LABORATORY FOR COMPUTER SCIENCE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

A Multithreaded Substrate and Compilation Model for the Implicitly Parallel Language pH

Computation Structures Group Memo 382-1
Revised November 11, 1997

Arvind   Alejandro Caro   Jan-Willem Maessen
MIT Laboratory for Computer Science

Shail Aditya
HP Laboratories

This research was supported by ARPA Contract Number: DABT63-95-C-0150

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

A Multithreaded Substrate and Compilation Model for the Implicitly Parallel Language pH

Arvind Alejandro Caro Jan-Willem Maessen Shail Aditya
MIT Laboratory for Computer Science* HP Laboratories†

November 2, 1997

Abstract

We describe the compilation of the non-strict, implicitly parallel language pH to symmetric multiprocessors (SMPs) in several steps. We introduce the λ_S calculus as a robust foundation for the semantics of pH. Next, we define a shared-memory threaded abstract machine (SMT) that captures the essence of our compilation target, a modern SMP. Finally, we describe a complete syntax directed translation of λ_S to SMT instructions. The paper makes three important contributions: it is the first implementation of pH based on *direct* semantics of barriers; second, in contrast to earlier work, the multithreaded code generated uses *suspensive* threads; and third, the compilation rules generate code from λ_S source code directly, without resorting to intermediate dataflow-style graphs.

1 Introduction

This paper describes the compilation of the pH language for symmetric multi-processors (SMPs). pH is a parallel dialect of the functional language Haskell. It has been designed to support general purpose parallel programming, including unstructured problems. Because the language is both implictly parallel and non-strict, pH programs are highly concurrent and eminently portable across machine architectures.

The philosophy underlying the design of pH is simple: to have broad impact, parallel programming must be easy. Let the programmer concentrate on the algorithms; let the compiler worry about efficient implementation. Today, this viewpoint does not find broad acceptance in the parallel computing community, a community that is obsessed, perhaps justifiably, with extracting every last drop of performance from outrageously expensive machines. Yet, as the cost of parallel machines drops, programmability is bound to become *the* overriding concern. We believe the development of powerful, relatively affordable SMPs represents the beginning of the end for the high-priesthood of parallel programming. Parallel languages like pH, HPF, and SISAL will make parallel machines accessible to the programming masses.

The compilation scheme for pH is based on fine-grain *multithreading*. These programs consist of a large number of threads which compute, communicate, and synchronize with minimal overhead. Multithreading is used for two reasons in the implementation of languages like pH. First, it is used to exploit the concurrency available in a program. Second, it is used to implement the correct semantics of the language since the order in which certain computations execute is determined at run-time, not at compile-time. Recent implementations of fine-grain multithreading rely on "user-level" threads rather than on operating system "light-weight" thread packages. O/S packages, like POSIX threads, carry too much baggage for efficient execution of small threads.

1.1 Related Work

The history of pH, and its predecessor Id, has been closely coupled with the evolution of dataflow architectures. The dataflow execution model offers vast instruction-level parallelism but requires cheap synchronization. Custom-built dataflow machines like Monsoon [12] proved to be ideal vehicles to execute languages like Id and pH. Unfortunately, these machines were doomed by the onslaught of commercial processor technology, despite the disadvantages of the standard von Neumann architecture in a parallel setting.

*`arvind,acaro,earwig@abp.lcs.mit.edu`
†`aditya@hplsag.hpl.hp.com`

Significant work has been done to implement languages like pH on standard processors. Traub's seminal work on partitioning [11] partially inspired the TAM work at Berkeley [6] and the pHluid work by Nikhil [7]. Still, both these systems convert programs into dataflow graphs with control flow information encoded as explicit "signals". Then, the graphs are divided into partitions, each of which is turned into a sequential thread. This approach to code generation suffers from several weaknesses due to its dataflow foundations. For example, the implementation of control flow makes poor use of the efficient mechanisms offered by von Neumann processors; also, execution efficiency on pipelined processors is hampered by short instruction sequences.

In contrast to earlier approaches, the compilation scheme described in this paper is a complete syntax directed translation of pH (in its kernel form) directly to multi-threaded code. The compilation process avoids dataflow graphs entirely and generates code that takes advantage of the strengths of the underlying von Neumann architecture.

We present the compilation of pH to SMPs in three parts. First, we introduce the λ_S calculus [4] as a robust foundation for the semantics of pH. Then, we define a shared-memory threaded abstract machine (SMT) that models the important properties of our real compilation target, an SMP. Finally, we describe the rules for translating the entire λ_S calculus into abstract machine instructions. This general approach is based on two previous research efforts: early work by Ariola and Arvind on compilation of Id to P-TAC [3]; and recent work by Aditya [1] utilizing a slightly different source language and target machine than the ones presented here.

1.2 Contributions of this Paper

We make three important contributions in this paper. We present the first implementation of pH based on the precise, *direct* semantics of barriers as captured by the λ_S calculus. Previous efforts, including our own Id compiler for Monsoon, used either poorly defined, ad-hoc semantics or precise semantics based on program transformations [2]. The new semantics require important changes in the implementation of free variables, bound variables, and closures. These changes would be quite difficult to retro-fit into existing Id/pH compilers.

Second, our compilation scheme makes use of *suspensive* threads. Suspensive threads make optimistic assumptions about synchronization. This type of thread will begin to execute assuming all the values it requires have been computed; it will only suspend if a required value is missing. In contrast, the non-suspensive threads used by TAM and pHluid make pessimistic assumptions about synchronization; a thread will begin to execute only when all the values it requires have been computed. We believe suspensive threads yield better code than non-suspensive threads for two reasons: overhead due to thread suspension is reduced and thread lengths increase.

Thread suspension overhead has two components: overhead per suspension and number of suspensions per program execution. Suspensive threads are likely to have higher cost per suspension than non-suspensive threads since state is saved between suspensions. Non-suspensive threads simply halt and go on to the next thread. But we believe this increased cost will be dramatically offset by a large *decrease* in the total number of suspensions in a program. In addition, suspensive threads will consist of longer code sequences that will make better use of registers than short, non-suspensive threads.

Our final contribution is a compilation scheme that generates threaded code directly from pH (λ_S), without resorting to dataflow graphs. From a theoretical point of view, we believe it is important to present pH as a language that stands on its own, apart from its dataflow roots. From an implementation point of view, we avoid many of the inefficiencies introduced into threaded code by the dataflow model of computation, and we believe we expose more opportunities for generating better code for SMPs.

