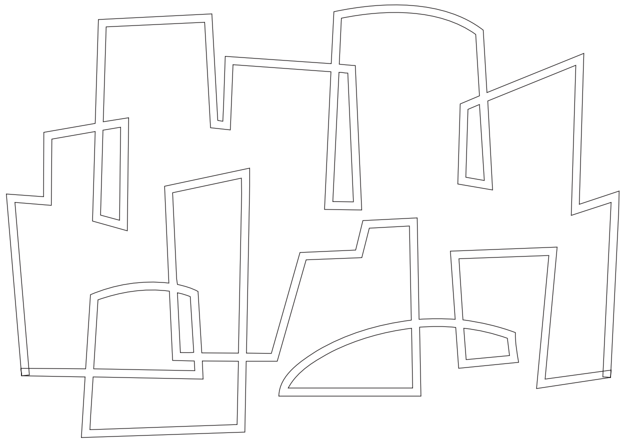
Semantics of pH: A parallel dialect of Haskell

Shail Aditya, Arvind, Jan-Willem Maeseen

1995, June

Computation Structures Group
Memo 377



Semantics of pH: A parallel dialect of Haskell

Computation Structures Group Memo 377-1
June 7, 1995

Shail Aditya Arvind Jan-Willem Maessen
MIT Laboratory for Computer Science
`{shail,arvind,earwig}@lcs.mit.edu`

Lennart Augustsson
Chalmers University
`augustss@cs.chalmers.se`

Rishiyur S. Nikhil
DEC Cambridge Research Laboratory
`nikhil@crl.dec.com`

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 pH: A parallel dialect of Haskell[*]

Shail Aditya[a] MIT   Arvind[a] MIT   Lennart Augustsson[b] Chalmers University

Jan-Willem Maessen[a] MIT   Rishiyur S. Nikhil[c] DEC, CRL

Abstract

The semantics of kernel pH are defined in the form of a parallel, normalizing interpreter. A description of I-structure and M-structure operators is also given within the same framework. Semantics of barriers in pH are presented by translation into the kernel language without barriers. The framework presented is also suitable for multithreaded compilation of pH.

1 What is pH?

pH [11] is a parallel variant of the Haskell programming language [8] with extensions for loops, synchronized side-effect operations, and explicit sequentialization. This paper discusses the operational semantics of these variations and extensions. The concrete syntax of pH is presented in the preliminary pH manual [11] and will track future Haskell versions.

There is more than one approach to parallel implementations of functional languages. For example, it is possible to exploit the implicit parallelism of a Haskell program by concurrent evaluation of the arguments of strict operators. The absence of side effects ensures that this concurrent evaluation cannot change the result. Strictness analysis techniques widen the scope of this idea, by allowing parallel evaluation of any strict function argument. Further, it is possible to provide programmer annotations to indicate sub-expressions which should be evaluated in parallel, even when the compiler cannot prove that the value of all the sub-expressions will be required. Unlike similar annotations in imperative languages, the annotations can only affect termination, and not the results (if any). All these approaches take *demand-driven* evaluation as the starting point: in the absence of annotations, an expression is evaluated only if its value is required.

An alternative approach is to use a *parallel* evaluation order which reduces all redexes in parallel, not only those whose value is required. This strategy also implements non-strict semantics, provided that no redex has its evaluation delayed indefinitely. A parallel evaluation strategy is followed by the Id programming language [10], in which any redex not in the body of a conditional or lambda abstraction is reduced. pH follows the same strategy. A brief description of pH may be given as follows.

[*]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.

[a]MIT Laboratory for Computer Science, Cambridge, MA 02139. Email: {shail,arvind,earwig}@lcs.mit.edu.

[b]Department of Computer Science, Chalmers University, Göteborg, Sweden. Email: augustss@cs.chalmers.se.

[c]Cambridge Research Laboratory, Digital Equipment Corporation, One Kendall Square, Building 700, Cambridge, MA 02139. Email: nikhil@crl.dec.com.

pH = Haskell syntax and type system + Id evaluation strategy + Id side-effect operators

In practice, implementations of pH may not guarantee that every redex is reduced eventually. This will imply that some pH programs may fail to give a result and terminate when they would do so in "traditional" Haskell. However, if pH gives a proper (non-error, non-bottom) result for a particular program then so does "traditional" Haskell, and these results are the same.

1.1 Why pH?

pH is an attempt to bring together the lazy functional community (as represented by Haskell) and the dataflow community (as represented by Id and Sisal [7]). It is hoped that by sharing a common base language, it will be easier to share ideas and implementations and freely exchange programs. By doing so, programmers need only master a single syntax and type system. There are important differences between pH and Haskell, however. By choosing Haskell as a framework, it becomes easier to isolate and therefore understand the differences between the two.

1.2 Structure of pH

pH is a layered language. pH(F) is the purely-functional subset of pH. Its main addition to Haskell is for- and while-loops. For example, one could compute the sum of the integers between 1 and n as follows.

Example 1:
```
let sum = 0
 in for i <- [1..n] do
        next sum = sum + i
    finally sum
```

Loops are purely functional, and can easily be translated into tail-recursive functions. The eager semantics of pH permits tail-recursion to be implemented more efficiently than lazy semantics.

pH(I) adds I-structures [4] to pH(F). I-structures are *single-assignment* data structures that allow fine-grain producer-consumer synchronization among parallel tasks. pH(I) preserves determinacy, *i.e.*, *confluence*, although the language is no longer referentially transparent.

pH(M) adds M-structures [6] to pH(I). M-structures are *multiple-assignment* data structures that allow mutual exclusion synchronization. M-structures introduce side-effects and non-determinacy. pH(M) also provides the ability to group statements for sequential execution using *barriers* in order to avoid unwanted race conditions among side-effect operations.

Full pH is the same as pH(M).

1.3 Outline

The rest of this paper concentrates on semantic issues in which pH differs from Haskell. Section 2 describes the parallel evaluation order of pH in the context of a kernel language and its multi-threaded interpreter. In Section 3 we extend the kernel language with I-structure and M-structure operations. Section 4 defines the semantics of barriers by translating them into the kernel language using explicit termination signals from side-effect operations. Finally, Section 5 concludes with some notes about current implementations.

c	$\in$	Constant
$f, t, x, y, z \ldots$	$\in$	Identifier
$SE, X, Y, Z, \ldots$	$\in$	Simple Expression
E	$\in$	Expression
PF^n	$\in$	Primitive Fn. with n arguments
CF^n	$\in$	Data Constructor with n arguments
S	$\in$	Statement
Constant	$::=$	*Integer* | *Float* | *Boolean* | `Nil`
SE	$::=$	Identifier | Constant
PF^1	$::=$	`hd` | `tl` | `select_`k
PF^2	$::=$	$+ \mid - \mid \cdots \mid < \mid > \mid \cdots$
CF^2	$::=$	`Cons`
CF^n	$::=$	`make_`n`_tuple`
E	$::=$	$SE \mid PF^n(x_1, \ldots, x_n) \mid CF^n(x_1, \ldots, x_n)$
		$\mid \lambda x.\ E \mid$ `case`$(x, E_1, \ldots, E_n)$
		$\mid$ `ap`$(f, z) \mid$ `sap`$(f, z) \mid$ Block
Block	$::=$	$\{\ S$ `in` $x\ \}$
S	$::=$	$\epsilon \mid x = E \mid S_1; \ldots; S_n$
Program	$::=$	Block

Figure 1: The kernel pH language.

