CSAIL

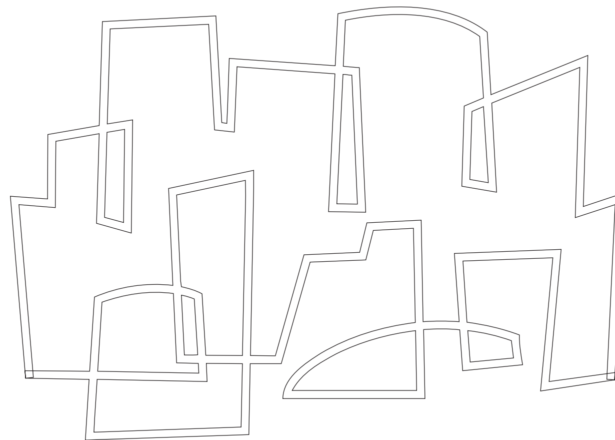
Computer Science and Artificial Intelligence Laboratory

Normalizing Strategies for Multithreaded Interpretation and Compilation of Non-Strict Languages

Shail Aditya

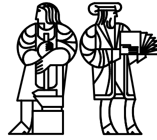
1995, May

Computation Structures Group
Memo 374



The Stata Center, 32 Vassar Street, Cambridge, Massachusetts 02139

LABORATORY FOR COMPUTER SCIENCE



MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Normalizing Strategies for Multithreaded Interpretation and Compilation of Non-Strict Languages

Computation Structures Group Memo 374
May 4, 1995

Shail Aditya
MIT Laboratory for Computer Science
shail@lcs.mit.edu

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.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Normalizing Strategies for Multithreaded Interpretation and Compilation of Non-Strict Languages

Shail Aditya
MIT Laboratory for Computer Science
shail@lcs.mit.edu

May 4, 1995

Abstract

Execution of programs written in non-strict languages such as Haskell or Id/pH require the ability to dynamically schedule multiple threads of computation. This is because the exact data dependencies among the sub-expressions (and hence their ordering) cannot be completely determined at compile-time. Synchronizing data structures such as I-structures and M-structures also need a multithreaded execution model because a computation that gets blocked on a synchronizing data structure can only be resumed if another computation produces that value. However, there is considerable flexibility in choosing the granularity of threads and their dynamic scheduling strategy when implementing such a multithreaded model on existing sequential and parallel platforms, leading to different trade-offs between the exposed parallelism and the resource requirements.

This paper presents a formal framework for characterizing and exploring the spectrum of such trade-off points – from purely eager, fine-grain dataflow execution to purely lazy, demand-driven evaluation. We present a non-strict, kernel language and its multithreaded execution model in the form of an abstract machine specification. The machine is capable of interpreting non-strict programs using different normalizing strategies that expose different amounts of parallelism and resource requirements. In particular, we present a *mixed* normalizing strategy for interpreting kernel language programs which falls somewhere between purely eager and purely lazy evaluation. We also present compilation rules for compiling kernel language programs into a set of threads that may be executed on the abstract machine following the mixed evaluation strategy.

1 Introduction

Non-strict functional languages such as Haskell [10] or Id/pH [12, 13] have the advantage of being non-procedural yet highly expressive: the user only specifies the relationships among sub-computations in the form of data dependencies rather than the exact sequence of operations that a machine must perform in order to obtain the desired answer. This leaves a lot of flexibility to the compiler and the run-time system of a language in partitioning and scheduling the sub-computations appropriately in order to expose the parallelism present within the computation and to map it efficiently onto a target architecture.

Traditionally, non-strict language implementations have used one of the following two contrasting execution strategies: a purely lazy, *demand-driven* strategy as used in lazy languages such as Haskell [14], and a purely eager, *data-driven* strategy as used in dataflow languages such as Id/pH

[4, 11]. The lazy strategy executes only those sub-computations that are absolutely necessary to produce the final answer, while the eager strategy executes all spawned tasks in parallel, restricted only by the data dependencies among them. The lazy strategy is geared towards reducing the total number of steps required to compute the answer while the eager strategy is geared towards exposing the maximum amount of parallelism within the computation. It is obvious that both these objectives are desirable, although strictly adhering to either one may turn out to be quite expensive in terms of the overall resource (memory and processor) requirements of the computation: a purely lazy strategy delays every sub-computation in memory until its value is needed by the final answer while a purely eager strategy may spawn many more tasks in parallel than are exploitable by the available number of processors.

The eager evaluation strategy of Id/pH also successfully handles computations involving I-structures [5] and M-structures [6]. I-structures are single-assignment data structures that allow fine-grain, producer-consumer synchronization among various parallel activities while M-structures are multiple-assignment data structures supporting fine-grain, mutual-exclusion synchronization. I-structures and M-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. The computation producing the value of a component therefore must be spawned *independently* in order to satisfy other computations that are waiting for that value. Purely lazy evaluation would lead to deadlock in the presence of such constructs because the waiting computations do not, in general, have a handle on the producing computations and therefore cannot make progress unless those computations are spawned independently. However, it is possible to design mixed eager and lazy evaluation strategies that can handle I-structures and M-structures while keeping the resource requirements of the program under control [9, 8].

In this paper, we define a formal framework for exploring and evaluating such mixed strategies for interpretation and compilation of non-strict programs from the standpoint of their efficient implementation on standard sequential and parallel machines. Our starting point is a non-strict, kernel language based on Id/pH with a graph rewriting semantics and a well-defined notion of normal forms (answers) [3, 2]. Subsequently, we formalize our multithreaded execution model in the form of an abstract machine which directly interprets kernel language programs. Different thread spawning and scheduling strategies on this machine allow us to explore different normalizing strategies for evaluating non-strict programs. In particular, we describe a *mixed* evaluation strategy for interpreting kernel language programs on our machine that lies between purely lazy and purely eager evaluation. We also define a compilation scheme for non-strict programs that execute on our machine using the mixed evaluation strategy.

The rest of the paper is organized as follows. In Section 2 we summarize the overall parallel execution model used throughout in this paper. In Section 3 we describe the kernel language and the general framework for our multithreaded abstract machine. In Section 4 we describe the mixed interpreter for the kernel language. Section 5 defines the translation required to convert that interpreter into a compiler. Finally, in Section 6 we present the conclusions.

2 Non-Strict, Fully Parallel Evaluation

Operationally, a strict computation of a function cannot return a result until the values of all its arguments are available; an expression needs the results of all its sub-expressions; and a data structure is not available until all its components are initialized. Non-strict computations, on the

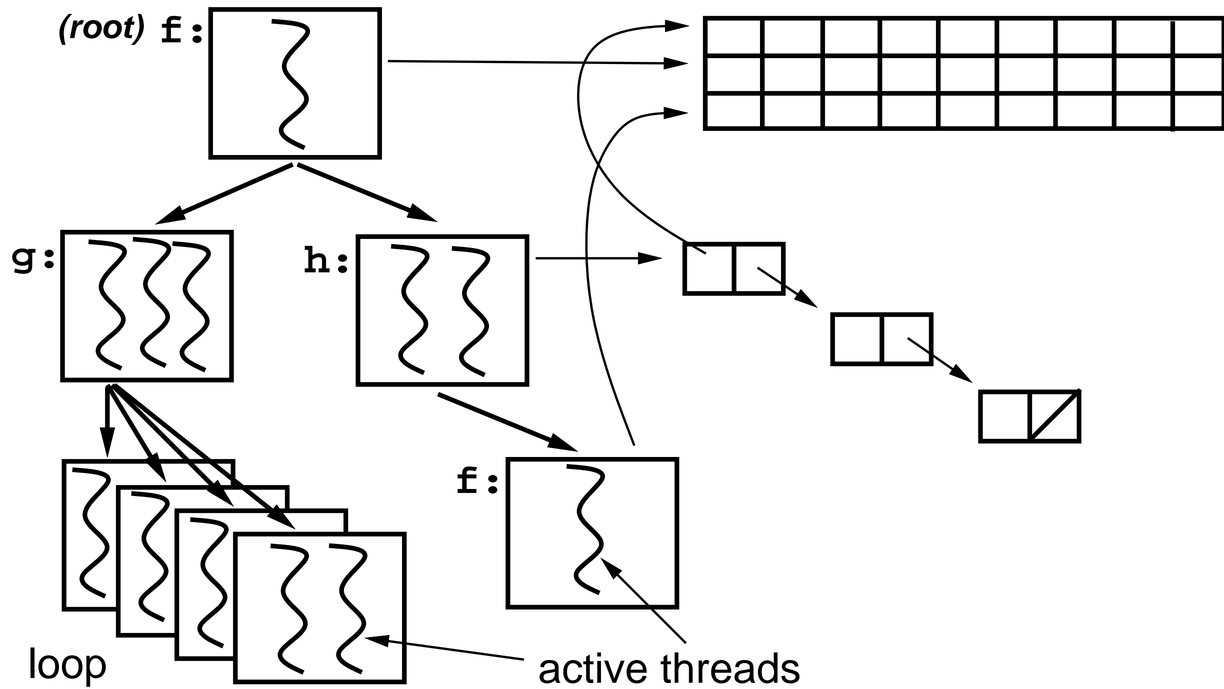


Figure 1: The Fully Parallel Execution Model of Id and pH.

other hand, relax these constraints (except when implied by data-dependencies): a function may return a partial result before all its arguments are available, or parts of a data structure may be read before it is fully computed. Id and pH languages follow an *eager* evaluation strategy for non-strict computations: 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 the Haskell language: 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.