1.3 Overview of the Paper

In Section 2, we introduce the λ_S calculus, including syntax, reduction rules, and evaluation strategy. The λ_S calculus provides a high-level view of pH program execution. Section 3 presents a more concrete view of the execution environment for compiled λ_S programs. Section 4 presents the SMT abstract machine and its instruction set. SMT is completely independent of the pH language and, indeed, could be used to implement any number of different languages. Finally, in Section 5, we bring together λ_S and SMT by providing

$$
\begin{array}{lcll}
E & ::= & x & :: \text{identifier} \\
 & | & \lambda x.E & :: \text{abstraction} \\
 & | & x\ y & :: \text{application} \\
 & | & \{S \text{ in } x\} & :: \text{letrec block} \\
 & | & \texttt{Case}(x, E_1 \ldots E_k) & :: \text{integer conditional} \\
 & | & PF^k(x_1 \ldots x_k) & :: k\text{-ary primitive} \\
 & | & Number | Boolean | \ldots | Location & :: \text{constant} \\
 & | & CN^k(x_1 \ldots x_k) & :: \text{constructor application} \\
 & | & \texttt{allocate}(x) & :: \text{mutable allocation} \\
 & | & \top & :: \text{error value (top)} \\
S & ::= & \epsilon & :: \text{empty binding} \\
 & | & x = E & :: \text{identifier binding} \\
 & | & S\ ;\ S & :: \text{parallel composition} \\
 & | & S \text{---} S & :: \text{barrier, sequential composition} \\
 & | & \texttt{iStore}(x, y) & :: \text{I-structure store} \\
 & | & \texttt{mStore}(x, y) & :: \text{M-structure store} \\
 & | & \top_S & :: \text{Error binding} \\
PF^1 & ::= & \texttt{not}|\texttt{iFetch}|\texttt{mFetch}|\texttt{Prj}^1|\texttt{Prj}^2| \ldots & :: \text{unary primitives} \\
PF^2 & ::= & + | - | * | \ldots & :: \text{binary primitives} \\
CN^2 & ::= & Cons | Pair | \ldots & :: \text{Data constructors}
\end{array}
$$

Note that all bound variables for a block are pairwise distinct. Also, the following identities apply:

$$
\begin{array}{lcll}
S_1\ ;\ S_2 & \equiv & S_2\ ;\ S_1 & :: \text{commutativity of ;} \\
S_1\ ;\ (S_2\ ;\ S_3) & \equiv & (S_1\ ;\ S_2)\ ;\ S_3 & :: \text{associativity of ;}
\end{array}
$$

Figure 1: Syntax of the Kernel λ_S Calculus

a complete syntax directed translation of the entire λ_S language into SMT instructions. The translation rules are designed to be simple; they are not intended to produce top-notch code. We also discuss several optimizations that can improve the quality of the code generated. Finally, we discuss the status of our implementation and directions for future work.

2 The λ_S Calculus

The λ_S calculus is a variant of the call-by-need λ calculus. The front-end of the compiler converts programs from the full pH language to the much simpler λ_S representation for use in subsequent phases. The calculus captures the semantics of pH precisely, including its unique features. λ_S uses recursive let bindings to express the sharing of values in a program. It also allows reductions to occur in parallel on different parts of a term. It includes I-structure and M-structure operations that perform necessary synchronization within the term without the need for a separate store. Finally, it provides a direct semantics for *barriers* in binding groups.

The syntax of the kernel λ_S calculus is presented in Figure 1. The structure of values is presented in Figure 2. The kernel syntax imposes several restrictions on terms, such as the use of identifiers instead of expressions in applications, primitive functions, and constructors. These restrictions do not curtail the power of the λ_S calculus; rather they make it a clean language for use by the compilation rules presented in Section 5. The following sections highlight several aspects of the λ_S calculus important to this paper. For a more thorough discussion of λ_S, refer to [4].

V	$::=$	$\lambda x.E$	:: Value: Lambda expression
	$\mid$	$Number \mid Boolean \mid \ldots \mid Location$	:: Constant
	$\mid$	$CN^k(x_1 \ldots x_k)$	:: Simple constructor application
VS	$::=$	$x = V$	:: Terminated binding
	$\mid$	$\mathtt{iStore}(V, V)$	:: Completed I-store
	$\mid$	$\mathtt{mStore}(V, V)$	:: Completed M-store
	$\mid$	$VS\ ;\ VS$	
P	$::=$	$V \mid \{\ VS \text{ in } V\ \}$	:: A terminated program

Figure 2: Values and Terminated Programs

2.1 λ_S vs. Lazy Functional Languages

Though non-strict, the λ_S calculus is quite different from the formal systems underlying lazy functional languages like Haskell [8, 9]. The first major difference between λ_S and lazy languages is that λ_S is evaluated *eagerly*. This means that λ_S will evaluate some expressions even if they are not necessary to compute the result of a computation. Lazy languages, in contrast, evaluate only those expressions that are absolutely necessary to produce a result.

In addition, the λ_S calculus includes *barriers* (indicated by '—'), a construct used to sequentialize the execution of statements. In the following example,

$$
\begin{array}{l}
a = \ldots; \\
b = \bot; \\
(x = \mathtt{Case}(a, 7, b); \\
\overline{} \\
S\ \ )
\end{array}
$$

the statement x above the barrier is considered to be in the *pre-region* of the barrier. The statement S is considered to be the *post-region* of the barrier. The barrier guarantees that everything in the pre-region will be computed before the post-region is executed. This implies that, recursively, the branch selected by the `Case` expression must terminate before the barrier *discharges*. In this particular example, if the value of a is 1 then the barrier will discharge; if the value of a is 2, the barrier will not discharge because the evaluation of b does not terminate.

Barriers are intimately tied to non-strict semantics and eager evaluation. In the previous example, x will be evaluated even if its value is not needed anywhere in the program. Lazy languages are not eager and have a highly sequential model of evaluation, so the guarantee of termination offered by barriers is not useful in such a setting.

The third important difference between λ_S and lazy functional languages is that λ_S is not purely functional. It incorporates side-effects through two kinds of synchronizing memory: I-structures and M-structures. An I-structure location can be written at most once and can be read many times. If any reads occur before the single write, the reads are said to "defer" until the write occurs. The computations that performed the reads will wait until the value of the location is computed and stored in the location.

An M-structure location can be read and written multiple times, but with mutual exclusion. Only one computation can read the value of an M-structure location at a time. Once the value is read successfully, subsequent readers are deferred until another value is written back. Then, one of the deferred readers will receive the value written back while the rest remain deferred.

It is always considered an error to write to a location that already contains a value. This "double-write" error immediately causes the program to evaluate to $\top$, the *inconsistent* state.

2.2 Reduction Rules

The key subset of the reduction rules of λ_S is presented in Figure 3. In this section, we discuss several important aspects of these.

β :

$$(\lambda x.e)\ a \quad \rightarrow \quad \{x' = a \text{ in } e[x'/x]\}$$

Instantiation:

$$\{x = v \ ; S \text{ in } y\} \quad \rightarrow \quad \{x = v[v/x] \ ; S[v/x] \text{ in } y[v/x]\}$$

Block Flattening:

$$x = \{S \text{ in } a\} \quad \rightarrow \quad x = a' \ ; S'$$

where a' and S' are α-renamed versions of a and S to avoid name conflicts

δ:

$$PF^k(v_1 \ldots v_k) \quad \rightarrow \quad \ldots$$

Conditional:

$$\texttt{Case}(j, e_1 \ldots e_k) \quad \rightarrow \quad e_j \qquad 1 \leq j \leq k$$

Projection:

$$\texttt{Prj}^j(CN^k(x_1 \ldots x_k)) \quad \rightarrow \quad x_j$$

Structure Allocation:

$$\texttt{allocate}(n) \quad \rightarrow \quad \ell$$

where ℓ represents the first of the n allocated locations

Barrier Discharge:

$$(x = v \ ; S_1) \text{—} S_2 \quad \rightarrow \quad x = v \ ; (S_1 \text{—} S_2)$$
$$(\texttt{iStore}(v, v) \ ; S_1) \text{—} S_2 \quad \rightarrow \quad \texttt{iStore}(v, v) \ ; (S_1 \text{—} S_2)$$
$$(\texttt{mStore}(v, v) \ ; S_1) \text{—} S_2 \quad \rightarrow \quad \texttt{mStore}(v, v) \ ; (S_1 \text{—} S_2)$$
$$(\epsilon \text{—} S) \quad \rightarrow \quad S$$