2 Parallel Evaluation Order

pH follows an *eager* evaluation strategy: 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. This is in contrast with a *lazy* evaluation strategy followed by Haskell: only those tasks are evaluated which are required to produce the result. This strategy imposes a sequential constraint on the overall computation, although the exact ordering of tasks is decided dynamically.

In this section we describe the eager evaluation model of pH by giving a parallel interpreter for the kernel language of pH. Our interpreter is based on a parallel abstract machine, which is in the spirit of the G-machine and its later variants [9, 5, 12]. We first describe the kernel language (Section 2.1) and then our parallel abstract machine (Section 2.2) and its instruction set (Section 2.3). It is followed by a description of the interpreter (Section 2.4).

2.1 The Kernel pH Language

The abstract syntax of the kernel pH language is shown in Figure 1. The kernel language ensures that every intermediate result of a complex expression is explicitly named using an identifier. This is convenient for expressing and preserving the sharing of subexpressions within a computation [3]. In order to simplify the description of the evaluation rules in this paper, we do not allow constants to appear as arguments to primitive functions.

Aside from the usual arithmetic primitives, the kernel language provides functions for constructing and selecting elements of lists and n-ary tuples. The language also provides n-ary `case`

ι	$\in$	Instruction		
ℓ	$\in$	Location		
ν	$\in$	Value	$::=$	Constant $\mid$ Location $\mid$ $\langle \lambda x.\ E, \rho \rangle$
ρ	$\in$	Environment Frame	$=$	Identifier $\rightarrow$ Location
σ	$\in$	Store	$=$	Location $\rightarrow$ $\{\langle \textsf{full}\ \nu \rangle \mid \langle \textsf{defer}\ \delta s \rangle\}$
ω	$\in$	Thread	$=$	Code $\times$ Environments
δ	$\in$	Suspension	$=$	Thread
δs	$\in$	Suspensions	$=$	*List*(Thread)
ωs	$\in$	Work Queue	$=$	*List*(Thread)
		Accumulator	$=$	Value
ιs	$\in$	Code	$=$	*List*(Instruction)
ρs	$\in$	Environment	$=$	*List*(Environment Frame)
		Processor	$=$	Accumulator $\times$ Code $\times$ Environment
		Machine	$=$	*List*(Processor) $\times$ Work Queue $\times$ Store

	Processor			Global Memory	
State	Accumulator	Code	Environment	Work Queue	Store
Initial (starting proc.)	_	ιs_0	[ ]	[ ]	{}
Initial (other proc.)	_	[ ]	[ ]	[ ]	{}
Final (all proc.)	_	[ ]	[ ]	[ ]	σ
Error (any proc.)	`storerr`	ιs	ρs	ωs	σ

Where $\iota s_0 = [\textsf{eval}(\text{Program}), \textsf{print}, \textsf{schedule}]$

Figure 2: Dynamic entities used by the abstract machine.

expressions which select one of the branches based on the value of the dispatch identifier, nested λ-expressions which may contain free identifiers, a general function application operator `ap`, and a parallel block construct that controls lexical scoping and enables precise sharing of subexpression values. The order of bindings in a block is not significant. The identifier following the `in` keyword in a block expression denotes the result of the block.

2.2 A Parallel Abstract Machine

Our parallel abstract machine consists of a number of sequential processors connected to a global shared memory. The important features of our machine are described in Figure 2. Each memory location can hold a value or a list of suspended threads, and is tagged with its status. A full location contains a value. A newly allocated location is tagged with defer and it is initialized to be empty. A *value* is either a constant, an allocated store location, or a function closure. The global memory holds three types of entities: heap storage, activation frames and a *work queue* which is a list of *threads*, that is, (*code*, *environment*) pairs.

The processor state consists of code, environment, and an *accumulator* which can hold a value. Any idle processor can get work by dequeuing a thread from the work queue and loading its code and environment space from the thread. Initially, it is assumed that the whole machine is empty. A program E is started by executing the following code on some processor in the empty environment.

$$[\textsf{eval}(E), \textsf{print}, \textsf{schedule}]$$

The eval instruction interprets an expression by structural decomposition in a given environment. Environments map program identifiers to frame locations, and are structured in a stack like manner. eval of a block allocates a new *activation frame* in the global memory to store the values of the bound identifiers of the block. It also creates a new *environment frame* for these identifiers and pushes it on the processor environment. The bound identifiers of the block point to locations in the newly allocated activation frame. New environment frames (but not activation frames) are also created in case of λ-expressions and function applications. The sharing of computations is achieved entirely through the activation frame locations: all references to an identifier within the scope of its definition lead to the same frame location. The separation of environment frames and activation frames permits environments to be copied freely and therefore allows us to treat function closures in the same way as simple values.

At each step, the processor executes the instruction at the head of the current code sequence, modifying the processor state in the process. The instruction set of the processor has instructions to load and store the accumulator, perform arithmetic and logic operations on store locations, and suspend the current thread in case of missing values. This is in contrast with lazy functional languages where unevaluated locations contain *thunks* that compute its value on-demand. Typically an instruction is popped from the code sequence after execution, but some instructions may also add new instructions to the code sequence. This is the primary way to alter the control flow. There are also instructions to push and pop environment frames, and enqueue and dequeue threads from the work queue.

The work queue is maintained as a FIFO queue in order to guarantee fair scheduling. This queue must be manipulated atomically. It is easy to modify the machine so that each processor has its own work queue which is maintained in a FIFO manner. In case a processor goes idle, it can pick up a thread from the back of the work queue of any other processor. As long as we do not arbitrarily suspend the currently running thread on any processor, this strategy still implements a fair schedule while considerably reducing the contention due to the atomic manipulation of the work queue.

The overall state of our parallel machine is described by the state of all the processors and the global memory and the work queue. Figure 2 also shows the initial, the final and the error states of the processors of our machine. The machine starts by scheduling an initial sequence of instructions on some processor that evaluates the program, prints the result present in the accumulator[1], and then continues to schedule the remaining work present in the work queue. This emphasizes the non-strict nature of this evaluation model: the program may print a result before final termination. The machine halts when both the code sequence and the work queue are empty. The machine is said to be *deadlocked* when there are suspended computations in the store but there is no work to be done. If there are no suspended computations, then the machine is said to have *terminated normally*. An error may be generated during execution due to exceptional conditions such as a type mismatch, an arithmetic overflow, or out of bounds access to data structures. However, if an attempt is made to write into a memory location that is already full then the whole machine is said to have reached an error state as shown in Figure 2.