We discuss the eager evaluation model of Id and pH in more detail 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 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 executes within the context of an *activation frame* which records function arguments and keeps temporary, local values. The program starts by allocating a root activation frame and initiating the main thread in order to evaluate the top-level expression. Function applications within a thread give rise to parallel child activations, while loop invocations give rise to multiple parallel iterations. Threads belonging to a function share its activation frame and may be active concurrently. A thread 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. Thus at any time the overall computation is represented by a tree of activation frames, exploiting both intra-procedural and inter-procedural parallelism.

In contrast, under lazy evaluation parallel computation is spawned 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.

As shown in Figure 1, all threads participating in the parallel computation share a globally addressable heap. An activation frame is deallocated when its associated function or the loop terminates, but the 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.

3 A Kernel Language

In this section we present a small parallel language and a general framework for its interpretation.

3.1 Language Syntax

Figure 2 shows the grammar of our kernel expression language. The grammar ensures that every intermediate result of a complex computation is explicitly named using an identifier. This is extremely desirable in order to be able to express and preserve sharing constraints among computations.

Aside from the usual arithmetic primitives, the strict primitives `hd` and `tl` are used to manipulate lists. `Cons` is a non-strict constructor: it returns the allocated `Cons`-cell before it is completely filled. The unary primitive function `Alloc` is used to allocate either an I-structure or an M-structure memory block of specified size. The type-checker ensures that this memory is used consistently. The memory access primitives operate directly on heap addresses: the address arithmetic is carried out separately.

There are several forms of application expressions in Figure 2. `Ap` is the general apply operation, $\texttt{Fap}^n$ is a full application of a known function to all its arguments, and $\texttt{Pap}^i$ denotes a partial application of a function to i arguments accumulated within the closure[1]. The latter two forms are generated during compiler optimizations.

The language also provides n-ary `Case` expressions that select one of the branches based on the value of the dispatch identifier, nested λ-expressions that may contain free identifiers, and a block construct that controls lexical scoping and enables precise sharing of sub-expression values.

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

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	$::=$	$\mathit{Integer} \mid \mathit{Float} \mid \mathit{Boolean} \mid \texttt{()} \mid \texttt{Nil}$
SE	$::=$	Identifier $\mid$ Constant
PF^1	$::=$	$\texttt{hd} \mid \texttt{tl} \mid \texttt{Alloc} \mid \texttt{I-fetch} \mid \texttt{M-take}$
PF^2	$::=$	$+ \mid - \mid \cdots \mid < \mid > \mid \cdots \mid \texttt{Cons} \mid \texttt{I-store} \mid \texttt{M-put}$
E	$::=$	$SE \mid PF^n(X_1, \ldots, X_n)$
		$\mid \texttt{Case}(X, E_1, \ldots, E_n) \mid \lambda x_1 \cdots x_n.\, E$
		$\mid \texttt{Ap}(F, X) \mid \texttt{Fap}^n(F^n, X_1, \ldots, X_n)$
		$\mid \texttt{Pap}^i(F^n, \underline{n-i}, X_1, \ldots, X_i) \mid$ Block
Block	$::=$	$\mid \{\, x_1 = E_1; \ldots; x_n = E_n \;\texttt{in}\; (X \mid y_1, \ldots, y_m) \,\}$
Program	$::=$	Block

Figure 2: The Syntax of Kernel Expression Language.

The order of bindings in a block is not significant. The first object in the list following the `in` keyword in a block expression denotes the result of the block. Subsequent objects in this list are the identifiers bound to expressions with side-effects. These denote auxiliary roots of the block that must be spawned eagerly in order to make progress.

3.2 Abstract Machine Framework

We would like model both direct interpretation and compiled execution of the above language within the same computational framework so that the interpretation of a program may be identified with compilation followed by execution on the same target. In this section we present this framework as an abstract machine that characterizes the book-keeping required for the evaluation of a program. This framework defines the design space for the various scheduling and resource management strategies to be explored later that would lead to various parallelism/resource requirement trade-offs.

Run-time Objects

First, we define what constitutes as an *answer*, or *value* of a computation (see Figure 3). Constants like integers, floats, booleans and underlined expressions (*e.g.*, $\underline{n}$) constitute literal values. Aggregate objects such as a `Cons`-cell or an `Alloc` memory block are allocated as a contiguous set of locations within a shared store. Each such location is also considered as a value (even if its contents are empty). λ-expressions and partial applications give rise to closure values that point to the function body and its lexical environment that keeps the values of the free identifiers of the function body.

Figure 3 also shows other run-time objects used by our abstract machine. Environments map program identifiers to values and stores map locations to values. However, in order to model various

ι	$\in$	Instruction		
ℓ	$\in$	Location		
ν	$\in$	Value	::=	Constant | Location | $\langle$clsr $(x_1 \ldots x_n).\ E, \rho s\rangle$
ρ	$\in$	Environment	=	Identifier $\rightarrow$ {$\langle$full $\nu\rangle$ | $\langle$empty $E\rangle$ | $\langle$thunk $\omega\rangle$ | $\langle$defer $\delta s\rangle$}
σ	$\in$	Store	=	Location $\rightarrow$ {$\langle$full $\nu\rangle$ | $\langle$empty$\rangle$ | $\langle$thunk $\omega\rangle$ | $\langle$defer $\delta s\rangle$}
δ	$\in$	Suspension	=	Code $\times$ Environments $\times$ Status
ω	$\in$	Work	=	Value $\times$ Code $\times$ Environments $\times$ Status
θ	$\in$	Status	::=	main | apply-1 | apply-0
ιs	$\in$	Code	=	$List$(Instruction)
ρs	$\in$	Environments	=	$List$(Environment)
δs	$\in$	Suspensions	=	$List$(Suspension)
ωs	$\in$	WorkQ	=	$List$(Work)

Given $\omega_0 =$ (), [eval(E)], [ ], main

Machine State	Value	Code	Environments	WorkQ	Store	Status
Initial State	_	[sched]	[ ]	$[\omega_0]$	{}	_
Final State	ν	[ ]	[ ]	[ ]	σ	main
Error State	err	ιs	ρs	ωs	σ	θ

Figure 3: Run-time Objects and the Machine State.

forms of evaluation strategies, both environment slots and store locations may contain other objects that need to be differentiated from values by means of various tags.

An unevaluated environment slot is either **empty** pointing to an expression to be evaluated within the current environment (eager semantics), or it carries a **thunk** closure (lazy semantics). Under eager semantics we need not create lexical closures for every bound identifier since each bound identifier is guaranteed to be evaluated eagerly and therefore the free identifiers referenced in their expressions would still be present in scope.

Similarly, a freshly allocated store location is either **empty** (I-structure semantics) or contains a **thunk** closure (functional semantics). The tag is changed to **defer** during evaluation and any read operations during this period are suspended. Once the evaluation completes, the value is stored back reactivating any suspended read operations and the tag is changed to **full**. Reactivated suspensions are collected into a queue of ready work.

Machine State

The state of the machine may be described by a 6-tuple $(\nu, \iota s, \rho s, \omega s, \sigma, \theta)$ consisting of a single accumulator register that carries the result value ν of each evaluation step, a code sequence ιs denoting the sequence of interpreter instructions yet to be executed, a stack of linked environment frames ρs, a queue of ready work ωs, the store σ, and a status flag θ denoting whether the current computation is the main thread of the computation or not.

Figure 3 also shows the initial, the final and the error states of the machine. The machine starts by scheduling an initial work ω_0 to evaluate the entire program expression E with an empty environment stack and an empty store. An underscore (_) denotes a "don't care" situation.

sched	::	Schedule the next work out of the work queue
eval(E)	::	Evaluate E in the current environment and leave result into the accumulator
update(x, ρ)	::	Update x in environment ρ with the current accumulator value and distribute the result to its pending suspensions
pushenv(ρs)	::	Push the environment frames ρs on the current stack
popenv$(\underline{n})$	::	Pop n environment frames from the current stack
$PF^n(x_1 \ldots x_n)$	::	Apply strict primitive operator PF^n to $x_1 \ldots x_n$
ap(f, x)	::	Apply the closure f to x
pap$(f, x_1 \ldots x_n)$	::	Partially apply the closure f to $x_1 \ldots x_n$
fap$(f, x_1 \ldots x_n)$	::	Fully apply the closure f to $x_1 \ldots x_n$
switch$(x, \iota s_1, \ldots, \iota s_n)$	::	Switch to the instruction stream indexed by x

Figure 4: Interpreter Instructions.

At every step the machine executes the next instruction from the code sequence and it halts when there are no more instructions to execute. Upon termination of the main thread, the final value is left in the accumulator which may contain pointers into the final store. Note that if the main thread terminates with an answer but the work queue is not yet empty, we have a choice whether or not to run the remaining threads. The initial thread ω_0 shown in Figure 3 would not schedule the remaining threads, but we may choose to do so explicitly. This choice has implications only for the termination behavior of the machine and not for producing the answer. In this sense, our machine models *non-strict* execution: we may get a result from the main thread, but the machine may continue executing. It is also possible to have a *deadlock* when there are suspended computations in the environment or in the heap and there is no work to be done.