Fetch:

$$\texttt{iStore}(\ell, v) \ ; x = \texttt{iFetch}(\ell) \quad \rightarrow \quad \texttt{iStore}(\ell, v) \ ; x = v$$
$$\texttt{mStore}(\ell, v) \ ; x = \texttt{mFetch}(\ell) \quad \rightarrow \quad x = v$$
$$((x = \texttt{iFetch}(\ell) \ ; S_1) \text{—} S_2) \quad \rightarrow \quad (t = \texttt{iFetch}(\ell) \ ; ((x = t \ ; S_1) \text{—} S_2))$$
$$((x = \texttt{mFetch}(\ell) \ ; S_1) \text{—} S_2) \quad \rightarrow \quad (t = \texttt{mFetch}(\ell) \ ; ((x = t \ ; S_1) \text{—} S_2))$$

Store Error, generation and propagation:

$$\texttt{iStore}(\ell, v_1) \ ; \texttt{iStore}(\ell, v_2) \quad \rightarrow \quad \top_S$$
$$\texttt{mStore}(\ell, v_1) \ ; \texttt{mStore}(\ell, v_2) \quad \rightarrow \quad \top_S$$
$$\top_S \ ; S \quad \rightarrow \quad \top_S$$
$$\top_S \text{—} S \quad \rightarrow \quad \top_S$$
$$x = \top \quad \rightarrow \quad \top_S$$
$$\{\top_S \text{ in } x\} \quad \rightarrow \quad \top$$

Notation: v,v_n is a value as in Figure 2; ℓ is a *Location*; a, a', x, x', x_n are identifiers; e, e_n are expressions; n is an expression which evaluates to a *Number*.

Figure 3: Reduction Rules for the Kernel λ_S Calculus

The β Rule and Instantiation

The β rule names the argument and substitutes that name in the body of the λ-abstraction rather than substituting the argument directly. This preserves the non-strict semantics of function calls in λ_S because it allows function calls to proceed even if arguments have not been computed.

The Instantiation rule only allows variables that have been bound to values to be instantianted elsewhere in the program. In other words, a reference to a variable will never get replaced by an expression that computes the variable; it will only get replaced by the value of the variable once that value has been computed.

The restriction imposed by the Instantiation rule has important implications for efficiency and correctness. From the point of view of efficiency, it allows the results of computations to be shared and avoids redundant computation. From the point of view of correctness, it ensures that expressions involving side-effects are computed only once.

Non-strictness of Constructors

Constructors are non-strict in λ_S. This means that a constructor can be created even before any of its components has been evaluated. This behavior is captured by the reduction rule for Projections. This rule states that a projection returns an identifier as opposed to a value. In other words, projections do not force the evaluation of a constructor field.

Barriers

Conceptually, barriers guarantee the termination of the pre-region before the execution of the post-region. The reduction rules for barriers accomplish this incrementally by moving bindings out of the pre-region as the variables become bound to values. Gradually, the number of bindings in the pre-region decreases until one of two things happens: no bindings remain, in which case the barrier discharges and the bindings the in post-region are evaluated; or at least one binding that does not evaluate to a value remains in the pre-region, in which case the barrier does not discharge.

I-structures and M-structures

The `allocate` primitive function returns the first location of a fresh block of memory. The reduction rule for `iFetch` tries to match each `iFetch` with its corresponding `iStore`. Once the match is found, the rule replaces the `iFetch` instruction with the value of the `iStore`. However, it keeps the `iStore` instruction around to match with other corresponding `iFetch` instructions. This models the write-once, read-many semantics of I-structures.

The rule for `mFetch` is slightly different. Again, matching `mFetch` and `mStore` instructions are paired-up. The `mFetch` instruction is replaced by the value in the corresponding `mStore`. But the rule goes one step further: unlike the `iFetch` rule, this rule deletes the `mStore` instruction. This guarantees that only one `mFetch` in the program will receive the value of a particular `mStore` instruction. This behavior is required by the mutual-exclusion semantics of M-structures.

The final two Fetch rules deal with `iFetch` or `mFetch` instructions in barrier pre-regions. The rules are structurally identical so we only discuss the `iFetch` case. Suppose an `iFetch` appears in a barrier pre-region, and its corresponding `iStore` appears outside the barrier pre-region. How should the two instructions be paired-up? In general, the `iFetch` statement cannot be pulled out directly without changing the termination properties of the pre-region.

One option is to move the `iStore` inside the pre-region. However, this approach is problematic for two reasons. First, it is dangerous to pull an external computation into a pre-region because the termination properties of the pre-region may be affected. A second complication in the case of `iStore` instructions is that the effects of the `iStore` would be hidden from other `iFetch` instructions outside the pre-region.

The rule given, on the other hand, introduces an additional identifier and pulls the `iFetch` out of the pre-region. This is enough to maintain the proper termination behavior of the pre-region while allowing the Fetch rules to match up `iStore` and `iFetch` instructions.

Store Errors

A store error occurs when an I-structure location is written twice or when an M-structure location is written twice without an intervening `mFetch` instruction. Store errors are regarded as an inconsistency in the program and cause evaluation to stop abruptly. The rules given in Figure 3 ruthlessly propagate store errors through the program up to the top-level expression. For theoretical reasons [4], store errors are regarded as $\top$ rather than $\perp$.

2.3 Normalizing Strategies

A normalizing strategy for applying reduction rules must make two guarantees. First, if there is any way a program can terminate (see Figure 2), then the strategy must find it. Second, the reduction must be accomplished in a finite number of steps. There are many normalizing strategies for λ_S. An important class of these avoids redexes inside `Case` expressions, λ-abstractions, and barrier post-regions, even though applying these reductions is perfectly legal in the calculus.

All normalizing strategies for λ_S must reduce redexes in parallel due to the presence of I-structures and M-structures in the language. Suppose you were implementing a sequential normalizing strategy for λ_S. At some point, you must choose one of many available redexes to evaluate. Unfortunately, if a redex that does not terminate is chosen, the sequential strategy will not produce a result *even though a different choice of redex might have led to termination.* A parallel strategy reduces many redexes at a time, so it cannot "go off the deep end" by making a bad choice of redex.

The most straightforward way to implement a parallel normalizing strategy is to evaluate all right-hand-sides of bindings in parallel, subject to the restrictions stated earlier to avoid redexes in `Case` expressions, λ-abstractions, and post-regions. This simple-minded approach is a double-edged sword. It generates lots of parallelism while maintaining the normalizing property of the strategy, but some of the parallelism it generates is quite useless. Consider the following example:

$$\begin{aligned} &x = f\ a; \\ &y = g\ b; \\ &z = x + y; \end{aligned}$$

Any attempt to evaluate z before x and y have been evaluated is wasted effort. On a sequential machine, the best way to execute this block of code would be to evaluate the bindings in the order given: x, y, and z. Compiler analysis can reveal opportunities for this type of "sequentialization" without destroying the normalizing property of the evaluation strategy. In the compilation rules of Section 5, we will indicate sequentialization by replacing ';' with '$\sim$'. This type of analysis is similar to partitioning [11, 10, 5].

2.4 Pre-Compilation

The source language used in the syntax directed translation presented in Section 5 is the kernel λ_S calculus with two small changes. First, all barriers are named with a unique identifier, so that '—' becomes '–b–', where b is the unique identifier for the barrier. This change allows the compiler to refer to a barrier as if it were a local "variable" and allows the allocation of resources to implement the barrier.