2.3 Instruction Set

Figure 3 shows the instructions used by our interpreter. We will describe the semantics of these instructions in terms of a state transition involving a 5-tuple $(\nu, \iota s, \rho s, \omega s, \sigma)$. The first three

[1]The main thread is the only place where a print instruction may appear.

eval(E)	::	Evaluate E in the current environment and leave the result in the accumulator
touch(ℓ)	::	Check if the location ℓ is full, if not suspend the current thread
rtouch(ℓ)	::	Check if the location ℓ is full, if not suspend the current thread including the rtouch instruction. (Thus, rtouch can be retried when the thread is activated again).
load(ℓ)	::	Load the accumulator with the value present in the location ℓ (the location must be full)
take(ℓ)	::	Load the accumulator with the value present in the location ℓ and mark the location empty (the location must be full)
loadi_$k(\ell)$	::	Load the accumulator with the value in the location at an offset k from the location pointed to by ℓ, or suspend the thread if the location is empty
store(ℓ)	::	Store the value in the accumulator in the location ℓ and reactivate any suspensions
storerr(ℓ)	::	Generate an error if the location ℓ is full
$\mathsf{PF}^n(\ell_1, \ldots, \ell_n)$	::	Apply strict primitive operator PF^n (such as $+$) to the values at locations $\ell_1, \ldots, \ell_n$ (the locations must be full)
switch$(\ell_x, \iota s_1, \ldots, \iota s_n)$	::	Switch to the branch indexed by the value at location ℓ_x
pushenv(ρ)	::	Push the environment frame ρ onto the environment stack
popenv	::	Pop the top-level environment frame from the environment stack
schedule	::	Schedule the next thread from the work queue
print	::	Print the accumulator contents

Figure 3: The instruction set of the parallel abstract machine.

elements of the tuple refer to the accumulator carrying a value ν, the code sequence ιs, and the environment ρs on the processor where the instruction is executed. The last two elements of the tuple are the work queue ωs, and the dynamic store σ which are shared amongst all processors.

All the instructions that modify the global store must be executed atomically. That is, while a processor reads and modifies a given location within the store, no other processor should be allowed to read or write that location. Transactions on different locations may proceed in parallel. In case where an instruction sequence ιs has to be executed atomically, we will indicate that by writing $atomic(\iota s)$.

Touch Instructions

The touch instruction tests the status of a location. If the status is full, it does nothing. Otherwise, (status is defer) the code sequence following the current instruction is suspended at that location along with the current environment. The rtouch instruction is similar except that the touch operation is retried.

touch:

$$\frac{\langle \_ \quad \mathsf{touch}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{full}\ \nu\rangle]\rangle}{\langle \_ \quad \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

$$\frac{\langle \_ \quad \mathsf{touch}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{defer}\ \delta s\rangle]\rangle}{\langle \_ \quad [\mathsf{schedule}] \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{defer}\ (\iota s, \rho s) : \delta s\rangle]\rangle}$$

rtouch:

$$\frac{\langle \_ \quad \mathsf{rtouch}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{full}\ \nu\rangle]\rangle}{\langle \_ \quad \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

$$\frac{\langle \_ \quad \mathsf{rtouch}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{defer}\ \delta s\rangle]\rangle}{\langle \_ \quad [\mathsf{schedule}] \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{defer}\ (\mathsf{rtouch}(\ell) : \iota s, \rho s) : \delta s\rangle]\rangle}$$

Note that these instructions do not start the evaluation of the expression that will fill the location being touched. In lazy evaluation this expression is always known and a thunk for it can be stored in this location. In that case, the above instructions can be easily generalized to enqueue the thunk into the work queue.

Load and Store Instructions

The load instruction loads the contents of a given location into the accumulator. The location must already be full. The take instruction is similar to load except that it leaves the location empty.

load:

$$\frac{\langle \_ \quad \mathsf{load}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{full}\ \nu\rangle]\rangle}{\langle \nu \quad \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

take:

$$\frac{\langle \_ \quad \mathsf{take}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{full}\ \nu\rangle]\rangle}{\langle \nu \quad \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{defer}\ [\,]\rangle]\rangle}$$

loadi_k is an indexed, indirect load instruction. It desugars into touching an indexed location and then loading its value into the accumulator.

loadi_k:

$$\frac{\langle \_ \quad \mathsf{loadi\_}k(\ell_x) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_x \mapsto \langle \mathsf{full}\ \ell\rangle]\rangle}{\langle \_ \quad \mathsf{touch}(\ell + k) : \mathsf{load}(\ell + k) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

The store instruction stores the contents of the accumulator into the given location. It also reactivates any suspensions waiting in the location by enqueuing them in the work queue.

store:

$$\frac{\langle \nu \quad \mathsf{store}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{defer}\ \delta s\rangle]\rangle}{\langle \nu \quad \iota s \quad \rho s \quad \omega s +\!\!+ \delta s \quad \sigma[\ell \mapsto \langle \mathsf{full}\ \nu\rangle]\rangle}$$

Storing in a location that is already full is regarded as an error. The storerr instruction tests the status of a location and produces an error if it is already full. This instruction can precede a store instruction to avoid multiple stores to the same location.

storerr:

$$\frac{\langle \_ \quad \mathsf{storerr}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{full}\ \nu\rangle]\rangle}{\langle \mathtt{storerr} \quad \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

$$\frac{\langle \_ \quad \mathsf{storerr}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{defer}\ \delta s\rangle]\rangle}{\langle \_ \quad \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

Arithmetic and Logic Instructions

The standard arithmetic and logic primitives are straightforward. All of them leave the result in the accumulator.

+:

$$\frac{\langle \_ \quad +(\ell_1, \ell_2) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_1 \mapsto \langle \mathsf{full}\ \underline{m}\rangle, \ell_2 \mapsto \langle \mathsf{full}\ \underline{n}\rangle]\rangle}{\langle \underline{m+n} \quad \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

Switch Instruction

The switch instruction selects one of its branches based on the value present in the location being dispatched upon.

switch $(1 \le m \le n)$:

$$\frac{\langle \_ \quad \mathsf{switch}(\ell_x, \iota s_1, \ldots, \iota s_n) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_x \mapsto \langle \mathsf{full}\ \underline{m}\rangle]\rangle}{\langle \_ \quad \iota s_m \mathbin{+\!\!+} \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

Environment Manipulation Instructions

The pushenv instructions pushes the given environment frame onto the local environment stack of a processor, while the popenv instruction pops one frame from the top. The contents of the accumulator are not disturbed while popping.

pushenv:

$$\frac{\langle \_ \quad \mathsf{pushenv}(\rho) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}{\langle \_ \quad \iota s \quad \rho : \rho s \quad \omega s \quad \sigma\rangle}$$

popenv:

$$\frac{\langle \nu \quad \mathsf{popenv} : \iota s \quad \rho : \rho s \quad \omega s \quad \sigma\rangle}{\langle \nu \quad \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

Thread Scheduling Instruction

The execution of a schedule instruction on a processor signifies either the termination or the suspension of a thread. The processor schedules a new thread from the head of the work queue or starts *idling* if the work queue is empty. The machine halts when all processors are idling.

schedule:

$$\frac{\langle \_ \quad [\mathsf{schedule}] \quad \rho s \quad (\iota s', \rho s') : \omega s \quad \sigma\rangle}{\langle \_ \quad \iota s' \quad \rho s' \quad \omega s \quad \sigma\rangle}$$

$$\frac{\langle \_ \quad [\mathsf{schedule}] \quad \rho s \quad [\,] \quad \sigma\rangle}{\langle \_ \quad [\,] \quad \rho s \quad [\,] \quad \sigma\rangle}$$

2.4 Expression Evaluation

In this section we describe the eval instruction which is used to destructure and evaluate expressions. It leaves the value of the expression in the accumulator.