An error state may be reached at any point during execution due to exceptional conditions such as a type mismatch, an arithmetic overflow, out of bounds access to data structures, or multiple assignment to I-structure components.

Instruction Set

The instruction set used by our machine is shown in Figure 4. Most of the instructions are self-explanatory. More detailed explanation follows in the next section.

4 A Mixed Interpreter for the Kernel Language

In this section we define a mixed interpreter for the fully parallel evaluation model shown in Figure 1 based on the framework given above. The interpreter operates mostly in a sequential, demand-driven style because it dynamically orders the various redexes of a partially-ordered parallel computation into a total order respecting the data dependencies among them. The interpreter, however, must still simulate parallel redex management because computations involving I-structures require multiple threads of control: producers of a data structure may be completely independent from its consumers and therefore must be spawned and executed independently[2]. Informally, our

[2] On the other hand, in a lazy functional language it is possible to maintain a single thread of control by performing only demand-driven evaluation.

interpreter maintains a queue of ready redexes, one of which is reduced at each step and new redexes are thrown back into the queue. A parallel implementation then simply becomes a matter of distributing ready redexes across several such interpreters.

We will define the interpreter as a set of state transition rules based on the current instruction at the head of the code sequence within the machine state.

4.1 Scheduling Work

We start with the sched instruction which schedules a new piece of work from the work queue.

sched:

$$\frac{\langle _ \quad \mathsf{sched} : \iota s \quad \rho s \quad \omega : \omega s \quad \sigma \quad \theta \rangle}{\langle \nu' \quad \iota s' \quad \rho s' \quad \omega s \quad \sigma \quad \theta' \rangle}$$

where $\omega = (\nu', \iota s', \rho s', \theta')$

The accumulator value, the environment stack, the status flag, and the code sequence appearing after the sched instruction are all thrown away and the corresponding new objects are loaded from the scheduled work. The store and the remaining work queue remain unchanged as expected.

Although the rule suggests that the next task at the head of the work queue gets scheduled, it is possible to schedule some tasks at a higher priority than others. In particular, we may want to bias the execution of tasks present within the work queue so that those that are directly required to produce the result along the main thread of execution have higher priority over other tasks. The main point of doing so is to ensure that the interpreter is normalizing, *i.e.*, if there exists an answer then the interpreter is able to find it. However, due to the presence of I-structures, the main thread may block on some heap location which may be filled by a side task. Therefore, the side tasks should also be able to execute in a fair schedule without monopolizing the machine when there is no other way of making progress. All we need to ensure is that, once the main thread is reactivated, it does not wait indefinitely for being scheduled. This normalizing scheduling strategy is enforced in our machine *via* the status flag θ and will be described along with the evaluation rule for function applications.

4.2 Expression Evaluation

The eval instruction is the core of the interpreter. It evaluates each kind of expression defined in the language. We will describe them one by one.

Constants

eval constant:

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

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

Identifiers

eval identifier (full):

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

eval identifier (empty):

$$\frac{\langle _ \quad \mathsf{eval}(x) : \iota s \quad \rho_0 : \ldots : \rho_k[x \mapsto \langle \mathsf{empty}\ E\rangle] : \rho s \quad \omega s \quad \sigma \quad \theta\rangle}{\langle _ \quad \iota s' \quad \rho_k[x \mapsto \langle \mathsf{defer}\ [\,]\rangle] : \rho s \quad \omega s \quad \sigma \quad \theta\rangle}$$

where $\iota s' = \mathsf{eval}(E) : \mathsf{update}(x, \rho_k) : \mathsf{pushenv}(\rho_0 \ldots \rho_{k-1}) : \iota s$

eval identifier (thunk):

$$\frac{\langle _ \quad \mathsf{eval}(x) : \iota s \quad \rho s[x \mapsto \langle \mathsf{thunk}\ \omega\rangle] \quad \omega s \quad \sigma \quad \theta\rangle}{\langle \nu \quad \iota s' \quad \rho s' \quad \omega s \quad \sigma \quad \theta\rangle}$$

where $\omega = (\nu, \iota s', \rho s', _)$
$\delta = (\iota s, \rho s[x \mapsto \langle \mathsf{defer}\ [\delta]\rangle], \theta)$

eval identifier (defer):

$$\frac{\langle _ \quad \mathsf{eval}(x) : \iota s \quad \rho s[x \mapsto \langle \mathsf{defer}\ \delta s\rangle] \quad \omega s \quad \sigma \quad \theta\rangle}{\langle _ \quad [\mathsf{sched}] \quad \rho s[x \mapsto \langle \mathsf{defer}\ \delta : \delta s\rangle] \quad \omega s \quad \sigma \quad \theta\rangle}$$

where $\delta = (\iota s, \rho s, \theta)$

Evaluation of an identifier first searches for its slot in the current stack of environment frames and then takes appropriate action according to its tag. If the slot is full, its contents are immediately loaded into the accumulator. If the slot is empty pointing to an expression, it is prepared for immediate evaluation under its definition environment along with subsequent updation of the environment slot. The current environment is restored after the evaluation. If the environment slot carries a thunk, then the current code sequence is suspended on the environment slot and the thunk is immediately scheduled with the current machine status. Finally, if the slot is already being evaluated (defer) then the current code sequence is simply suspended.

Since environments may be shared, the effect of tag or content manipulations on environment slots should be visible globally. Practically, an environment may be implemented as an array of frame slots whose tag and contents may be updated in place, or they may be implemented as a mapping from identifiers to fixed store locations whose contents may be updated.

Strict Primitive Functions

eval strict PF:

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

The evaluation of strict primitive function applications proceeds a little differently from non-strict ones. In the case of a strict primitive function, first we evaluate all its arguments and then we execute the primitive application. The argument evaluations may also be done in parallel. The evaluation of the various strict primitive functions is described below.

+:

$$\frac{\langle _ \quad +(x_1, x_2) : \iota s \quad \rho s[x_1 \mapsto \langle \mathsf{full}\ \underline{m}\rangle, x_2 \mapsto \langle \mathsf{full}\ \underline{n}\rangle] \quad \omega s \quad \sigma \quad \theta\rangle}{\langle \underline{m+n} \quad \iota s \quad \rho s \quad \omega s \quad \sigma \quad \theta\rangle}$$

Arithmetic functions simply perform the desired operation and leave the result in the accumulator.

`alloc`:

$$\frac{\langle _ \quad \texttt{alloc}(x) : \iota s \quad \rho s[x \mapsto \langle \textsf{full}\ \underline{n} \rangle] \quad \omega s \quad \sigma \quad \theta \rangle}{\langle \ell \quad \iota s \quad \rho s \quad \omega s \quad \sigma' \quad \theta \rangle}$$

where $\sigma' = \sigma + \{\ell \mapsto \langle \textsf{empty} \rangle, \ldots, (\ell + n - 1) \mapsto \langle \textsf{empty} \rangle\}$

The `alloc` function allocates n new locations in the store and initializes them to be **empty**. These locations may be used as I-structures or M-structures.

`I-fetch` (full):

$$\frac{\langle _ \quad \texttt{I-fetch}(x) : \iota s \quad \rho s[x \mapsto \langle \textsf{full}\ \ell \rangle] \quad \omega s \quad \sigma[\ell \mapsto \langle \textsf{full}\ \nu \rangle] \quad \theta \rangle}{\langle \nu \quad \iota s \quad \rho s \quad \omega s \quad \sigma \quad \theta \rangle}$$

`I-fetch` (empty):

$$\frac{\langle _ \quad \texttt{I-fetch}(x) : \iota s \quad \rho s[x \mapsto \langle \textsf{full}\ \ell \rangle] \quad \omega s \quad \sigma[\ell \mapsto \langle \textsf{empty} \rangle] \quad \theta \rangle}{\langle _ \quad [\textsf{sched}] \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \textsf{defer}\ [\delta] \rangle] \quad \theta \rangle}$$

where $\delta = (\iota s, \rho s, \theta)$

`I-fetch` (thunk):

$$\frac{\langle _ \quad \texttt{I-fetch}(x) : \iota s \quad \rho s[x \mapsto \langle \textsf{full}\ \ell \rangle] \quad \omega s \quad \sigma[\ell \mapsto \langle \textsf{thunk}\ \omega \rangle] \quad \theta \rangle}{\langle _ \quad [\textsf{sched}] \quad \rho s \quad \omega : \omega s \quad \sigma[\ell \mapsto \langle \textsf{defer}\ [\delta] \rangle] \quad \theta \rangle}$$

where $\delta = (\iota s, \rho s, \theta)$

`I-fetch` (defer):

$$\frac{\langle _ \quad \texttt{I-fetch}(x) : \iota s \quad \rho s[x \mapsto \langle \textsf{full}\ \ell \rangle] \quad \omega s \quad \sigma[\ell \mapsto \langle \textsf{defer}\ \delta s \rangle] \quad \theta \rangle}{\langle _ \quad [\textsf{sched}] \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \textsf{defer}\ \delta : \delta s \rangle] \quad \theta \rangle}$$

where $\delta = (\iota s, \rho s, \theta)$

The evaluation of `I-fetch`, `hd`, `tl`, and `M-take` is similar in nature. In each case, the course of action is determined by the status of the store location being accessed. In the case of `I-fetch`, `hd`, and `tl` operations, if the location is **full**, its contents are loaded into the accumulator immediately without changing its status. If the location is **empty** or **defer**, the current code sequence is suspended. If the location carries a **thunk**, it is pushed into the work queue. The evaluation of the `M-take` operation is similar, except that it leaves the location **empty** after a successful `M-take` operation.