The second change made to the kernel language is that all non-essential blocks—those not appearing inside `Case` expressions, λ-abstractions, and post-regions—are flattened into a single block.

3 Execution Environment

Our ultimate goal in this paper is to present a compiler that translates λ_S into machine code for a low-level parallel abstract machine. This section provides an overview of the execution environment implemented in the compiled code. We make the following assumptions about the abstract machine: the existence of a single global memory shared by all processors; that memory locations contain indicators which tell whether or not a value is present in the location (full vs. defer); and the existence of a set of registers per processor. The detailed description of the machine is left to Section 4.

Figure 4 illustrates the structure of the execution environment embodied in the code produced by our λ_S compiler. This execution environment is the result of numerous design decisions, and we describe the most important of these in the following sections.

3.1 Locations for Variables

Identifiers are not merely placeholders for values. Rather, in λ_S, identifiers represent the *names* of values that may or may not be computed during the execution of the program. This distinction between *values* and what we intuitively refer to as *proxies for values* also appears in lazy and lenient languages. Lazy languages

$$
\begin{array}{l}
\{\quad \ldots \\
\quad a = Pair(5,6); \\
\quad f = \lambda x.\{p = Prj^2(a); \\
\qquad\qquad\quad q = g\ p; \\
\qquad\qquad\quad \text{—}b\text{—} \\
\qquad\qquad\quad \mathtt{in}\ q\}; \\
\\
\quad g = \lambda y.\ y * y; \\
\\
\quad z = f\ a; \\
\quad \mathtt{in}\ z \\
\}
\end{array}
$$

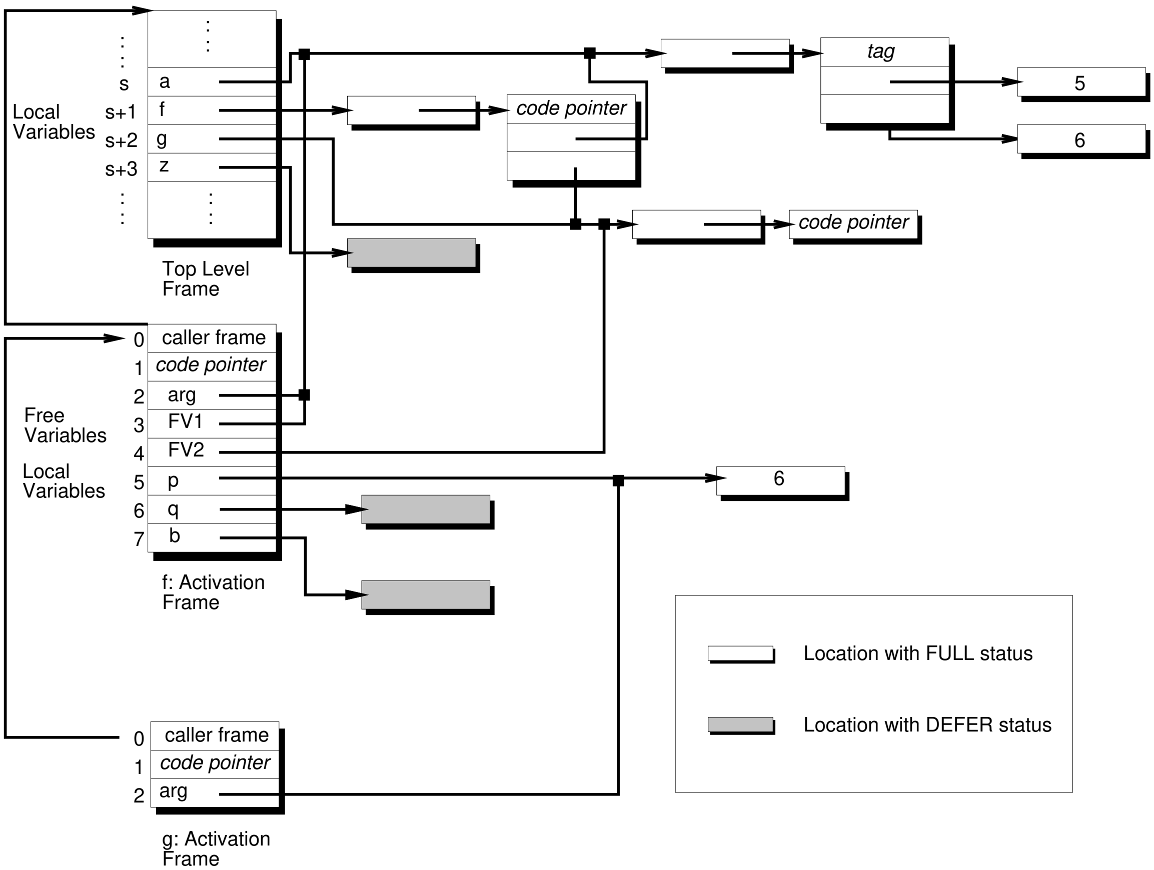

Figure 4: Execution Environment of Compiled Code. The figure shows the state of the machine before the execution of the $*$ in function g.

like Haskell make a distinction between two different kinds of objects: values and unevaluated expressions (or "thunks"). Values are required only by strict operators, while thunks can supplant values anywhere else in a program. A thunk encapsulates all the computation required to produce a value.

In contrast, lenient languages, like Id, have evolved from dataflow computation models. Dataflow has no concept of an "unevaluated" expression; all dataflow operators work with values. Lenient languages have been extended with I-structures and M-structures, and these languages do distinguish between values that have been computed and those that remain to be computed. A computation that requests a value that has not been computed simply waits until that value is produced elsewhere in the program. No thunk is used to represent the value; rather, presence bits are used to *synchronize* between consumers and producers of values.

Correct implementation of the semantics of λ_S requires a mechanism by which programs can refer to values without ever computing them. Consider the problem of creating a closure. A closure contains a vector of free variables that captures those pieces of the lexical environment that *may* be used by the code of the closure. What object should be stored in the closure for a free variable? We might consider storing the value of the free variable, but if the free variable is not actually used in the closure at run-time, then we will have computed its value prematurely. In other words, closures are non-strict with respect to free variables.

As an alternative, we might consider using a thunk to represent the value of the free variable. But λ_S is a language that is evaluated eagerly; lazy thunks have no place in the semantics of such a language. So what are we to do? Not surprisingly, λ_S dictates that we store the *name* of the free variable in the closure. A concrete implementation of this idea involves the use of pointers. Suppose that every variable in a program is assigned a location in memory; a pointer to this location is a representation of the name of the variable. It uniquely identifies the variable, yet it is independent of whether or not the variable has been computed.

This is exactly the behavior we require, and consequently, the locations of values, rather than actual values, are used frequently in the compilation rules to implement the proper semantics of λ_S. In particular, locations are used to guarantee that barriers, conditionals, closures, and non-strict function calls behave as expected.

3.2 Activation Frames

Activation frames provide each function invocation with private space for storing local variables. In our compiling scheme, activation frames are allocated in the shared heap as "flat" blocks of memory. This structure was chosen for its simplicity rather than for its efficiency. The first two slots of a frame contain a pointer to the activation frame of the caller and the program counter of the caller (*i.e.* the return address). These two values are used to restore the state of the machine and to return to the caller when the function completes. The third slot in the frame contains the location in memory of (a pointer to) the argument of the function.

As explained in the previous section, the use of a location for the argument allows for the correct implementation of non-strict function calls. A function can execute and return a result even if the value of its argument has not been computed. The only requirement is that the storage for the argument be allocated by the time the function is called; otherwise, there will be no meaningful value for the location of the argument. This requirement is easy to satisfy in the compiled code.