Evaluation of a constant simply loads that constant into the accumulator.

eval constant:

$$\frac{\langle \_ \quad \mathsf{eval}(c) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}{\langle c \quad \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}$$

Evaluation of an identifier first touches it and then loads its value in the accumulator.

eval identifier:

$$\frac{\langle \_ \quad \mathsf{eval}(x) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}{\langle \_ \quad \mathsf{touch}(\rho s(x)) : \mathsf{load}(\rho s(x)) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}$$

In case of a strict primitive function, we touch all its arguments before executing the primitive application.

eval strict PF:

$$\frac{\langle \_ \quad \mathsf{eval}(PF^n(x_1, \ldots, x_n)) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}{\langle \_ \quad \mathsf{touch}(\rho s(x_1)) : \ldots : \mathsf{touch}(\rho s(x_n)) : \mathsf{PF}^n(\rho s(x_1), \ldots, \rho s(x_n)) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}$$

All data selector functions such as hd, tl, and $\mathsf{select\_}k$ are directly implemented using the indexed, indirect load instruction $\mathsf{loadi\_}k$.

The non-strict constructor `Cons` allocates a pair of empty locations in the store and immediately returns the pointer to the first location as its value. It also pushes work into the work queue to evaluate the arguments of `Cons` and to fill these locations eagerly. The n-ary tuple constructor `make_`n`_tuple` operates in the same fashion. Note that under lazy evaluation we would have stored thunks into these locations which would be evaluated on demand.

eval `Cons`:

$$\frac{\langle \_ \quad \mathsf{eval}(\mathtt{Cons}(x_1, x_2)) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}{\langle \ell \quad \iota s \quad \rho s \quad \omega s +\!\!+ [\omega_1, \omega_2] \quad \sigma' \rangle}$$

where $\sigma' = \sigma + \{\ \ell \mapsto \langle \mathsf{defer}\ [\,] \rangle, (\ell + 1) \mapsto \langle \mathsf{defer}\ [\,] \rangle\ \}$
$\omega_1 = ([\mathsf{eval}(x_1), \mathsf{store}(\ell), \mathsf{schedule}], \rho s)$
$\omega_2 = ([\mathsf{eval}(x_2), \mathsf{store}(\ell + 1), \mathsf{schedule}], \rho s)$

λ-expressions are evaluated to a closure value. The closure records the locations for the free identifiers of the λ-body in a new environment frame.

eval λ-expression:

$$\frac{\langle \_ \quad \mathsf{eval}(\lambda x.E) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}{\langle \langle \lambda x.\, E, \rho \rangle \quad \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}$$

where $y_1, \ldots, y_m = FV(\lambda x.E)$
$\rho = \{\ y_1 \mapsto \rho s(y_1), \ldots, y_m \mapsto \rho s(y_m)\ \}$

Evaluation of a function application proceeds by first touching the function and then applying the resulting closure to the argument. The function argument is touched prior to application only if the function application is strict (`sap`).

eval ap:

$$\frac{\langle \_ \quad \mathsf{eval}(\mathsf{ap}(f,z)) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}{\langle \_ \quad \mathsf{touch}(\rho s(f)) : \mathsf{ap}(\rho s(f), \rho s(z)) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}$$

eval sap:

$$\frac{\langle \_ \quad \mathsf{eval}(\mathsf{sap}(f,z)) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}{\langle \_ \quad \mathsf{touch}(\rho s(f)) : \mathsf{touch}(\rho s(z)) : \mathsf{ap}(\rho s(f), \rho s(z)) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}$$

Given a closure, the ap instruction extends the closure environment frame with the given argument location and starts the evaluation of the function body after pushing the new environment frame onto the current environment. The old environment is restored after the evaluation of the body produces a result in the accumulator[2].

ap:

$$\frac{\langle \_ \quad \mathsf{ap}(\ell_f, \ell_z) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_f \mapsto \langle \mathsf{full}\ \langle \lambda x.\, E, \rho \rangle \rangle] \rangle}{\langle \_ \quad \mathsf{pushenv}(\rho') : \mathsf{eval}(E) : \mathsf{popenv} : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}$$

where $\rho' = \rho + \{\ x \mapsto \ell_z\ \}$

The evaluation of the case expression proceeds by first touching the dispatch expression. This must yield an integer within the range of the dispatch which is used to select the appropriate branch.

eval case:

$$\frac{\langle \_ \quad \mathsf{eval}(\mathsf{case}(x, E_1, \ldots, E_n)) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}{\langle \_ \quad \mathsf{touch}(\rho s(x)) : \mathsf{switch}(\rho s(x), [\mathsf{eval}(E_1)], \ldots, [\mathsf{eval}(E_n)]) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}$$

The evaluation of a block expression allocates a new activation frame in the store with a fresh location for each bound identifier of the block. It also allocates new environment frame which points to the locations in the new activation frame. This environment frame is pushed onto the current environment. Following eager evaluation, auxiliary threads are added to the work queue to evaluate each of the right-hand side expressions in the extended environment. The result identifier is also evaluated under this extended environment.

eval block:

$$\frac{\langle \_ \quad \mathsf{eval}(\{\ x_1 = E_1; \ldots; x_n = E_n\ \mathtt{in}\ x\ \}) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}{\langle \_ \quad \mathsf{pushenv}(\rho) : \mathsf{eval}(x) : \mathsf{popenv} : \iota s \quad \rho s \quad \omega s +\!\!+ \omega s' \quad \sigma' \rangle}$$

$$\begin{aligned} \text{where} \quad & \rho = \{\ x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\ \} \\ & \sigma' = \sigma + \{\ \ell_1 \mapsto \langle \mathsf{defer}\ [\,] \rangle, \ldots, \ell_n \mapsto \langle \mathsf{defer}\ [\,] \rangle\ \} \\ & \omega s' = \ [([\mathsf{eval}(E_1), \mathsf{store}(\ell_1), \mathsf{schedule}], \rho : \rho s), \\ & \qquad \ldots \\ & \qquad ([\mathsf{eval}(E_n), \mathsf{store}(\ell_n), \mathsf{schedule}], \rho : \rho s)] \end{aligned}$$

It is also possible to initialize the newly allocated block locations with the thunks for their respective right-hand side expressions, and not enqueue threads into the work queue. These thunks will be enqueued when the locations are touched.