`I-store` (empty):

$$\frac{\langle _ \quad \texttt{I-store}(x, z) : \iota s \quad \rho s[x \mapsto \langle \textsf{full}\ \ell \rangle, z \mapsto \langle \textsf{full}\ \nu \rangle] \quad \omega s \quad \sigma[\ell \mapsto \langle \textsf{empty} \rangle] \quad \theta \rangle}{\langle () \quad \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \textsf{full}\ \nu \rangle] \quad \theta \rangle}$$

`I-store` (defer):

$$\frac{\langle _ \quad \texttt{I-store}(x, z) : \iota s \quad \rho s[x \mapsto \langle \textsf{full}\ \ell \rangle, z \mapsto \langle \textsf{full}\ \nu \rangle] \quad \omega s \quad \sigma[\ell \mapsto \langle \textsf{defer}\ \delta s \rangle] \quad \theta \rangle}{\langle () \quad \iota s \quad \rho s \quad \omega s \mathbin{+\!\!+} \omega s' \quad \sigma[\ell \mapsto \langle \textsf{full}\ \nu \rangle] \quad \theta \rangle}$$

where $\omega s' = [(\nu, \iota s, \rho s, \theta) \mid (\iota s, \rho s, \theta) \leftarrow \delta s]$

The evaluation of `I-store` and `M-put` is similar. Both of them test the status of the store location being updated. If the location is already **full** or carries a **thunk**, it leads to a run-time error (not shown). If the location is **empty**, the argument value is stored into it and the status is changed to full. If the location contains one or more suspensions (**defer**), the `I-store` operation reactivates all of them and puts them into the work queue as shown, while the `M-put` operation reactivates only one of them (not shown).

Non-Strict Primitive Functions

The only non-strict primitive function we have is the `Cons` constructor.

`Cons`:

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

where $\sigma' = \sigma + \{\ell \mapsto \langle \mathsf{empty} \rangle, (\ell + 1) \mapsto \langle \mathsf{empty} \rangle\}$
$\omega_1 = (_, [\mathsf{eval}(x_1) : \mathtt{I\text{-}store}(\ell, x_1) : \mathsf{sched}], \rho s, \mathsf{apply\text{-}1})$
$\omega_2 = (_, [\mathsf{eval}(x_2) : \mathtt{I\text{-}store}(\ell + 1, x_2) : \mathsf{sched}], \rho s, \mathsf{apply\text{-}1})$

The `Cons` constructor 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 fill these locations eagerly. Under lazy evaluation, on the other hand, we would have stored thunks in the locations themselves that would be evaluated on demand.

λ-abstraction

eval λ-expression:

$$\frac{\langle _ \quad \mathsf{eval}(\lambda x_1 \ldots x_n . E) : \iota s \quad \rho s \quad \omega s \quad \sigma \quad \theta \rangle}{\langle \langle \mathsf{clsr}\ (x_1 \ldots x_n).\ E, [\rho] \rangle \quad \iota s \quad \rho s \quad \omega s +\!\!+ \omega s' \quad \sigma \quad \theta \rangle}$$

where $y_1 \ldots y_m = FV(\lambda x_1 \ldots x_n . E)$
$\rho = \{\ y_1 \mapsto \langle \mathsf{defer}\ [\,] \rangle, \ldots, y_m \mapsto \langle \mathsf{defer}\ [\,] \rangle\ \}$
$\omega s' = [(_, [\mathsf{eval}(y_1) : \mathsf{update}(y_1, \rho) : \mathsf{sched}], \rho s, \mathsf{apply\text{-}1}) :$
$\cdots :$
$(_, [\mathsf{eval}(y_m) : \mathsf{update}(y_m, \rho) : \mathsf{sched}], \rho s, \mathsf{apply\text{-}1})]$

λ-expressions are evaluated into a closure value. A new lexical environment is allocated that contains slots for the free identifiers of the λ-body. Separate tasks are added to the work queue to fill these new environment slots. Under lazy evaluation, we would have stored thunks into the new slots that would copy the values on demand.

An interesting aspect of eager evaluation is that we may spawn the evaluation of the free identifiers of a λ-expression without having demanded their value. This allows us to ensure that free identifiers of a function closure never refer directly to their original definition environments as shown above. We give each closure its own environment into which the values of its free identifiers are copied. This property clearly separates the lifetime of the original environments of the free identifiers from the lifetime of the environment captured within the closure allowing us to deallocate the original environments when all the spawned tasks associated with them have terminated. In [1] we shows a systematic way of generating termination signals from every spawned task so that such resource management may be scheduled by the compiler.

Function Application

Evaluation of function applications is the most interesting and the most complex part of the interpreter. First, the function is evaluated to a closure. Then depending on whether or not its arity is satisfied, either a new closure is built or a function application is performed. We only show the rules for the general application; the rules for partial application and full application are similar.

eval Ap:

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

ap unsatisfied:

$$\frac{\langle _ \quad \mathsf{ap}(f,z) : \iota s \quad \rho s[f \mapsto \langle \mathsf{full}\ \langle \mathsf{clsr}\ (x_1 \ldots x_n).\ E, \rho s' \rangle \rangle] \quad \omega s \quad \sigma \quad \theta \rangle}{\langle \langle \mathsf{clsr}\ (x_2 \ldots x_n).\ E, \rho' : \rho s' \rangle \quad \iota s \quad \rho s \quad \omega s +\!\!+ [\omega] \quad \sigma \quad \theta \rangle}$$

where $\rho' = \{ x_1 \mapsto \langle \mathsf{defer}\ [\,] \rangle \}$
$\omega = (_, [\mathsf{eval}(z) : \mathsf{update}(x_1, \rho') : \mathsf{sched}], \rho s, \mathsf{apply\text{-}1})$

ap satisfied (main):

$$\frac{\langle _ \quad \mathsf{ap}(f,z) : \iota s \quad \rho s[f \mapsto \langle \mathsf{full}\ \langle \mathsf{clsr}\ (x_n).\ E, \rho s' \rangle \rangle] \quad \omega s \quad \sigma \quad \mathsf{main} \rangle}{\langle _ \quad \mathsf{eval}(E) : \mathsf{popenv}(\underline{m}) : \iota s \quad (\rho' : \rho s') +\!\!+ \rho s \quad \omega s +\!\!+ [\omega] \quad \sigma \quad \mathsf{main} \rangle}$$

where $\rho' = \{ x_n \mapsto \langle \mathsf{defer}\ [\,] \rangle \}$
$\omega = (_, [\mathsf{eval}(z) : \mathsf{update}(x_n, \rho') : \mathsf{sched}], \rho s, \mathsf{apply\text{-}1})$
$m = \mathit{length}(\rho' : \rho s')$

ap satisfied (apply-1):

$$\frac{\langle _ \quad \mathsf{ap}(f,z) : \iota s \quad \rho s[f \mapsto \langle \mathsf{full}\ \langle \mathsf{clsr}\ (x_n).\ E, \rho s' \rangle \rangle] \quad \omega s \quad \sigma \quad \mathsf{apply\text{-}1} \rangle}{\langle _ \quad \mathsf{eval}(E) : \mathsf{popenv}(\underline{m}) : \iota s \quad (\rho' : \rho s') +\!\!+ \rho s \quad \omega s +\!\!+ [\omega] \quad \sigma \quad \mathsf{apply\text{-}0} \rangle}$$

where $\rho' = \{ x_n \mapsto \langle \mathsf{defer}\ [\,] \rangle \}$
$\omega = (_, [\mathsf{eval}(z) : \mathsf{update}(x_n, \rho') : \mathsf{sched}], \rho s, \mathsf{apply\text{-}1})$
$m = \mathit{length}(\rho' : \rho s')$

ap satisfied (apply-0):

$$\frac{\langle \nu \quad \mathsf{ap}(f,z) : \iota s \quad \rho s[f \mapsto \langle \mathsf{full}\ \langle \mathsf{clsr}\ (x_n).\ E, \rho s' \rangle \rangle] \quad \omega s \quad \sigma \quad \mathsf{apply\text{-}0} \rangle}{\langle \nu \quad [\mathsf{sched}] \quad \rho s \quad \omega s +\!\!+ [\omega] \quad \sigma \quad \mathsf{apply\text{-}0} \rangle}$$

where $\omega = (\nu, \mathsf{ap}(f,z) : \iota s, \rho s, \mathsf{apply\text{-}1})$

If the arity of the closure is not satisfied, then a new closure needs to be built. This operation is similar to the evaluation of a λ-expression. A new environment slot is allocated that would receive the value of the supplied argument and a task is spawned to fill this slot. A new closure is built using this environment.