The slots for the free variable locations follow the argument in the activation frame. As before, these slots contain the locations in the heap assigned to the values of the free variables. The rest of the slots in the activation frame are dedicated to storing the locations of the local variables of the function, including barriers.

The structure of the activation frame for each function is determined statically at compile-time. The mapping between source identifier names and slots in the activation frame is maintained by a compile-time data structure α. There is a single α per lambda abstraction in the source program; this corresponds to the fact that at run-time the structure (not the content) of the activation frame of a particular function is always the same.

3.3 Constructors and Closures

Figure 4 also illustrates the structure of the two simple data structures used by λ_S: constructors and closures. Both of these are represented as flat chunks of memory. The first slot of a constructor contains the *tag*, an integer identifying the constructor. The slots representing the fields of the constructor follow the tag. Each of these contains the location where the value of the field is stored. Constructors, like functions, are non-strict, and locations allow us to allocate a constructor before the values of any of its fields have been computed.

The structure of a closure is equally straightforward. The first slot contains (a pointer to) the code of the closure while subsequent slots contain the locations of the free variables of the closure. Closures are non-strict with respect to the values of their free variables; as with frames and constructors, the use of locations enables a straightforward implementation of these semantics.

3.4 Barriers

The presentation of the λ_S calculus described the semantics of barriers in terms of "lifting" out of the pre-region bindings that have terminated. In contrast, the compiled code implements barriers by using a counter for each barrier in the program. The counter is used to keep track of the number of active bindings in any pre-region. As each binding terminates, the associated counter is decremented. Once a counter reaches zero, the barrier is effectively discharged.

It is important to notice that barriers can be nested, so that different parts of the program belong to different barrier pre-regions. Any particular statement, however, is associated with only one barrier. As the program executes, the identity of the "current" barrier changes. λ_S programs are compiled so that register rB always contains the location of the current barrier; the code generated for a statement in the pre-region uses register rB to locate the current barrier and assert that the statement has terminated.

Our compilation rules change the "current" barrier (the value in rB) when a new pre-region is encountered or when an "old" barrier is restored after the current pre-region is discharged. This is one of the many possible implementations of barriers. An equally valid alternative is to establish a new barrier every time a new *control region* is entered. Control regions are defined by `Case` expressions, λ-abstractions, and barriers pre-regions. Such a scheme would not require the compiled code to maintain the value of rB dynamically. However, it would require a much larger number of barrier counters than our current scheme.

4 SMT: A Shared-Memory Threaded Abstract Machine

4.1 Overview

The structure of our shared-memory threaded abstract machine SMT is illustrated in Figure 5. The machine consists of a number of sequential processors connected to a global shared memory. Each processor has its own register set, but all processors share a common *work queue*. Tasks are executed from this shared queue in a manner that guarantees fair scheduling. We chose this machine organization because we wanted to closely model the characteristic features of a modern symmetric multiprocessor (SMP), our expected compilation target.

Each location in the SMT shared memory can contain either a *value* or a list of *deferred* computations. Locations are tagged with their status, which is initially defer and changes to full when a value is stored in the location. A value is either a constant, an allocated memory location (a pointer), or a *barrier* object. Barriers are explained below in more detail.

Each SMT processor contains a register file R with a finite number of registers r0,…,rn. Machine instructions operate on the contents of registers. There are also two special registers: rB and rF. Register rB contains the location of (a pointer to) the current barrier object. Register rF contains the location of the current activation frame.

SMT also contains a global work queue shared by all processors, where threads that are ready to execute are stored. The work queue is managed to guarantee fair scheduling. Idle processors extract work from the front of the queue while freshly enabled work is placed at the end of the queue. Threads in SMT are *suspensive*. By executing one of a small set of instructions (touch, touchbar, take), a thread can be made to wait for a value that has not been computed. When the value is finally computed, the waiting thread is again placed on the work queue. It is up to the compiler or the programmer to guarantee that all values requested will be computed eventually. Otherwise the machine will enter a deadlock state at some point in the computation.

A thread ω is described by a four-tuple:

$$(\iota s,\ \nu,\ \ell_B,\ \ell_F)$$

These values are used to initialize the state of a processor when the thread is scheduled to execute:

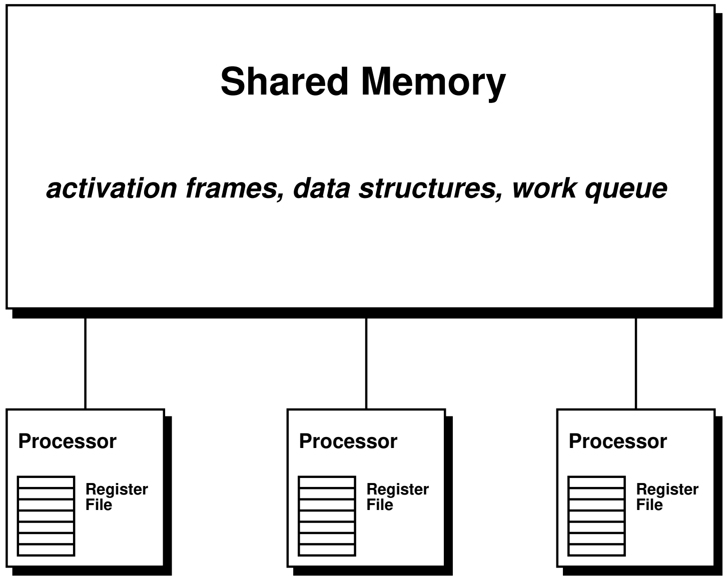

Figure 5: SMT: A Shared-Memory Threaded Abstract Machine

- ιs : represents a sequence (list) of instructions. As mentioned earlier, each individual processor of SMT is a sequential processor, and ιs provides a convenient abstraction for the tasks of instruction fetch and dispatch. A processor extracts the first instruction from ιs, executes it, extracts the next instruction, executes it, and so on until no instructions are left in the sequence.
- ν : a value used to initialize register r0. Register r0 is treated specially by the SMT instruction set: the value of the register is saved by the suspensive instructions and is restored when a thread is scheduled. Thus, the value of r0 is valid across suspension points. This behavior simplifies the compilation rules presented in Section 5.
- ℓ_B : a location used to initialize register rB. Every instruction in a thread is associated with a particular barrier in the program. When a thread is scheduled, it must establish the location of its barrier in register rB.
- ℓ_F : a location used to initialize register rF. Every thread is part of a function, and every function instance is assigned an activation frame for local storage. When a thread is scheduled, it must establish the location of its activation frame in register rF to have access to its local variables.

  Barrier objects are used to implement the barrier constructs of λ_S. These are three-tuples

$$(n,\ \ell,\ \omega)$$

consisting of the following fields:

- n : the barrier count. When the barrier is initialized, this counter is set to a value greater than zero. It is updated by the incbar and decbar instructions. When it reaches zero, the pre-region associated with the barrier is complete and the barrier *discharges*.
- ℓ : the value to be placed in register rB after the barrier discharges. This value identifies the barrier associated with the *post-region* of the current barrier. In essence, a dynamic link is established between barrier regions at run-time so that when a particular barrier *pre-region* terminates, its enclosing "parent" barrier can be located and re-established for the post-region.
- ω : a thread representing the post-region of the barrier. When the barrier discharges — when n reaches zero — ω is ready to execute.