3 Side-Effect Operations

Functional programming languages purposely do not permit assignment or mutable storage. Instead, such features are often offered through various implementation tricks such as by using

[2]Note that restoring the environment does not imply that the function environment can be deallocated because some threads of the function body may not have terminated yet.

$$
\begin{array}{lcl}
\text{Constant} & ::= & \cdots \mid \bullet \\
PF^1 & ::= & \cdots \mid \texttt{iAlloc} \mid \texttt{mAlloc} \mid \texttt{iFetch} \mid \texttt{mFetch} \mid \texttt{W} \\
PF^2 & ::= & \cdots \mid \texttt{iStore} \mid \texttt{mStore} \mid \texttt{\&} \\
S & ::= & \cdots \mid (S_1 \;\texttt{---}\; S_2)
\end{array}
$$

Figure 4: Kernel language extensions for side-effect operations.

monadic programming techniques. Alternatively, side-effects may be introduced *via* the traditional assignment operator after specifying a precise sequential ordering on all operations (*e.g.*, Scheme and ML). Id, motivated by concerns for parallelism, offers yet another alternative for incorporating side-effects. In Id, it is possible to specify only a partial order on side-effect operations and still retain an overall consistent picture of the computation.

pH supports imperative operations on I-structure [4] and M-structure [6] objects. I-structures allow the creation of a data structure to be separated from the definition of its components: attempts to use the value of a component are automatically delayed until that component is defined; attempts to redefine a component lead to an error state. M-structures, on the other hand, are fully mutable data structures whose components can be redefined repeatedly: an `mFetch` operation reads and empties a full component; an `mStore` operation (re)defines it; two successive `mStore` operations on the same component lead to an error state unless separated by an `mFetch` operation.

For some programs in pH we need a way to sequentialize M-structure operations in order to avoid race conditions implied by its parallel evaluation order. This is accomplished by the use of *control regions* and *barriers* [2]. A control region is informally defined as a set of concurrent threads that are under the same control dependence and therefore always execute together. Threads within the same control region may execute in any order or in an interleaved manner as long as the data dependencies among them are respected. Barriers provide a mechanism to detect the termination of a set of parallel activities enclosed within a control region. A barrier (`---`) creates 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*. Intuitively, no computation within the post-region is allowed to proceed until all the side-effect computations within the pre-region have terminated. This semantics is different from those proposed in [2] where a barrier waits for the termination of *all* computations rather than just those which cause side-effects.

In the rest of this section we describe the pH extensions to deal with I-structure and M-structure operations. The semantics of barriers is described in Section 4.

3.1 Kernel Language Extensions

We extend the kernel language to incorporate side-effect operations and barriers as shown in Figure 4. We add primitives to allocate, read, and write I-structures and M-structures. The fetch and store operations directly address store locations: indexed addressing is handled separately in the kernel language.

We also extend the syntax for block bindings to allow barriers—two sets of parallel block bindings may now be sequentialized by using a barrier (`---`) between them. Additional constants ($\bullet$), primitive operators (`W` and `&`) and strict function application (`sap`) are used to translate barriers into an ordinary set of bindings.

3.2 Evaluation Rules for Side-Effect Operations

In this section we describe additional evaluation rules for internal primitive operators involving I-structures, M-structures, and barriers. All these operators are strict on all their inputs.

I-structure Instructions

The iAlloc instruction allocates an empty I-structure array. The iFetch and iStore instructions address an I-structure location directly; all address arithmetic must be done separately. The iFetch instruction is similar to $\mathsf{loadi\_}k$. The iStore instruction updates an empty location with a value and reactivates any suspended continuations. Storing to an already full location is considered to be an error.

iAlloc:

$$\frac{\langle \_ \quad \mathsf{iAlloc}(\ell_x) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_x \mapsto \langle \mathsf{full}\ \underline{n}\rangle]\rangle}{\langle \ell \quad \iota s \quad \rho s \quad \omega s \quad \sigma'\rangle}$$

where $\sigma' = \sigma + \{\ \ell \mapsto \langle \mathsf{defer}\ [\,]\rangle, \ldots, (\ell + n - 1) \mapsto \langle \mathsf{defer}\ [\,]\rangle\ \}$

iFetch:

$$\frac{\langle \_ \quad \mathsf{iFetch}(\ell_x) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_x \mapsto \langle \mathsf{full}\ \ell\rangle]\rangle}{\langle \_ \quad \mathsf{touch}(\ell) : \mathsf{load}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

iStore:

$$\frac{\langle \_ \quad \mathsf{iStore}(\ell_x, \ell_z) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_x \mapsto \langle \mathsf{full}\ \ell\rangle]\rangle}{\langle \_ \quad \mathit{atomic}([\mathsf{storerr}(\ell), \mathsf{load}(\ell_z), \mathsf{store}(\ell)]) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

M-structure Instructions

The mAlloc and mStore instructions are identical to the corresponding instructions for I-structures. The mFetch instruction is similar to the iFetch except that it empties the location being accessed and has to be retried if the location is already empty.

mAlloc:

$$\frac{\langle \_ \quad \mathsf{mAlloc}(\ell_x) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_x \mapsto \langle \mathsf{full}\ \underline{n}\rangle]\rangle}{\langle \ell \quad \iota s \quad \rho s \quad \omega s \quad \sigma'\rangle}$$

where $\sigma' = \sigma + \{\ \ell \mapsto \langle \mathsf{defer}\ [\,]\rangle, \ldots, (\ell + n - 1) \mapsto \langle \mathsf{defer}\ [\,]\rangle\ \}$

mFetch:

$$\frac{\langle \_ \quad \mathsf{mFetch}(\ell_x) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_x \mapsto \langle \mathsf{full}\ \ell\rangle]\rangle}{\langle \_ \quad \mathit{atomic}([\mathsf{rtouch}(\ell), \mathsf{take}(\ell)]) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

mStore:

$$\frac{\langle \_ \quad \mathsf{mStore}(\ell_x, \ell_z) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_x \mapsto \langle \mathsf{full}\ \ell\rangle]\rangle}{\langle \_ \quad \mathit{atomic}([\mathsf{storerr}(\ell), \mathsf{load}(\ell_z), \mathsf{store}(\ell)]) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

4 Semantics of M-Barriers

In this section we describe the semantics of barriers by translating them into ordinary block bindings. The idea is to systematically construct a composite termination signal from all the side-effect

Expressions

$$
\begin{array}{lcl}
\mathbf{TE} & :: & \text{Expression} \to \text{Expression} \\
\\
\mathbf{TE}[\![c]\!] & = & c, \quad \bullet \\
\mathbf{TE}[\![x]\!] & = & x, \quad \bullet \\
\mathbf{TE}[\![\texttt{mFetch}(x)]\!] & = & \{\ y = \texttt{mFetch}(x); \\
 & & \texttt{in}\ y, \quad \texttt{W}(y)\ \} \\
\mathbf{TE}[\![\texttt{mStore}(x,z)]\!] & = & \{\ y = \texttt{mStore}(x,z); \\
 & & \texttt{in}\ y, \quad \texttt{W}(y)\ \} \\
\mathbf{TE}[\![PF^n(x_1,\ldots,x_n)]\!] & = & PF^n(x_1,\ldots,x_n), \quad \bullet \\
\mathbf{TE}[\![\lambda x.\,E]\!] & = & \lambda x.\,\mathbf{TE}[\![E]\!], \quad \bullet \\
\mathbf{TE}[\![\texttt{case}\ (x,E_1,\ldots,E_n)]\!] & = & \texttt{case}\ (x,\mathbf{TE}[\![E_1]\!],\ldots,\mathbf{TE}[\![E_n]\!]) \\
\mathbf{TE}[\![\texttt{ap}(f,x)]\!] & = & \texttt{ap}(f,x) \\
\mathbf{TE}[\![\{\ S\ \texttt{in}\ x\ \}]\!] & = & \{\ S'\ \texttt{in}\ x, \quad s\ \} \\
\text{where} \quad S', s = \mathbf{TS}[\![S]\!] & &
\end{array}
$$