If the arity of the closure is satisfied, then the course of action depends on the status flag of the machine θ. The status flag denotes how many function applications are allowed for the current thread. If the status is main, then an arbitrary number of function applications are allowed. In case of apply-1 status, only one (current) application is allowed: the function body is evaluated under the reduced status apply-0. An application with the status apply-0 is not expanded, instead the current task is descheduled and is thrown back into the work queue with the increased status apply-1. This ensures that if the main thread gets blocked, then all other tasks get a fair chance to execute. Once the main thread is rescheduled, it executes to completion or until it gets blocked again.

If an application is allowed to proceed then it executes as follows. A new environment slot is allocated for the argument and a task is spawned to fill it. Next, the body expression is setup for evaluation under the appropriate environment chain obtained from the closure. Finally, the original caller environment is restored by popping the additional environment frames.

Case Expression

eval `Case`:

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

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

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

The evaluation of the `Case` expression proceeds by first evaluating the dispatch expression. This must yield an integer within the range of the dispatch which is used to select the appropriate branch. We also allow a boolean switch instruction that selects one of two instruction streams based on the boolean value of the first argument.

Block Expression

eval block:

$$\frac{\langle _ \quad \mathsf{eval}(\{\ x_1 = E_1; \ldots; x_n = E_n\ \mathtt{in}\ (x \mid y_1 \ldots y_m)\ \}) : \iota s \quad \rho s \quad \omega s \quad \sigma \quad \theta\rangle}{\langle _ \quad \mathsf{eval}(x) : \mathsf{popenv}(1) : \iota s \quad \rho : \rho s \quad \omega s +\!\!+\, \omega s' \quad \sigma \quad \theta\rangle}$$

where $\rho = \{\ x_1 \mapsto \langle \mathsf{empty}\ E_1\rangle, \ldots, x_n \mapsto \langle \mathsf{empty}\ E_n\rangle\ \}$

$$\begin{aligned} \omega s' = \ & [(_, [\mathsf{eval}(y_1) : \mathsf{sched}], \rho : \rho s, \mathsf{apply\text{-}1}) : \\ & \cdots : \\ & (_, [\mathsf{eval}(y_m) : \mathsf{sched}], \rho : \rho s, \mathsf{apply\text{-}1})] \end{aligned}$$

Finally, the evaluation of a block expression creates a new environment frame with a slot for each block binding. The main result expression is set for subsequent evaluation. Auxiliary tasks are added to the work queue to evaluate the other roots. These would be scheduled when the main thread cannot make any progress. Note that each environment slot is initialized directly with the expression that would compute its value. This is much simpler than creating a thunk for each slot since all the free identifiers of the expression are guaranteed to be accessible through the enclosing lexical environment stack.

4.3 Book-keeping Instructions

A few other instructions are used by the interpreter for book-keeping.

update identifier:

$$\frac{\langle \nu \quad \mathsf{update}(x, \rho_k) : \iota s \quad \rho_0 : \ldots : \rho_k[x \mapsto \langle \mathsf{empty}\ _\rangle] : \rho s \quad \omega s \quad \sigma \quad \theta\rangle}{\langle \nu \quad \iota s \quad \rho_0 : \ldots : \rho_k[x \mapsto \langle \mathsf{full}\ \nu\rangle] : \rho s \quad \omega s \quad \sigma \quad \theta\rangle}$$

$$\frac{\langle \nu \quad \mathsf{update}(x, \rho_k) : \iota s \quad \rho_0 : \ldots : \rho_k[x \mapsto \langle \mathsf{defer}\ \delta s\rangle] : \rho s \quad \omega s \quad \sigma \quad \theta\rangle}{\langle \nu \quad \iota s \quad \rho_0 : \ldots : \rho_k[x \mapsto \langle \mathsf{full}\ \nu\rangle] : \rho s \quad \omega s +\!\!+\, \omega s' \quad \sigma \quad \theta\rangle}$$

where $\omega s' = [(\nu, \iota s, \rho s, \theta) \ \mid\ (\iota s, \rho s, \theta) \leftarrow \delta s]$

pushenv :

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

popenv :

$$\frac{\langle _ \quad \mathsf{popenv}(\underline{n}) : \iota s \quad \rho_0 : \ldots : \rho_{n-1} : \rho s \quad \omega s \quad \sigma \quad \theta \rangle}{\langle _ \quad \iota s \quad \rho s \quad \omega s \quad \sigma \quad \theta \rangle}$$

The update operation is usually performed immediately after the evaluation of an identifier. It updates the environment slot of the identifier with the value computed in the accumulator. It also distributes that value to all the waiting suspensions in that slot (if any) and adds them to the work queue. The pushenv and popenv operations manipulate the environment stack as shown. Popping an environment frame from the environment stack does not deallocate that frame. As discussed earlier, frame deallocation must be performed independently after all the spawned tasks that refer to that frame have terminated.

5 A Compiler for the Mixed Evaluation Model

In this section we will define a compiler for the mixed evaluation model based on the same machinery as described in Section 3. 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 code sequences called *threads*; and finally, we describe how these code sequences are executed on the abstract machine by giving additional execution rules for the new instructions.

5.1 Compilation Machinery

The compilation process involves a systematic separation of control and data of a program. We transform a program that is dynamically interpreted along with its data into a fixed sequence of instructions that may be repeatedly executed with different sets of data. Essentially, this involves a systematic elimination of all eval instructions while interpreting the program, replacing them with compiled sequences of low-level instructions, or threads, that compute the values of the original expressions.

Compilation also involves a precise specification and management of processor and memory resources: layout of environments and heap data structures, a mapping for all program identifiers to their corresponding environment slots, and scheduling of parallel tasks to fill these slots. We may also need additional instructions that address and manipulate the machine state.

The semantic categories and translation functions used during compilation are shown in Figure 5 and are discussed below.

Threaded Compilation

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).

$$
\begin{array}{llll}
\alpha & \in & \text{Identmap} & = \text{Identifier} \to (\text{Integer} \times \text{Integer}) \\
\tau & \in & \text{Threadmap} & = \text{Identifier} \to \text{Code} \\
\mathbf{TId}[\![\ ]\!] & :: & \multicolumn{2}{l}{\text{Identifier} \to \text{Threadmap} \to \text{Identmap} \to \text{Instruction}} \\
\mathbf{TExp}[\![\ ]\!] & :: & \multicolumn{2}{l}{\text{Expression} \to \text{Threadmap} \to \text{Identmap} \to (\text{Code} \times \text{Threadmap})}
\end{array}
$$

$$
\begin{array}{lrcl}
\text{Compilation:} & (\iota s, \tau) & = & \mathbf{TExp}[\![\text{Program}]\!]\ \{\ \}\ \{\ \} \\
\text{Initial Work:} & \omega_0 & = & (), \iota s, [\,], \mathsf{main}
\end{array}
$$

Figure 5: Compile-time Objects.

The precise compilation rule for each source language construct appear in Section 5.2. The threads are allowed to block and get rescheduled. No attempt is made to combine several threads into larger ones at this stage. Partitioning techniques [15, 7] may be used to create longer threads that are deadlock-free.

Environments and Identifiers

All bound identifiers within a block or a function are assigned a slot within an environment frame that is allocated at run-time. An identifier reference is compiled into a pair of index offsets $\langle i, j \rangle$ into the current environment stack. The first index i refers to the position of the environment frame from the top of the stack (zero based). The second index j gives the offset of the identifier slot within that frame (zero based). A mapping from identifier names to index-offset pairs (α) is maintained during compilation. It is also possible to use a different scheme where we preallocate the environment storage required by all blocks in a given procedure within a single flat environment that is allocated at the time of function application. Under this scheme, an identifier reference becomes a single offset into the current procedure frame. We do not use this scheme here for simplicity[3].

Machine State

We refer to components of the current machine state at compile-time using the following notation: the accumulator by $\Diamond$, the code sequence by code, the environment stack by stack, the work queue by workq, the global heap by heap, and the machine status by status. The top of the stack environment frame would be denoted by $\mathsf{stack}[0]$.

Additional Machine Instructions

The additional machine instructions are shown in Figure 6. The detailed explanations appear along with the execution rules given in Section 5.3.

5.2 Compilation Rules

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

[3] However, the actual Id compiler uses this scheme by computing the total number of slots needed by a function. It may also share slots between mutually exclusive sets of identifiers such as the branches of a `Case` expression.

$\mathsf{load}(c \mid x)$	::	Load a constant or the value of an environment slot into the accumulator (the slot must be **full**)
$\mathsf{enter}(x, \iota s)$	::	Check status of the environment slot x and execute thread ιs if necessary
$\mathsf{makeclsr}(\underline{n}, \iota s_c, \iota s_d, \rho s)$	::	Make a closure for the pair of threads $\iota s_c, \iota s_d$ with arity n and the environment stack ρs
apply	::	Test arity of the closure in the accumulator, applying it to the argument environment on stack, otherwise make a new closure
fapply	::	Apply the closure in the accumulator to the full argument environment on stack
$\mathsf{pushwork}(\iota s, \rho s)$	::	Make and push work with the thread ιs, the environment chain ρs, the current accumulator and apply-1 status
$\mathsf{allocletenv}(\underline{n})$	::	Allocate a new block environment with $0 \ldots (n-1)$ empty slots
$\mathsf{alloclamenv}(\underline{n})$	::	Allocate a new function environment with $0 \ldots (n-1)$ defer slots

Figure 6: Additional Compiler Instructions.