ν	::	Value	::=	Constant | Location | Barrier
$\underline{c}$	::	Constant	::=	Number | Boolean | ...
ℓ	::	Location	=	An allocated location in memory (a pointer)
$\underline{\ell}$	::	Location	=	A location specified via an addressing mode
$\mathcal{B}$	::	Barrier	=	Number $\times$ Location $\times$ (Work $\mid \diamond$)
σ	::	Store	=	Location $\mapsto \{\langle \mathsf{full}\ \nu\rangle \mid \langle \mathsf{defer}\ \omega s\rangle \mid \diamond\}$
ι	::	Instruction		
ιs	::	Code	=	*List*(Instruction)
ω	::	Thread	=	Code $\times$ Value $\times$ Location $\times$ Location
ωs	::	Threads	=	*List*(Thread)
R	::	Register File		
r0,...,rn	::	Register	=	Specific registers in R
rB	::	Register	=	Special barrier register in R
rF	::	Register	=	Special activation frame register in R
ra,rb,rc	::	Register	=	Any register in R
(ra)	::	Value	=	Contents of register ra
$\diamond$	::	Empty		
_	::	"Don't care"		

Figure 6: Run-time Objects in SMT

4.2 SMT Instruction Set

This section presents the instruction set of the abstract machine. The semantics of each instruction are presented as a set of rules based on *before* and *after* machine states, as in:

$$\frac{\textit{before state}}{\textit{after state}}$$

The *before* state is a "pattern" that must match the current processor state in order for the instruction to execute. The *after* state specifies the changes made to the processor/machine state by the instruction. These states are specified using the notation in Figure 6. The rules have been designed so that at most one *before* state can match the state of the processor at any point in the execution of a program. It is possible, nevertheless, that no *before* state matches the current processor state. This condition arises from a dynamic error and forces the entire machine to halt.

Addressing Modes

A number of machine instructions require a location as one of their operands. We define several addressing modes that can be used to specify these locations. Note that the value for the displacement ($\underline{i}$) used in the indexed modes is always a compile-time constant:

Mode	Example	Meaning
Register	load(ra, rb))	ra $\leftarrow \nu$, where $\sigma[$ (rb) $\mapsto \langle \mathsf{full}\ \nu\rangle$]
Register Indexed	load(ra, rb[$\underline{i}$])	ra $\leftarrow \nu$, where $\sigma[$ (rb) + $\underline{i} \mapsto \langle \mathsf{full}\ \nu\rangle$]
Register Indirect	load(ra, @rb)	ra $\leftarrow \nu$, where $\sigma[$ (rb) $\mapsto \langle \mathsf{full}\ \ell\rangle,\ \ell \mapsto \langle \mathsf{full}\ \nu\rangle$]
Register Indexed Indirect	load(ra, @rb[$\underline{i}$])	ra $\leftarrow \nu$, where $\sigma[$ (rb) + $\underline{i} \mapsto \langle \mathsf{full}\ \ell\rangle,\ \ell \mapsto \langle \mathsf{full}\ \nu\rangle$]

These addressing modes apply to all instructions that are load-like, *i.e.* touch, take, etc. Addressing modes are also used in a similar way in store-like instructions such as istore, incbar, and so on.

Processor State

The state of a processor in SMT can be described as a four-tuple using the notation introduced earlier:

$$\langle \mathsf{R} \quad \iota s \quad \omega s \quad \sigma \rangle$$

where R is the local processor register file, ιs is the local instruction sequence, ωs is the global work queue, and σ is the global store. The Initial, Final, and Error states of the machine are described in the following table:

State	Processor	Register File	Code	Work Queue	Memory
Initial	initial	_	ιs_0	[ ]	{}
	all others	_	[ ]	[ ]	{}
Normal Halt	any	$(\mathsf{r0}) \neq$ *error*	halt:ιs	[ ]	σ
	all others	$(\mathsf{r0}) \neq$ *error*	schedule:ιs	[ ]	σ
Deadlock	all	_	schedule:ιs	[ ]	σ
Store Error	any	$(\mathsf{r0}) = \underline{\mathsf{storerr}}$	_	_	σ

Assumptions and Conventions

In the descriptions below, the syntax for an instruction always places the destination register before the source registers. Also, source registers can be replaced by compile-time constants where appropriate. Finally, access to the global work-queue is assumed to atomic, *i.e.* no other processor can change the work queue between the *before* and *after* parts of a rule.

Thread Instructions

The spawn instruction is used to create new activities on the thread queue.

spawn:

$$\frac{\langle\ \mathsf{R} \quad \mathsf{spawn}(\iota s', \mathsf{ra}, \mathsf{rb}, \mathsf{rc}) : \iota s \quad \omega s \quad \sigma\ \rangle}{\langle\ \mathsf{R} \quad \iota s \quad \omega s +\!\!+ [\omega] \quad \sigma\ \rangle}$$

where $\omega = (\iota s',\ (\mathsf{ra}),\ (\mathsf{rb}),\ (\mathsf{rc}))$

The schedule instruction is used to start a new thread. It extracts a thread from the front of the work queue and initializes the machine state accordingly. If the work queue is empty, then we set the processor in a scheduling loop. Essentially, we make the processor wait until some other processor spawns work into the queue.

schedule:

$$\frac{\langle\ \mathsf{R} \quad \mathsf{schedule} : \iota s \quad (\iota s', \nu, \ell_B, \ell_F) : \omega s \quad \sigma\ \rangle}{\langle\ \mathsf{R}' \quad \iota s' \quad \omega s \quad \sigma\ \rangle}$$

where $\mathsf{R}' = \mathsf{R}[\ \mathsf{r0} \leftarrow \nu;\ \mathsf{rB} \leftarrow \ell_B;\ \mathsf{rF} \leftarrow \ell_F\ ]$

$$\frac{\langle\ \mathsf{R} \quad \mathsf{schedule} : \iota s \quad [\,] \quad \sigma\ \rangle}{\langle\ \mathsf{R} \quad \mathsf{schedule} : \iota s \quad [\,] \quad \sigma\ \rangle}$$

Memory Instructions

The touch instruction is used to check the status of a location in memory. If the location is full, the instruction does nothing. Otherwise, the status of the location is defer. In this case, the touch instruction suspends the current thread and adds it to the list of threads suspended at the location. It also replaces the instruction sequence with a single schedule instruction. This will have the effect of executing the next thread in the work queue.

touch:

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

$$\frac{\langle\ \mathsf{R}\quad \mathsf{touch}(\underline{\ell}) : \iota s\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{defer}\ \omega s'\rangle]\ \rangle}{\langle\ \mathsf{R}\quad [\mathsf{schedule}]\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{defer}\ \omega : \omega s'\rangle]\ \rangle}$$

where $\omega = (\iota s,\ (\mathsf{r0}),\ (\mathsf{rB}),\ (\mathsf{rF}))$

The load instruction reads a value from memory into a machine register. It expects the status of the location in memory to be full.

load:

$$\frac{\langle\ \mathsf{R}\quad \mathsf{load}(\mathsf{ra}, \underline{\ell}) : \iota s\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{full}\ \nu\rangle]\ \rangle}{\langle\ \mathsf{R}'\quad \iota s\quad \omega s\quad \sigma\ \rangle}$$

where $\mathsf{R}' = \mathsf{R}[\ \mathsf{ra} \leftarrow \nu\ ]$

The store instruction writes the contents of a register into memory, without regard for the status of the target location.