Statements

$$
\begin{array}{lcl}
\mathbf{TS}[\![]\!] & :: & \text{Statement} \to \mathit{List}(\text{Statement}) \times \text{Identifier} \\
\\
\mathbf{TS}[\![\epsilon]\!] & = & (s = \bullet), \quad s \\
\mathbf{TS}[\![x = E]\!] & = & (x, s = \mathbf{TE}[\![E]\!]), \quad s \\
\mathbf{TS}[\![S_1;\ldots;S_n]\!] & = & (S_1';\ldots;S_n'; s = s_1\ \texttt{\&}\ \cdots\ \texttt{\&}\ s_n), \quad s \\
\text{where} \quad S_i', s_i = \mathbf{TS}[\![S_i]\!] \quad 1 \le i \le n & & \\
\mathbf{TS}[\![S_1\ \texttt{---}\ S_2]\!] & = & (S_1'; \\
 & & f = \lambda s.\{\ S_2'\ \texttt{in}\ y_1,\ldots,y_m,s_2\ \}; \\
 & & y_1,\ldots,y_m,s_2' = \texttt{sap}(f,s_1)), \quad s_2'
\end{array}
$$

$$
\begin{array}{llcl}
\text{where} & S_i', s_i & = & \mathbf{TS}[\![S]\!] \quad 1 \le i \le 2 \\
 & y_1,\ldots,y_m & = & BV(S_2)
\end{array}
$$

Figure 5: Semantics of M-barriers.

operations taking place within the pre-region of a barrier (including its child regions) which would then be used to trigger the operations present in the post-region of the barrier. The notion of signal generation and composition may be understood by looking at the translation of a simple parallel block as shown below.

$$
\begin{array}{lcl}
\mathbf{TE}[\![\ \{\ x_1 = E_1; & = & \{\ x_1, s_1 \ = \ \mathbf{TE}[\![E_1]\!]; \\
\qquad \cdots & & \quad \cdots \\
\qquad x_n = E_n; & & \quad x_n, s_n \ = \ \mathbf{TE}[\![E_n]\!]; \\
\qquad \texttt{in}\ x\ \}\ ]\!] & & \quad \texttt{in}\ x, s_1\ \texttt{\&}\ \cdots\ \texttt{\&}\ s_n\ \}
\end{array}
$$

An expression E_i is translated as $\mathbf{TE}[\![E_i]\!]$ which dynamically returns a value (bound to x_i) along with an explicit termination signal s_i. The operator `W` is used to detect the termination of each side-effect operation (`mFetch` and `mStore`) within E_i. The value of s_i would become $\bullet$ as soon as all the side-effect operations in E_i have terminated. The operator `&` is then used to combine all such signals into a single signal for the whole block.

The primitive W operator produces a signal $\bullet$ when a given identifier becomes a value. The general evaluation rule for strict primitive functions would ensure that the identifier being tested is touched before applying the W operator. Similarly, the primitive & operator is used to combine signals from two subexpressions into one signal after they have been touched.

W:

$$\frac{\langle \_ \quad \mathsf{W}(\ell_x) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_x \mapsto \langle \mathsf{full}\ \nu \rangle] \rangle}{\langle \bullet \quad \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}$$

&:

$$\frac{\langle \_ \quad \&(\ell_1, \ell_2) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_1 \mapsto \langle \mathsf{full}\ \bullet \rangle, \ell_2 \mapsto \langle \mathsf{full}\ \bullet \rangle] \rangle}{\langle \bullet \quad \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}$$

The complete barrier translation appears in Figure 5. Note that tuple return values and non-refutable pattern matching is used only for clarity—these operations are directly desugared into primitive operators in the obvious way. The only non-trivial base cases for signal generation are those for the `mFetch` and `mStore` primitive operators. The `mFetch` operation is considered to have terminated when the value of the location being fetched is returned. Similarly, the `mStore` operation is considered to have terminated when it returns the value being stored.

In the translation of S_1 `---` S_2, note that S_2 gets protected by a λ-expression, so that it does not get evaluated until the λ-expression is applied to something. That something is the termination signal of S_1, and we use `sap` to ensure that the application does not commence until the termination signal is available. Finally, since the bound variables of S_2 have now gone into an inner scope (in the λ-expression), we return them all and rebind them again in the outer scope.

The barrier semantics shown here are different from the semantics presented in [2] in that here we are concerned only with the termination of the side-effect operations present within the pre-region of a barrier, while the semantics presented in [2] waited for the termination of the entire computation within the pre-region.

5 Conclusion

In this paper we have presented the various semantic issues in the design of pH at which it differs from or extends the Haskell programming language. The major difference is that pH uses a parallel eager evaluation strategy as opposed to the lazy evaluation strategy of Haskell. First, we described the kernel pH language and a normalizing interpreter for it that implements this parallel evaluation order. Next, we extended the language and the interpreter with synchronizing side-effect operations, I-structures and M-structures. Finally, we described a notion of sequentialization of side-effects using barriers. We showed a systematic translation of a kernel pH program with barriers into one without barriers using primitive termination detection operators.

The readers may note that our interpreter used an explicit instruction stream rather than directly evaluating kernel language expressions. This organization allows us to define a multithreaded compilation scheme for the kernel language within the same framework. The process of compilation can be defined as simply generating all the threads statically instead of manipulating the instruction stream dynamically. It essentially gets rid of the eval instruction. Such a compilation scheme for the kernel language is described in [1].

Currently, there is a working implementation of the pH language using the Haskell/pH HBCC front-end (written in Haskell) from Chalmers University and the Id compiler Monsoon back-end (written in Lisp) from MIT. The HBCC front-end produces a kernel pH intermediate format that is converted into dataflow graphs and fed into the Monsoon back-end. Currently, we have exercised this compiler with purely functional Haskell programs or pH programs derived by automatically transliterating Id programs. At MIT, we are currently working on a new pH back-end for commercial uniprocessor and multiprocessor architectures that is closer in spirit to the interpretation scheme described in this paper.

References

[1] Shail Aditya. Normalizing Strategies for Multithreaded Interpretation and Compilation of Non-Strict Languages. CSG Memo 374, MIT Laboratory for Computer Science, Cambridge, MA 02139, May 1995.

[2] Shail Aditya, Arvind, and Joseph E. Stoy. Semantics of Barriers in a Non-Strict, Implicitly-Parallel Language. In *Proc. Functional Programming Languages and Computer Architecture, La Jolla, CA*, Cambridge, MA 02139, June 1995. Also available as CSG Memo 367-1, MIT Lab. for Computer Sc., Cambridge, MA 02139.

[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, 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.

[5] Lennart Augustsson and Thomas Johnsson. Parallel Graph Reduction with the $< \nu, G >$-machine. In *Proc. Fourth Intl. Conf. on Functional Programming Languages and Computer Architecture, London*, pages 202–213. ACM Press, September 1989.

[6] 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.

[7] A. P. W. Böhm, D. C. Cann, J. T. Feo, and R. R. Oldehoeft. SISAL 2.0 Reference Manual. Technical Report UCRL-MA-109098, Lawrence Livermore National Laboratory, December 1991.