Constants

An immediate constant is simply loaded into the accumulator.

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

Identifiers

An identifier reference compiles into an enter instruction that accesses the right environment slot and computes its value.

$$\mathbf{TExp}[\![x]\!]\ \tau\ \alpha = [\mathbf{TId}[\![x]\!]\ \tau\ \alpha], \quad \phi$$

where

Block-bound Identifier:	$\mathbf{TId}[\![x]\!]\ \tau\ \alpha$	$=$	$\mathsf{enter}(\alpha(x), \tau(x))$
λ-bound Identifier:	$\mathbf{TId}[\![x]\!]\ \tau\ \alpha$	$=$	$\mathsf{enter}(\alpha(x), [\,])$

The threads corresponding to identifiers bound in a block would be known at compile time while the threads corresponding to function parameters must be dynamically linked. The two different compilations above reflect this difference.

Primitive Functions

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

$$\begin{array}{l}\mathbf{TExp}[\![PF^n(x_1 \ldots x_n)]\!]\ \tau\ \alpha = \\ \quad [(\mathbf{TId}[\![x_1]\!]\ \tau\ \alpha) : \ldots : (\mathbf{TId}[\![x_n]\!]\ \tau\ \alpha) : PF^n(\alpha(x_1), \ldots, \alpha(x_n))], \quad \phi\ \}\end{array}$$

Non-strict primitives, such as `Cons` are compiled directly, without first evaluating their arguments. The run-time evaluation rule shown in Section 4 takes care of spawning the appropriate

tasks to fill the allocated heap locations.

$$\mathbf{TExp}[\![\texttt{Cons}(x_1, x_2)]\!]\ \tau\ \alpha = [\texttt{Cons}(\alpha(x_1), \alpha(x_2))], \quad \phi$$

λ-abstraction

The compilation of a function is the most complex part of the compiler. In addition to having the caller-callee linkage setup properly, we also wish to be able to optimize for full arity function applications as opposed to higher-order application. Therefore, we use two separate entry points into the function body: one *direct* entry point (ιs_d) for full arity applications, and another *chained* entry point (ιs_c) for higher-order applications.

$$
\begin{array}{l}
\mathbf{TExp}[\![\lambda x_1 \ldots x_n.\ E]\!]\ \tau\ \alpha = \\
\quad \{\ \ \iota s_b, \tau_b = \mathbf{TExp}[\![E]\!]\ \tau\ \alpha_b; \\
\quad\quad \iota s_d = \iota s_b \mathbin{+\!\!+} [\mathsf{update}(0, \mathsf{stack}[0]) : \mathsf{popenv}(2) : \mathsf{sched}]; \\
\quad\quad \iota s_{x_1} = [(\mathbf{TId}[\![x_1]\!]\ \tau\ \alpha_c) : \mathsf{update}(1, \mathsf{stack}[0]) : \mathsf{sched}]; \\
\quad\quad \ldots \\
\quad\quad \iota s_{x_n} = [(\mathbf{TId}[\![x_n]\!]\ \tau\ \alpha_c) : \mathsf{update}(\underline{n}, \mathsf{stack}[0]) : \mathsf{sched}]; \\
\quad\quad \iota s_c = \mathsf{pushwork}(\iota s_{x_1}, \mathsf{stack}) : \ldots : \mathsf{pushwork}(\iota s_{x_n}, \mathsf{stack}) : \\
\quad\quad\quad \mathsf{popenv}(\underline{n+1}) : \mathsf{pushenv}(\Diamond) : \iota s_d; \\
\quad\quad \iota s_{y_1} = [(\mathbf{TId}[\![y_1]\!]\ \tau\ \alpha') : \mathsf{update}(0, \mathsf{stack}[0]) : \mathsf{sched}]; \\
\quad\quad \ldots \\
\quad\quad \iota s_{y_m} = [(\mathbf{TId}[\![y_m]\!]\ \tau\ \alpha') : \mathsf{update}(\underline{m-1}, \mathsf{stack}[0]) : \mathsf{sched}]; \\
\quad\quad \mathtt{in}\ [\mathsf{alloclamenv}(\underline{m}) : \mathsf{pushwork}(\iota s_{y_1}, \Diamond : \mathsf{stack}) : \ldots : \mathsf{pushwork}(\iota s_{y_m}, \Diamond : \mathsf{stack}) : \\
\quad\quad\quad \mathsf{makeclsr}(\underline{n}, \iota s_c, \iota s_d, [\Diamond])], \quad \tau_b\ \}
\end{array}
$$

where

$$
\begin{array}{l}
y_1 \ldots y_m = FV(\lambda x_1 \ldots x_n.E) \\
\alpha_b = \{\ x_1 \mapsto \langle 0, 1\rangle, \ldots, x_n \mapsto \langle 0, \underline{n}\rangle, y_1 \mapsto \langle 1, \underline{0}\rangle, \ldots, y_m \mapsto \langle 1, \underline{m-1}\rangle\ \} \\
\alpha_c = \{\ x_1 \mapsto \langle \underline{n}, 0\rangle, \ldots, x_n \mapsto \langle 1, 0\rangle\ \} \\
\alpha'(x) = \langle \mathit{fst}(\alpha(x) + 1), \mathit{snd}(\alpha(x))\rangle
\end{array}
$$

Figure 7 shows the environment setup for the chained and direct points. The direct entry point ιs_d is setup under the assumption that all arguments to the function are in one flat environment frame and all its free identifiers are in another flat environment frame. This arrangement is reflected in the identifier map α_b which is used during the compilation of the function body. The first slot in the argument environment is reserved for storing the return continuation of the caller function at the time of function application. This return continuation is reactivated at the end of the function body by updating this slot with the final result.

For higher-order applications, the arguments present in the closure chain must be explicitly copied into the flat argument environment which is present at the top of the stack as well as in the accumulator. The chained entry point spawns off tasks to copy the arguments from the environment stack using a chain-destructuring identifier map α_c. Then it adjusts the environment stack and jumps to the direct entry point which executes the body of the function under the flat argument environment.

The λ-abstraction itself compiles to produce a closure which requires an environment frame to store its free identifiers. Following eager semantics, we arrange to push parallel tasks that would fill the values of the free identifiers into their appropriate slot in this frame.

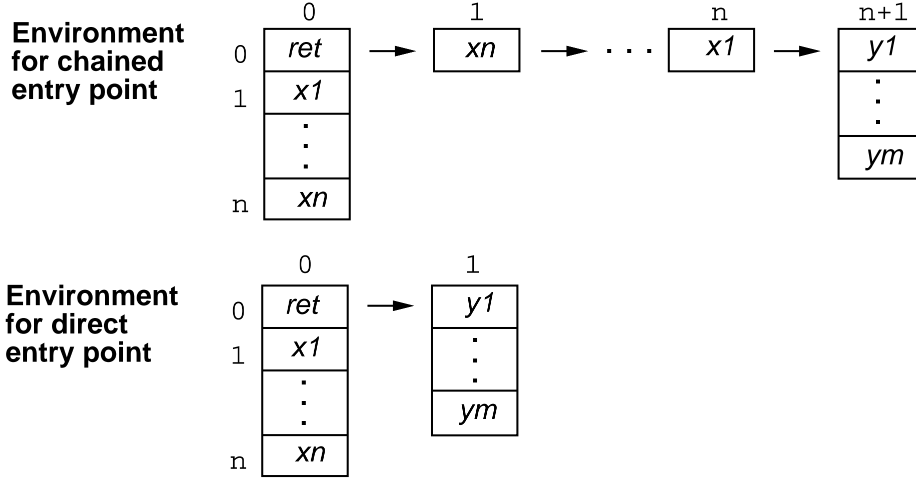


Figure 7: Environment setup for chained and direct entry points of a function.

Function Application

In a non-strict, eager function application we evaluate the function and the argument in parallel. A new 1-element environment is allocated for the argument and is filled in parallel. This is similar to the case of the free identifiers of a λ-abstraction. The function is then evaluated to a closure whose arity determines whether the application scheme would evaluate the body thread (arity satisfied) or build a new closure (arity unsatisfied). This is described in the evaluation rule for application given in Section 5.3.

$$\begin{aligned}&\mathbf{TExp}[\![\texttt{Ap}(f,z)]\!]\ \tau\ \alpha =\\ &\quad\{\ \ \iota s_z = [(\mathbf{TId}[\![z]\!]\ \tau\ \alpha') : \mathsf{update}(0,\mathsf{stack}[0]) : \mathsf{sched}];\\ &\quad\quad \texttt{in}\ [\mathsf{alloclamenv}(1) : \mathsf{pushenv}(\diamond) : \mathsf{pushwork}(\iota s_z,\mathsf{stack}) : (\mathbf{TId}[\![f]\!]\ \tau\ \alpha') : \mathsf{apply}],\quad \phi\ \}\end{aligned}$$

where

$$\alpha'(x) = \langle \mathit{fst}(\alpha(x)+1), \mathit{snd}(\alpha(x))\rangle$$

The full application scheme is similar, except that it builds the final argument environment in a single shot instead of argument by argument. It also reserves the first slot in the argument environment to store the return continuation of the caller.