store:

$$\frac{\langle\ \mathsf{R}\quad \mathsf{store}(\underline{\ell}, \mathsf{ra}) : \iota s\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \_\ \_\rangle]\ \rangle}{\langle\ \mathsf{R}\quad \iota s\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{full}\ (\mathsf{ra})\rangle]\ \rangle}$$

In contrast, the istore instruction checks the status of the target location. If the status is defer, the instruction not only writes the value to memory but also adds all threads waiting for that value to the end of the work queue. If the status of the location is full, then istore signals an error by storing the error code <u>storerr</u> into register r0 and by replacing the instruction sequence with [halt]

istore:

$$\frac{\langle\ \mathsf{R}\quad \mathsf{istore}(\underline{\ell}, \mathsf{ra}) : \iota s\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{defer}\ \omega s'\rangle]\ \rangle}{\langle\ \mathsf{R}\quad \iota s\quad \omega s +\!\!+ \omega s'\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{full}\ (\mathsf{ra})\rangle]\ \rangle}$$

$$\frac{\langle\ \mathsf{R}\quad \mathsf{istore}(\underline{\ell}, \mathsf{ra}) : \iota s\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{full}\ \nu\rangle]\ \rangle}{\langle\ \mathsf{R}'\quad [\mathsf{halt}]\quad \omega s\quad \sigma\ \rangle}$$

where $\mathsf{R}' = \mathsf{R}[\ \mathsf{r0} \leftarrow \underline{\mathsf{storerr}}\ ]$

The take instruction is used to read values from memory with mutual exclusion. If the status of a memory location is full, the take instruction returns the value in the location and sets the status of the location to defer. If the value of the location is defer, the instruction suspends the current thread *including the* take *instruction* and waits for the value of the location to arrive. Including the take instruction in the suspension allows us to use a simple istore instruction to force all activities waiting for the location to reissue the take.

Only one of these subsequent takes will succeed, and thus we implement mutual exclusion on the location.

take:

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

where $\mathsf{R}' = \mathsf{R}[\ \mathsf{ra} \leftarrow \nu\ ]$

$$\frac{\langle\ \mathsf{R}\quad \mathsf{take}(\mathsf{ra},\underline{\ell}) : \iota s\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{defer}\ \omega s'\rangle]\ \rangle}{\langle\ \mathsf{R}\quad [\mathsf{schedule}]\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{defer}\ \omega : \omega s'\rangle]\ \rangle}$$

where $\omega = (\mathsf{take}(\mathsf{ra},\underline{\ell}) : \iota s,\ (\mathsf{r0}),\ (\mathsf{rB}),\ (\mathsf{rF}))$

Primitive Function Instructions

These instructions are used to implement the standard arithmetic functions available on all machines. Operands are always registers (or constants). We use addition as an example.

plus:

$$\frac{\langle\ \mathsf{R}\quad \mathsf{plus}(\mathsf{ra},\mathsf{rb},\mathsf{rc}) : \iota s\quad \omega s\quad \sigma\ \rangle}{\langle\ \mathsf{R}'\quad \iota s\quad \omega s\quad \sigma\ \rangle}$$

where $\mathsf{R}' = \mathsf{R}[\ \mathsf{ra} \leftarrow (\mathsf{rb}) + (\mathsf{rc})\ ]$

The $\mathsf{allocate}$ instruction abstracts the primitive mechanisms for allocating fresh memory locations from the shared global store. Notice that rb is expected to contain the number of new memory locations to allocate.

allocate:

$$\frac{\langle\ \mathsf{R}\quad \mathsf{allocate}(\mathsf{ra},\mathsf{rb}) : \iota s\quad \omega s\quad \sigma\ \rangle}{\langle\ \mathsf{R}'\quad \iota s\quad \omega s\quad \sigma'\ \rangle}$$

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

Control Flow Instructions

The exec instruction changes the instruction sequence. It expects a new instruction sequence in register ra. In addition, it saves (a reference to) the instruction sequence after the exec in register rb. This instruction is similar to the "branch-and-link" instructions found on many real processors.

exec:

$$\frac{\langle\ \mathsf{R}\quad \mathsf{exec}(\mathsf{ra},\mathsf{rb}) : \iota s\quad \omega s\quad \sigma\ \rangle}{\langle\ \mathsf{R}'\quad (\mathsf{ra})\quad \omega s\quad \sigma\ \rangle}$$

where $\mathsf{R}' = \mathsf{R}[\ \mathsf{rb} \leftarrow \iota s\ ]$

The switch instruction chooses an instruction sequence based on the value in register ra.

switch:

$$\frac{\langle\ \mathsf{R}\quad \mathsf{switch}(\mathsf{ra}, \iota s_1, \ldots, \iota s_k) : \iota s\quad \omega s\quad \sigma\ \rangle}{\langle\ \mathsf{R}\quad \iota s_{(\mathsf{ra})} +\!\!+\, \iota s\quad \omega s\quad \sigma\ \rangle}$$

where $1 \leq (\mathsf{ra}) \leq k$

Barrier Instructions

The following instructions initialize and manipulate barrier objects. The initbar instruction is used to create a fresh barrier object. Only the first two fields of the barrier object are initialized. The value of the third field is left empty.

initbar:

$$\frac{\langle\ \mathsf{R}\quad \mathsf{initbar}(\underline{\ell}, \mathsf{ra}, \mathsf{rb}) : \iota s\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{defer}\ [\ ]\rangle]\ \rangle}{\langle\ \mathsf{R}'\quad \iota s\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{full}\ \nu\rangle]\ \rangle}$$

where $\nu = ((\mathsf{ra}),\ (\mathsf{rb}),\ \diamond)$
$\mathsf{R}' = \mathsf{R}[\ \mathsf{rB} \leftarrow \underline{\ell}\ ]$

The touchbar instruction checks the status of a barrier. If the barrier has discharged, as indicated by a counter value of zero, the instruction updates rB and makes the location of the barrier empty. If the barrier has not discharged, the instruction suspends the current thread and stores it in the third field of the barrier object. The compiler or programmer should guarantee that only one touchbar is performed per barrier.

touchbar:

$$\frac{\langle\ \mathsf{R}\quad \mathsf{touchbar}(\underline{\ell}) : \iota s\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{full}\ (0,\ \ell_B,\ \diamond)\rangle]\ \rangle}{\langle\ \mathsf{R}'\quad \iota s\quad \omega s\quad \sigma[\underline{\ell} \mapsto \diamond]\ \rangle}$$

where $\mathsf{R}' = \mathsf{R}[\ \mathsf{rB} \leftarrow \ell_B\ ]$

$$\frac{\langle\ \mathsf{R}\quad \mathsf{touchbar}(\underline{\ell}) : \iota s\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{full}\ (n,\ \ell_B,\ \diamond)\rangle]\ \rangle}{\langle\ \mathsf{R}\quad [\mathsf{schedule}]\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{full}\ (n,\ \ell_B,\ \omega)\rangle]\ \rangle}$$

where $n > 0$
$\omega = (\iota s,\ (\mathsf{r0}),\ \ell_B,\ (\mathsf{rF}))$

The incbar instruction increments the value of the counter in a barrier object. Notice that we do not care if the third field of the object is either a thread (ω) or empty ($\diamond$); we simply leave it unchanged.

incbar:

$$\frac{\langle\ \mathsf{R}\quad \mathsf{incbar}(\underline{\ell}) : \iota s\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{full}\ (n,\ \ell_B,\ \_)\rangle]\ \rangle}{\langle\ \mathsf{R}\quad \iota s\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{full}\ (n+1,\ \ell_B,\ \_)\rangle]\ \rangle}$$

The decbar instruction decrements the value of the counter in a barrier object. If the value of the counter is greater than one, the counter is simply decremented. If the value of the counter is one, then the instruction checks the value of the third field of the barrier. If this field is empty, then the counter is simply decremented. This situation arises when the last decbar for a particular barrier occurs before the touchbar for the barrier.