[8] Paul Hudak, Simon Peyton Jones, and Philip Wadler (editors). Report on the Programming Language Haskell: A Non-strict Purely Functional Language, Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[9] Thomas Johnsson. Efficient Compilation of Lazy Evaluation. *Proc. ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices*, 19(6):58–69, June 1984.

[10] 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.

[11] Rishiyur S. Nikhil, Arvind, James Hicks, Shail Aditya, Lennart Augustsson, Jan-Willem Maessen, and Yuli Zhou. pH Language Reference Manual, Version 1.0—preliminary. CSG Memo 369, Laboratory for Computer Science, MIT, Cambridge, MA 02139, January 1995.

[12] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

τ	$\in$	Threadmap	$=$	Identifier $\rightarrow$ Code
α	$\in$	Identmap	$=$	Identifier $\rightarrow$ (Integer $\times$ Integer)
ρ	$\in$	Environment	$=$	(Integer $\times$ Integer) $\rightarrow$ Location

$$\mathbf{TExp}[\![\ ]\!] \;::\; \text{Expression} \rightarrow \text{Identmap} \rightarrow (\text{Code} \times \text{Threadmap})$$

$$\text{Compilation:} \quad (\iota s_0, \tau) \;=\; \mathbf{TExp}[\![\text{Program}]\!]\ \{\ \}$$

Figure 6: Compile-time Objects.

A Compilation of Kernel pH

The compilation process involves separating program control from the data it manipulates at run-time. Essentially, this involves a systematic elimination of all eval instructions from the program, replacing them with compiled sequences of low-level instructions that compute the values of the original expressions. We also need to define the layout of environments and the mapping from identifier names to environment slots at compile-time.

In this section we will define a compiler for kernel pH programs based on the same machinery as described in Section 2. Instead of dynamically generating and interpreting code sequences as in Section 2, we will now generate a fixed set of code sequences called *threads* at compile-time. The compiler is defined in three phases: first, we describe the extensions to the abstract machine and its instruction set; next, we describe the compilation scheme for the source program transforming them into a collection of threads; and finally, we describe how these code sequences are executed on the abstract machine by giving additional execution rules for the new instructions.

A.1 Compilation Machinery

The semantic categories and translation functions used during compilation are shown in Figure 6 and are discussed below.

Every program expression bound to an identifier is compiled into a single primary thread that computes the value of that identifier. Other identifiers bound within that computation give rise to their own threads that are collected into a compile-time mapping from identifiers to threads (τ). The precise compilation rule for each source language construct appear in Section A.2.

The run-time environment mapping from identifier names to locations is split into two mappings: one compile-time mapping (α) from identifier names to a pair of index offsets $\langle i, j \rangle$ into the current environment stack, and another run-time mapping (ρ) from index-offset pairs to locations. The first index i in a pair $\langle i, j \rangle$ refers to the position of the environment frame from the top of the current environment stack (zero based). The second index j gives the offset within that frame (zero based) that carries the store location for that identifier.

During compilation, we may sometimes need to refer to some of the components of the current machine state. We use $\diamond$ for the accumulator and stack for the environment stack.

Finally, the additional machine instructions are shown in Figure 7. The detailed explanations appear along with the execution rules given in Section A.3.

$\mathsf{loadc}(c)$	::	Load a constant into the accumulator
$\mathsf{makeclsr}(\iota s, \rho)$	::	Make a closure for the thread ιs with the environment ρ
$\mathsf{apply}(\langle i_f, j_f\rangle, \langle i_z, j_z\rangle)$	::	Apply the closure f to the argument z
$\mathsf{pushwork}(\iota s, \rho s)$	::	Make and push work with the thread ιs, and the environment stack ρs
$\mathsf{copyenv}(\langle i_0, j_0\rangle \ldots \langle i_{n-1}, j_{n-1}\rangle)$	::	Allocate an environment using existing identifier locations
$\mathsf{allocenv}(\underline{n})$	::	Allocate an environment with n new identifier locations

Figure 7: Additional Compiler Instructions.

A.2 Compilation Rules

In this section we describe the compilation process of kernel language expressions into a collection of threads.

Constants

An immediate constant is simply loaded into the accumulator.

$$\mathbf{TExp}[\![c]\!]\ \alpha = [\mathsf{loadc}(c)], \quad \phi$$

Identifiers

An identifier reference compiles into a sequence of **touch** and **load** instructions that touch the preassigned environment slot and load its value into the accumulator.

$$\mathbf{TExp}[\![x]\!]\ \alpha = [\mathsf{touch}(\alpha(x)), \mathsf{load}(\alpha(x))], \quad \phi$$

Primitive Functions

For strict primitives, we first touch their arguments and then apply the primitive function. The identifier translation gives the precise coordinates of each argument being applied to the primitive function.

$$\mathbf{TExp}[\![PF^n(x_1 \ldots x_n)]\!]\ \alpha = [\mathsf{touch}(\alpha(x_1)), \ldots, \mathsf{touch}(\alpha(x_n)), \mathsf{PF}^n(\alpha(x_1), \ldots, \alpha(x_n))], \quad \phi$$

Data selector functions such as `hd`,`tl` and `select_`k are implemented using the indirect load instruction loadi_k.

$$\mathbf{TExp}[\![\texttt{select\_}k(x)]\!]\ \alpha = [\mathsf{touch}(\alpha(x)), \mathsf{loadi\_}k(\alpha(x))], \quad \phi$$

The `Cons` operator is no longer considered as a primitive for compilation purposes. It is desugared into a block of statements that allocate an I-structure and fill it as shown below.

$$\begin{array}{ll} \texttt{Cons}(x_1, x_2) \Longrightarrow \{ & t_1 = \texttt{iAlloc}(2); \\ & t_2 = \texttt{iStore}(t_1, x_1); \\ & t_3 = t_1 + 1; \\ & t_4 = \texttt{iStore}(t_3, x_2); \\ & \texttt{in}\ t_1\ \} \end{array}$$

`make_`n`_tuple` is desugared similarly.

λ-abstraction

The compilation of a function must ensure that both the free identifiers of the function and the dynamically bound identifiers at an application can be accessed properly from within the function body. Since there is no return stack in our machine, we must also explicitly provide a place to save the caller continuation at the time of function application which is reactivated by storing the result of the callee.

$$\begin{aligned}\mathbf{TExp}[\![\lambda x.\ E]\!]\ \alpha = \{\ & \iota s_b, \tau_b = \mathbf{TExp}[\![E]\!]\ \alpha_b;\\ & \iota s_d = \iota s_b +\!\!+ [\mathsf{store}(\langle 0,0\rangle) : \mathsf{popenv} : \mathsf{schedule}];\\ & \mathtt{in}\ [\mathsf{copyenv}(\alpha(y_1),\ldots,\alpha(y_m)) : \mathsf{makeclsr}(\iota s_d, \Diamond)],\quad \tau_b\ \}\end{aligned}$$

where

$$\begin{aligned} y_1 \ldots y_m &= FV(\lambda x.E)\\ \alpha_b &= \{\ x \mapsto \langle 0,1\rangle, y_1 \mapsto \langle 0,2\rangle, \ldots, y_m \mapsto \langle 0, \underline{m+1}\rangle\ \}\end{aligned}$$

We compile the function body under the assumption that its argument and all its free identifiers are accessible through the top-level environment frame as reflected in the identifier map α_b. The first slot in the top-level frame points to the location where the return continuation of the caller function is stored. This environment structure and the location to store the return continuation will be setup during function application. Note that the return continuation is reactivated at the end of the function body by updating this slot with the final result.