$$\begin{aligned}&\mathbf{TExp}[\![\texttt{Fap}^n(f,z_1,\ldots,z_n)]\!]\ \tau\ \alpha =\\ &\quad\{\ \ \iota s_{z_1} = [(\mathbf{TId}[\![z_1]\!]\ \tau\ \alpha') : \mathsf{update}(1,\mathsf{stack}[0]) : \mathsf{sched}];\\ &\quad\quad \ldots\\ &\quad\quad \iota s_{z_n} = [(\mathbf{TId}[\![z_n]\!]\ \tau\ \alpha') : \mathsf{update}(\underline{n},\mathsf{stack}[0]) : \mathsf{sched}];\\ &\quad\quad \texttt{in}\ [\mathsf{alloclamenv}(\underline{n+1}) : \mathsf{pushenv}(\diamond) : \mathsf{pushwork}(\iota s_{z_1},\mathsf{stack}) : \ldots :\\ &\quad\quad\quad \mathsf{pushwork}(\iota s_{z_n},\mathsf{stack}) : (\mathbf{TId}[\![f]\!]\ \tau\ \alpha') : \mathsf{fapply}],\quad \phi\ \}\end{aligned}$$

where

$$\alpha'(x) = \langle \mathit{fst}(\alpha(x)+1), \mathit{snd}(\alpha(x))\rangle$$

Case Expression

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

$$
\begin{array}{l}
\mathbf{TExp}[\![\mathtt{Case}(x, E_1, \ldots, E_n)]\!]\ \tau\ \alpha = \\
\quad \{\ \ \iota s_1, \tau_1 = \mathbf{TExp}[\![E_1]\!]\ \tau\ \alpha; \\
\quad\quad \cdots \\
\quad\quad \iota s_n, \tau_n = \mathbf{TExp}[\![E_n]\!]\ \tau\ \alpha; \\
\quad\quad \mathtt{in}\ [(\mathbf{TId}[\![x]\!]\ \tau\ \alpha) : \mathsf{switch}(\alpha(x), \iota s_1, \ldots, \iota s_n)], \quad \tau_1 + \cdots + \tau_n\ \}
\end{array}
$$

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 then compiled into threads that update the appropriate environment slots with their respective values. 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.

Some (or all) of the bound identifiers of a block may be designated as roots that are eagerly spawned by creating parallel tasks for them. The result of the block is computed by its primary root.

$$
\begin{array}{l}
\mathbf{TExp}[\![\{\ x_1 = E_1; \cdots; x_n = E_n\ \mathtt{in}\ (x \mid y_1 \ldots y_m)\ \}]\!]\ \tau\ \alpha = \\
\quad \{\ \ \iota s_1, \tau_1 = \mathbf{TExp}[\![E_1]\!]\ \tau'\ \alpha'; \\
\quad\quad \iota s_1' = \iota s_1 +\!\!+ [\mathsf{update}(0, \mathsf{stack}[0]) : \mathsf{sched}]; \\
\quad\quad \cdots \\
\quad\quad \iota s_n, \tau_n = \mathbf{TExp}[\![E_n]\!]\ \tau'\ \alpha'; \\
\quad\quad \iota s_n' = \iota s_n +\!\!+ [\mathsf{update}(\underline{n-1}, \mathsf{stack}[0]) : \mathsf{sched}]; \\
\quad\quad \mathtt{in}\ [\mathsf{allocletenv}(\underline{n}) : \mathsf{pushenv}(\diamond) : \mathsf{pushwork}(\tau'(y_1), \mathsf{stack}) : \ldots : \\
\quad\quad\quad \mathsf{pushwork}(\tau'(y_m), \mathsf{stack}) : (\mathbf{TId}[\![x]\!]\ \tau'\ \alpha') : \mathsf{popenv}(1)], \quad \tau_b + \tau_1 + \cdots + \tau_n\ \}
\end{array}
$$

where

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

5.3 Execution Rules for Additional Instructions

In this section we provide the execution rules for the additional instructions given in Figure 6. Note that identifier names are now translated into index-offset pairs $\langle i, j\rangle$ pointing into the environment stack.

Loading Accumulator

The load instruction is used to load an immediate constant or a previously computed value into the accumulator.

load constant:

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

load identifier:

$$\frac{\langle _ \quad \mathsf{load}(\langle i, j \rangle) : \iota s \quad \rho_0 : \ldots : \rho_i[\ldots \langle \mathsf{full}\ \nu \rangle_j \ldots] : \rho s \quad \omega s \quad \sigma \quad \theta \rangle}{\langle \nu \quad \iota s \quad \rho_0 : \ldots : \rho_i : \rho s \quad \omega s \quad \sigma \quad \theta \rangle}$$

Evaluating Identifiers

The enter instruction checks and loads the value of an identifier into the accumulator, computing its value if necessary. If the identifier is block-bound then the enter instruction directly points to thread to compute its value. If the identifier is currently unevaluated, then the current instruction stream is temporarily suspended (on the identifier slot) and the evaluation is carried out inline. The final updation of the slot with the identifier's value would reactivate the suspension.

For λ-bound identifiers, the function application is responsible for spawning a separate task to compute the value and fill the identifier slot, therefore the slot would be marked as defer. In this case, the current thread is simply suspended.

enter identifier (full):

$$\frac{\langle _ \quad \mathsf{enter}(\langle i, j \rangle, _) : \iota s \quad \rho_0 : \ldots : \rho_i[\ldots \langle \mathsf{full}\ \nu \rangle_j \ldots] : \rho s \quad \omega s \quad \sigma \quad \theta \rangle}{\langle \nu \quad \iota s \quad \rho_0 : \ldots : \rho_i : \rho s \quad \omega s \quad \sigma \quad \theta \rangle}$$

enter identifier (empty):

$$\frac{\langle _ \quad \mathsf{enter}(\langle i, j \rangle, \iota s_x) : \iota s \quad \rho_0 : \ldots : \rho_i[\ldots \langle \mathsf{empty} \rangle_j \ldots] : \rho s \quad \omega s \quad \sigma \quad \theta \rangle}{\langle _ \quad \iota s_x \quad \rho_i[\ldots \langle \mathsf{defer}\ [\delta] \rangle_j \ldots] : \rho s \quad \omega s \quad \sigma \quad \theta \rangle}$$

where $\rho s' = \rho_0 : \ldots : \rho_i : \rho s$
$\delta = (\iota s, \rho s', \theta)$

enter identifier (thunk):

$$\frac{\langle _ \quad \mathsf{enter}(\langle i, j \rangle, _) : \iota s \quad \rho_0 : \ldots : \rho_i[\ldots \langle \mathsf{thunk}\ \omega \rangle_j \ldots] : \rho s \quad \omega s \quad \sigma \quad \theta \rangle}{\langle \nu \quad \iota s' \quad \rho s' \quad \omega s \quad \sigma \quad \theta \rangle}$$

where $\omega = (\nu, \iota s', \rho s', _)$
$\delta = (\iota s, \rho_0 : \ldots : \rho_i[\ldots \langle \mathsf{defer}\ [\delta] \rangle_j \ldots] : \rho s, \theta)$

enter identifier (defer):

$$\frac{\langle _ \quad \mathsf{enter}(\langle i, j \rangle, _) : \iota s \quad \rho_0 : \ldots : \rho_i[\ldots \langle \mathsf{defer}\ \delta s \rangle_j \ldots] : \rho s \quad \omega s \quad \sigma \quad \theta \rangle}{\langle _ \quad [\mathsf{sched}] \quad \rho_0 : \ldots : \rho_i[\ldots \langle \mathsf{defer}\ \delta : \delta s \rangle_j \ldots] : \rho s \quad \omega s \quad \sigma \quad \theta \rangle}$$

where $\delta = (\iota s, \rho_0 : \ldots : \rho_i[\ldots \langle \mathsf{defer}\ \delta : \delta s \rangle_j \ldots] : \rho s, \theta)$

Closure Creation

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

makeclsr:

$$\frac{\langle _ \quad \mathsf{makeclsr}(\underline{n}, \iota s_c, \iota s_d, \rho s') : \iota s \quad \rho s \quad \omega s \quad \sigma \quad \theta \rangle}{\langle \langle \mathsf{clsr}\ (\underline{n}).\ (\underline{n}, \iota s_c, \iota s_d), \rho s' \rangle \quad \iota s \quad \rho s \quad \omega s \quad \sigma \quad \theta \rangle}$$

Function Application

There are two forms of function applications: general application (apply) and full arity application (fapply). In each case, the application is allowed to proceed only if the machine has either main or apply-1 status flag, otherwise the current thread is descheduled. This is in accordance with the normalization scheduling strategy discussed earlier.

