,X_2)]\!] & = & (\mathtt{I\text{-}store}(X_0,X_1,X_2); \\
 & & s = \mathcal{W}(X_0) + \mathcal{W}(X_1) + \mathcal{W}(X_2)), \quad s \\
\mathbf{TS}[\![\mathtt{M\text{-}store}(X_0,X_1,X_2)]\!] & = & (\mathtt{M\text{-}store}(X_0,X_1,X_2); \\
 & & s = \mathcal{W}(X_0) + \mathcal{W}(X_1) + \mathcal{W}(X_2)), \quad s
\end{array}
$$

Figure 8: Barrier Translation Rules based on Mixed Evaluation.

As expected, the mixed evaluation model is much more relaxed in terms of the condition governing the discharge of a barrier. The termination of the pre-region of a barrier now solely depends upon the termination of the actual computation spawned within it and not on the arrival of unused values. This may cause some programs to terminate properly under this semantics that would have deadlocked under dataflow barrier semantics as shown below:

Example 13:
```
{ ( def f x = x + a ;
    ---
    a = 1; )
  in f 3 }
```

Under dataflow barrier semantics, the value of a is still required to discharge the barrier since it is a free identifier of the pre-region, but since a is bound in the post-region, the value never arrives and the program deadlocks. Under mixed barrier semantics, function definitions always terminate immediately; so the barrier discharges, allowing the overall expression to produce a value and terminate properly.

Both dataflow and mixed termination semantics provide the power to control the termination of actual computations including those involving side-effects. The major difference between them shows up while using them for resource management. Dataflow termination semantics automatically provide a full resource management capability, by ensuring that even unused values have been received by the time a termination signal is produced from a control region. We may need to move to an environment-based implementation in order to provide a similar capability under the mixed termination semantics.

8 Conclusions and Future Work

In this paper we have presented two different termination semantics for barriers in an eagerly evaluated, implicitly parallel, imperative language. In each case, the semantics have been presented as a translation from a language with barriers into a kernel language without barriers where signals and triggers have been made explicit using a weak head-normal form operator ($\mathcal{W}$) and a strict application operator (Sap).

The purely eager termination semantics were motivated by the dataflow execution model and have been implemented in the Id/pH compiler.[9] This model lacks certain desirable semantic properties such as the currying equivalence; and this motivated us to devise an alternate, mixed termination semantics that combines eager spawning of computation along with lazy handling of identifier references.

At this point we do not have enough experience to judge whether an environment-based implementation of the mixed evaluation approach would be as efficient as the data-driven implementation of purely eager evaluation. It may also be possible to combine both approaches in a single implementation. We plan to work on these issues in the future.

9 Acknowledgments

The authors would like to thank Zena Ariola, James Hicks, Jan-Willem Maessen, and Rishiyur Nikhil for several in-

[9]The current Id compiler implementation of barriers has some bugs that were uncovered through this research.

sightful discussions that immensely helped in distilling the important issues while formalizing the semantics of barriers.

The research described in this paper was funded in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-92-J-1310.

References

[1] Zena M. Ariola and Arvind. P-TAC: A Parallel Intermediate Language. In *Proc. Functional Programming Languages and Computer Architecture*, September 1989.

[2] Zena M. Ariola and Arvind. A Syntactic Approach to Program Transformations. In *Proc. Partial Evaluation and Semantics-Based Program Manipulation*, pages 116–129. ACM Press, September 1991.

[3] Zena M. Ariola and Arvind. Properties of a First-order Functional Language with Sharing. CSG Memo 347-1, Laboratory for Computer Science, MIT, Cambridge, MA 02139, June 1994. To appear in Theoretical Computer Science, September 1995.

[4] Arvind and David E. Culler. Dataflow Architectures. In *Annual Reviews in Computer Science*, volume 1, pages 225–253. Annual Reviews Inc., Palo Alto, CA, 1986.

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

[6] Paul S. Barth. *Atomic Data Structures for Parallel Computing*. PhD thesis, Laboratory for Computer Science, MIT, Cambridge, MA 02139, March 1992. Available as Technical Report MIT/LCS/TR-532.

[7] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-Strict, Functional Language with State. In *Proc. Functional Programming Languages and Computer Architecture*, pages 538–568. Springer-Verlag, 1991. LNCS 523.

[8] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1985.

[9] P. Hudak and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR777, Department of Computer Science, Yale University, April 1990.

[10] Paul Hudak. Mutable Abstract Datatypes -or- How to Have Your State and Munge It Too. Research Report YALEU/DCS/RR-914, Department of Computer Science, Yale University, New Haven, CT 06520, December 1992. Revised May 1993.

[11] Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proc. Functional Programming Languages and Computer Architecture*, pages 190–203. Springer-Verlag, 1985. LNCS 201.

[12] John Launchbury and Simon L. Peyton Jones. Lazy Functional State Threads. In *Proc. Programming Language Design and Implementation*, pages 24–35. ACM Press, June 1994.

[13] Rishiyur S. Nikhil. Id Language Reference Manual Version 90.1. Technical Report CSG Memo 284-2, Laboratory for Computer Science, MIT, Cambridge, MA 02139, July 15 1991.

[14] Rishiyur S. Nikhil, Arvind, and James Hicks. pH Language Proposal (Preliminary). Circulated on the pH mailing list, September 1993.

[15] Simon L. Peyton Jones and Philip Wadler. Imperative Functional Programming. In *Proc. Principles of Programming Languages*, pages 71–84. ACM Press, January 1993.

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

[17] Thorsten von Eicken, David Culler, and Klaus Erik Schauser. Berkeley Id90 I/O proposal. Draft Memo, August 1991.