The λ-abstraction itself compiles to produce a closure which carries an environment frame pointing to its free identifiers.

Function Application

A function application first evaluates the function and then applies it to the given argument. The application rule is described later in Section A.3.

$$\mathbf{TExp}[\![\mathsf{ap}(f,z)]\!]\ \alpha = [\mathsf{touch}(\alpha(f)) : \mathsf{apply}(\alpha(f), \alpha(z),]),\quad \phi$$

Case Expression

For `case` expressions, we simply compile threads for each branch and generate a switch instruction to select the appropriate one.

$$\begin{aligned}&\mathbf{TExp}[\![\mathtt{case}(x, E_1,\ldots,E_n)]\!]\ \alpha =\\ &\quad \{\ \ \iota s_1, \tau_1 = \mathbf{TExp}[\![E_1]\!]\ \alpha;\\ &\qquad \ldots\\ &\qquad \iota s_n, \tau_n = \mathbf{TExp}[\![E_n]\!]\ \alpha;\\ &\qquad \mathtt{in}\ [\mathsf{touch}(\alpha(x)) : \mathsf{switch}(\alpha(x), \iota s_1, \ldots, \iota s_n)],\quad \tau_1 + \cdots + \tau_n\ \}\end{aligned}$$

Block Expression

For a block, first we need to allocate an environment to hold all the bound identifiers. Each of the right-hand sides are compiled into threads that update the appropriate environment slots with their respective values. These threads are pushed into the work queue for evaluation. The "letrec" semantics of the block are maintained by allowing the new threads being compiled to refer to themselves *via* an extended thread map. The identifier map is also adjusted to reflect the new environment frame being pushed onto the environment stack.

$$
\begin{array}{l}
\mathbf{TExp}[\![\{\ x_1 = E_1; \cdots; x_n = E_n \ \mathtt{in}\ x\ \}]\!]\ \alpha = \\
\quad \{\ \ \iota s_1, \tau_1 = \mathbf{TExp}[\![E_1]\!]\ \alpha'; \\
\qquad \iota s_1' = \iota s_1 +\!\!+ [\mathsf{store}(\langle 0, 0\rangle) : \mathsf{schedule}]; \\
\qquad \ldots \\
\qquad \iota s_n, \tau_n = \mathbf{TExp}[\![E_n]\!]\ \alpha'; \\
\qquad \iota s_n' = \iota s_n +\!\!+ [\mathsf{store}(\langle 0, \underline{n-1}\rangle]) : \mathsf{schedule}]; \\
\qquad \mathtt{in}\ [\mathsf{allocenv}(\underline{n}) : \mathsf{pushenv}(\Diamond) : \mathsf{pushwork}(\iota s_1', \mathsf{stack}) : \ldots : \\
\qquad\quad \mathsf{pushwork}(\iota s_n', \mathsf{stack}) : \mathsf{enter}(\alpha'(x)) : \mathsf{popenv}], \quad \tau_b + \tau_1 + \cdots + \tau_n\ \}
\end{array}
$$

where

$$
\begin{array}{l}
\tau_b = \{x_1 \mapsto \iota s_1', \ldots, x_n \mapsto \iota s_n'\ \} \\
\alpha'(x) = \mathbf{if}\ x = x_k\ (1 \le k \le n)\ \mathbf{then}\ \langle 0, k\rangle\ \mathbf{else}\ \langle \mathit{fst}(\alpha(x)+1), \mathit{snd}(\alpha(x))\rangle
\end{array}
$$

A.3 Execution Rules for Additional Instructions

In this section we provide the execution rules for the additional instructions given in Figure 7. Note that identifier names have been translated into index-offset pairs $\langle i, j\rangle$ pointing into the environment stack that must be dereferenced at run-time to access the store location.

Loading Accumulator

The loadc instruction is used to load an immediate constant into the accumulator.

loadc:

$$
\frac{\langle \_ \quad \mathsf{loadc}(c) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}{\langle c \quad \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}
$$

Closure Creation

The makeclsr instruction simply creates a new closure from the given thread and the environment stack.

makeclsr:

$$
\frac{\langle \_ \quad \mathsf{makeclsr}(\iota s, \rho) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}{\langle\langle \mathsf{clsr}\ (1).\ \iota s, \rho\rangle \quad \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}
$$

Function Application

Function application first allocates a flat argument environment for the function call, copying the closure environment into it. It also copies the current argument mapping and allocates a new location to save the return continuation of the caller. The return continuation would enable the callee to resume the caller after the function call has completed. Finally, the entry point present within the closure is scheduled.

apply:

$$\frac{\langle \_ \quad \mathsf{apply}(\langle i_f, j_f\rangle, \langle i_z, j_z\rangle) : \iota s \quad \rho s \quad \omega s \quad \sigma[\rho s[i_f, j_f] \mapsto \langle \mathsf{full}\ \langle \mathsf{clsr}\ (1).\ \iota s', \rho'\rangle\rangle]\rangle}{\langle \_ \quad \iota s' \quad [\rho_b] \quad \omega s \quad \sigma'\rangle}$$

where $\rho_b = \ell : \rho s[i_z, j_z] : \rho'$
$\sigma' = \sigma + \{\ \ell \mapsto \langle \mathsf{defer}\ [(\iota s, \rho s)]\rangle\ \}$

Resource Management

The pushwork instruction creates a thread with the given instruction thread and the environment chain.

pushwork:

$$\frac{\langle \_ \quad \mathsf{pushwork}(\iota s', \rho s') : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}{\langle \_ \quad \iota s \quad \rho s \quad \omega s \mathbin{+\!\!+} [(\iota s', \rho s')] \quad \sigma\rangle}$$

The copyenv and allocenv instructions construct environments. copyenv is used to create a new environment mapping from existing identifier locations such as those for the free identifiers of a function. allocenv is used for new identifiers bindings in a block. It allocates new locations in the store and initializes them to be empty.

copyenv:

$$\frac{\langle \_ \quad \mathsf{copyenv}(\langle i_0, j_0\rangle \ldots \langle i_{n-1}, j_{n-1}\rangle) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}{\langle \rho \quad \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

where $\rho = [\rho s[i_0, j_0], \ldots, \rho s[i_{n-1}, j_{n-1}]]$

allocenv:

$$\frac{\langle \_ \quad \mathsf{allocenv}(\underline{n}) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}{\langle \rho \quad \iota s \quad \rho s \quad \omega s \quad \sigma'\rangle}$$

where $\rho = [\ell_0, \ldots, \ell_{n-1}]$
$\sigma' = \sigma + \{\ \ell_0 \mapsto \langle \mathsf{defer}\ [\,]\rangle, \ldots, \ell_{n-1} \mapsto \langle \mathsf{defer}\ [\,]\rangle\ \}$