General function application must test the arity on the closure (in the accumulator) and construct another closure with one less remaining arity or expand the function application. In the latter case, first a flat argument environment is allocated for the function call, saving the return continuation of the caller in its first slot. This would enable the callee to resume the caller after the function call has completed. Then the chain destructuring entry point present within the closure is scheduled which is responsible for destructuring the closure chain into the flat argument environment.

apply unsatisfied ($m > 1$):

$$\frac{\langle\langle \mathsf{clsr}\ (\underline{m}).\ (\underline{n}, \iota s_c, \iota s_d), \rho s'\rangle \quad \mathsf{apply} : \iota s \quad \rho' : \rho s \quad \omega s \quad \sigma \quad \theta\rangle}{\langle\langle \mathsf{clsr}\ (\underline{m-1}).\ (\underline{n}, \iota s_c, \iota s_d), \rho' : \rho s'\rangle \quad \iota s \quad \rho s \quad \omega s \quad \sigma \quad \theta\rangle}$$

apply satisfied (main):

$$\frac{\langle\langle \mathsf{clsr}\ (1).\ (\underline{n}, \iota s_c, \iota s_d), \rho s'\rangle \quad \mathsf{apply} : \iota s \quad \rho' : \rho s \quad \omega s \quad \sigma \quad \mathsf{main}\rangle}{\langle \rho_r \quad \iota s_c \quad \rho_r : \rho' : \rho s' \quad \omega s \quad \sigma \quad \mathsf{main}\rangle}$$

where $\rho_r = [\langle \mathsf{defer}\ [\delta]\rangle_0\ \langle \mathsf{defer}\ [\,]\rangle_1 \ldots \langle \mathsf{defer}\ [\,]\rangle_n]$
$\delta = (\iota s, \rho s, \mathsf{main})$

apply satisfied (apply-1):

$$\frac{\langle\langle \mathsf{clsr}\ (1).\ (\underline{n}, \iota s_c, \iota s_d), \rho s'\rangle \quad \mathsf{apply} : \iota s \quad \rho' : \rho s \quad \omega s \quad \sigma \quad \mathsf{apply\text{-}1}\rangle}{\langle \rho_r \quad \iota s_c \quad \rho_r : \rho' : \rho s' \quad \omega s \quad \sigma \quad \mathsf{apply\text{-}0}\rangle}$$

where $\rho_r = [\langle \mathsf{defer}\ [\delta]\rangle_0\ \langle \mathsf{defer}\ [\,]\rangle_1 \ldots \langle \mathsf{defer}\ [\,]\rangle_n]$
$\delta = (\iota s, \rho s, \mathsf{apply\text{-}0})$

apply satisfied (apply-0):

$$\frac{\langle\langle \mathsf{clsr}\ (1).\ (\underline{n}, \iota s_c, \iota s_d), \rho s'\rangle \quad \mathsf{apply} : \iota s \quad \rho s \quad \omega s \quad \sigma \quad \mathsf{apply\text{-}0}\rangle}{\langle _ \quad [\mathsf{sched}] \quad \rho s \quad \omega s +\!\!+ [\omega] \quad \sigma \quad \mathsf{apply\text{-}0}\rangle}$$

where $\omega = (\langle \mathsf{clsr}\ (1).\ (\underline{n}, \iota s_c, \iota s_d), \rho s'\rangle, \mathsf{apply} : \iota s, \rho s, \mathsf{apply\text{-}1})$

Full arity application simply schedules the direct entry point of the closure. It also saves the return continuation into the first slot of the flat argument environment which would enable the callee to resume the caller at the end of the functional call.

fapply(main):

$$\frac{\langle\langle \mathsf{clsr}\ (\underline{n}).\ (\underline{n}, \iota s_c, \iota s_d), \rho s'\rangle \quad \mathsf{fapply} : \iota s \quad \rho' : \rho s \quad \omega s \quad \sigma \quad \mathsf{main}\rangle}{\langle _ \quad \iota s_d \quad \rho'[\langle \mathsf{defer}\ [\delta]\rangle_0 \ldots] : \rho s' \quad \omega s \quad \sigma \quad \mathsf{main}\rangle}$$

where $\delta = (\iota s, \rho s, \mathsf{main})$

fapply(apply-1):

$$\frac{\langle\langle \mathsf{clsr}\ (\underline{n}).\ (\underline{n}, \iota s_c, \iota s_d), \rho s'\rangle \quad \mathsf{fapply} : \iota s \quad \rho' : \rho s \quad \omega s \quad \sigma \quad \mathsf{apply\text{-}1}\rangle}{\langle _ \quad \iota s_d \quad \rho'[\langle \mathsf{defer}\ [\delta]\rangle_0 \ldots] : \rho s' \quad \omega s \quad \sigma \quad \mathsf{apply\text{-}0}\rangle}$$

where $\delta = (\iota s, \rho s, \mathsf{apply\text{-}0})$

fapply(apply-0):

$$\frac{\langle\langle \mathsf{clsr}\ (\underline{n}).\ (\underline{n}, \iota s_c, \iota s_d), \rho s'\rangle \quad \mathsf{fapply} : \iota s \quad \rho s \quad \omega s \quad \sigma \quad \mathsf{apply\text{-}0}\rangle}{\langle _ \quad [\mathsf{sched}] \quad \rho s \quad \omega s +\!\!+ [\omega] \quad \sigma \quad \mathsf{apply\text{-}0}\rangle}$$

where $\omega = (\langle \mathsf{clsr}\ (\underline{n}).\ (\underline{n}, \iota s_c, \iota s_d), \rho s'\rangle, \mathsf{fapply} : \iota s, \rho s, \mathsf{apply\text{-}1})$

Resource Management

The **pushwork** instruction creates a task with the given instruction thread and the environment chain along with the current contents of the accumulator and the **apply-1** flag.

pushwork:

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

where $\omega = (\nu, \iota s', \rho s', \mathsf{apply\text{-}1})$

The **allocletenv** and **alloclamenv** instructions allocate environments for blocks and functions respectively. The block environment slots are initialized to be **empty** and would be filled when the corresponding **enter** instruction is executed. The function environment slots are initialized to be in the **defer** state since separate tasks are spawned to fill them.

allocletenv:

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

where $\rho = [\langle \mathsf{empty}\rangle_0 \ldots \langle \mathsf{empty}\rangle_{n-1}]$

alloclamenv:

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

where $\rho = [\langle \mathsf{defer}\ [\,]\rangle_0 \ldots \langle \mathsf{defer}\ [\,]\rangle_{n-1}]$

6 Conclusions

In this paper we have presented a framework for compiling and interpreting non-strict, implicitly parallel languages using a multithreaded execution model on sequential or parallel targets. We started with a kernel language and a multithreaded abstract machine specification. Then we defined a mixed evaluation interpreter for the kernel language that embodies the notion of multithreading at an abstract level, separates the environment and the heap storage, and performs mostly demand-driven evaluation of multiple threads of control. Finally, we defined a compiler for the mixed evaluation strategy that converts the kernel language into a fixed set of threads taking position on environment allocation and data representations. Both the interpreter and the compiler are defined at a sufficiently high-level of abstraction so as to allow ample room for experiment with various compilation and scheduling strategies.

This work grew out of the need to reorganize the compilation strategy for the Id/pH compiler from a dataflow graph based approach to a kernel language based approach, both from the perspective of exploring more efficient execution strategies for non-strict programs on a multithreaded substrate and from the perspective of clearly specifying the compilation and execution rules in a formally defined abstract machine framework. The current scheme of mixing lazy and eager evaluation strategies was inspired by our earlier work on defining the semantics of barriers for non-strict, implicitly parallel languages [1].

7 Acknowledgments

The author would like to thank Arvind, Lennart Augustsson, Satyan Coorg, Joanna Kulik, Jan-Willem Maessen, and Joseph Stoy for several insightful discussions and suggestions.

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] Shail Aditya, Arvind, and Joseph E. Stoy. Semantics of Barriers in a Non-Strict, Implicitly-Parallel Language. CSG Memo 367-1, MIT Laboratory for Computer Science, Cambridge, MA 02139, April 1995. To appear in Proc. *Functional Programming Languages and Computer Architecture*, La Jolla, CA, June 1995.

[2] Zena M. Ariola and Arvind. Compilation of Id. In *Proceedings of the fourth Workshop on Languages and Compilers for Parallel Computing, Santa Clara, California*, August 1991. Also available as CSG Memo 341, 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 and Rishiyur S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990. An earlier version appeared in *Proceedings of the PARLE Conference, Eindhoven, The Netherlands*, Springer-Verlag LNCS Volume 259, June 15-19, 1987.

[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, 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] Satyan R. Coorg. Partitioning Non-strict Languages for Multi-threaded Code Generation. Master's thesis, MIT Laboratory for Computer Science, Cambridge, MA 02139, May 1994.

[8] Goldstein, Seth Copen and Schauser, Klaus Eric and Culler, David. Lazy threads, stacklets, and synchronization. In *Proceedings of POOMA '94*, 1994.

[9] Steven K. Heller. *Efficient Lazy Data-Structures on a Dataflow Machine*. PhD thesis, Laboratory for Computer Science, MIT, Cambridge, MA 02139, February 1989. Available as Technical Report MIT/LCS/TR-438.

[10] 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.

[11] Rishiyur S. Nikhil. The Parallel Programming Language Id and its Compilation for Parallel Machines. CSG Memo 313, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, July 1990. Presented at Workshop on Massive Parallelism, Amalfi, Italy, October 1989.

[12] 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.

[13] 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.

[14] 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.

[15] Kenneth R. Traub, David E. Culler, and Klaus E. Schauser. Global Analysis for Partitioning Non-Strict Programs into Sequential Threads. In *Proc. ACM Conf. on LISP and Functional Programming, San Francisco, CA*, June 1992. Also available as Motorola Technical Report MCRC-TR-26.