On the other hand if the third field contains a suspended thread, the decbar instruction schedules the thread and changes the location to contain an empty value rather than a valid barrier object. This situation results when the touchbar instruction occurs before the last decbar.

decbar:

$$\frac{\langle\ \mathsf{R}\quad \mathsf{decbar}(\underline{\ell}) : \iota s\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{full}\ (n,\ \ell_B,\ \omega)\rangle]\ \rangle}{\langle\ \mathsf{R}\quad \iota s\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{full}\ (n-1,\ \ell_B,\ \omega)\rangle]\ \rangle}$$

where $n > 1$

$$\frac{\langle\ \mathsf{R}\quad \mathsf{decbar}(\underline{\ell}) : \iota s\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{full}\ (1,\ \ell_B,\ \diamond)\rangle]\ \rangle}{\langle\ \mathsf{R}\quad \iota s\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{full}\ (0,\ \ell_B,\ \diamond)\rangle]\ \rangle}$$

$$\frac{\langle\ \mathsf{R}\quad \mathsf{decbar}(\underline{\ell}) : \iota s\quad \omega s\quad \sigma[\underline{\ell} \mapsto \langle \mathsf{full}\ (1,\ \ell_B,\ \omega)\rangle]\ \rangle}{\langle\ \mathsf{R}\quad \iota s\quad \omega s +\!\!+ [\omega]\quad \sigma[\underline{\ell} \mapsto \diamond]\ \rangle}$$

Miscellaneous Instructions

The move instruction transfers the contents of one register to another.

move:

$$\frac{\langle\ \mathsf{R}\quad \mathsf{move}(\mathsf{ra}, \mathsf{rb}) : \iota s\quad \omega s\quad \sigma\ \rangle}{\langle\ \mathsf{R}'\quad \iota s\quad \omega s\quad \sigma\ \rangle}$$

where $\mathsf{R}' = \mathsf{R}[\ \mathsf{ra} \leftarrow (\mathsf{rb})\ ]$

The print instruction displays the contents of a register.

print:

$$\frac{\langle\ \mathsf{R}\quad \mathsf{print}(\mathsf{ra}) : \iota s\quad \omega s\quad \sigma\ \rangle}{\langle\ \mathsf{R}\quad \iota s\quad \omega s\quad \sigma\ \rangle}$$

The halt instruction stops execution on all processors. The state of the processor on which the instruction is executed is not modified, *e.g.* the halt instruction is not removed from the front of the instruction sequence. This helps us establish whether or not the machine halted normally or due to an error.

halt:

$$\frac{\langle\ \mathsf{R}\quad \mathsf{halt} : \iota s\quad \omega s\quad \sigma\ \rangle}{\text{“stop execution on all processors”}}$$

5 A λ_S Compiler for SMT

The compiler generates code sequences assuming the execution environment described in Section 3. It is particularly useful to consult Figure 4 when studying the rules that follow.

5.1 Compilation Rules

The compilation rules are described in terms of three translation functions: TP, TE, and TS. These functions take λ_S source code for programs, expressions, and statements respectively as input; they return a sequence of abstract machine instructions. TE and TS are mutually recursive and are used for compiling the bulk of a λ_S program. TP is used to generate the "glue" code that sets up the initial execution environment for a program.

The compilation rules are presented as if they were written in a high-level programming language and often make use of variables and block-structure.

Identifier Map

The translation functions make use of an identifier map α. The purpose of this map is to translate a source-language identifier at compile-time into an offset (slot) in the activation frame. At run-time, the slot will contain a location where the value of the identifier can be found. The contents of this map are completely static and can be generated directly from the lexical structure of the source program. Also, since there is a single flat activation frame per function at run-time, there is also a single unique α for each lambda abstraction at compile-time.

References to Code

In the compilation rules, we manipulate sequences of instructions and even store these sequences in registers and data structures. We appeal to the reader's intuition on this point. What is really stored is a "reference" to the sequence, not the sequence itself.

Register r0

The compiled code makes use of register r0 in a special way. The values of all expressions are computed and placed in register r0. This allows subsequent computations to make use of the value r0. This is particularly evident in the compilation rule for statements in Section 5.1.2. Also notice that suspensive instructions like touch always save the value of r0.

5.1.1 TE: Compilation of Expressions

Constants

The compilation rule for constants is the simplest. The value of the constant is simply moved to register r0. As mentioned above, the compilation rules follow the convention that expression values are placed in r0.

$$\mathbf{TE}[\![c]\!]\ \alpha = [\ \mathsf{move}(\mathsf{r0}, \underline{\mathsf{c}})\ ]$$

Identifiers

To get the value of an identifier, we must first make sure that the memory location where it is stored is full. The touch instruction performs this operation. If the location is indeed full, the load instruction proceeds. If the location is not full, the thread will suspend before the load instruction and will wait for the value of the identifier to be computed by some other thread.

The addressing mode used here is important. We are performing an indexed indirection through register rF, the activation frame pointer register. The α map is used to find the slot in the frame corresponding to the identifier. At run-time, this slot contains the *location* where the value of the identifier can be found. This touch-load idiom appears in several compilation rules.

$$\mathbf{TE}[\![x]\!]\ \alpha = [\ \mathsf{touch}(@\mathsf{rF}[\underline{\mathsf{x}}]), \mathsf{load}(\mathsf{r0}, @\mathsf{rF}[\underline{\mathsf{x}}])\ ]$$

where $\underline{\mathsf{x}} = \alpha(x)$

Primitive Functions

Primitive functions are strict in all their arguments. We touch each identifier first. When all these have succeeded, we can be certain that the values of the identifiers have been computed. We load each value into a register and perform the operation using the underlying primitive instructions of the machine (like plus). Notice that following the established convention, the value of the primitive operation is returned in r0.

$$\mathbf{TE}[\![PF^n(x_1, \ldots, x_n)]\!]\ \alpha = [\ \begin{array}{l}\mathsf{touch}(@\mathsf{rF}[\underline{\mathsf{x}_1}]), \ldots, \mathsf{touch}(@\mathsf{rF}[\underline{\mathsf{x}_n}]),\\ \mathsf{load}(\mathsf{r1}, @\mathsf{rF}[\underline{\mathsf{x}_1}]), \ldots, \mathsf{load}(\mathsf{rn}, @\mathsf{rF}[\underline{\mathsf{x}_n}]), \mathsf{pf}^n(\mathsf{r0}, \mathsf{r1}, \ldots, \mathsf{rn})\ ]\end{array}$$

where $\underline{\mathsf{x}_1} = \alpha(x_1), \ldots, \underline{\mathsf{x}_n} = \alpha(x_n)$

iFetch and mFetch

`iFetch` and `mFetch` are treated separately from the other primitive functions by the compilation rules. First, these rules load the value of the argument x into r0. The value of the identifier is a memory location from which a second load or take is performed. In this sequence, it is crucial that we load the value of x into r0 because the value of the register must be preserved if the memory operation that uses the register suspends. The only register preserved by suspensive operations like touch and take is r0.

$$\mathbf{TE}[\![\mathtt{iFetch}(x)]\!]\ \alpha = [\ \mathsf{touch}(@\mathsf{rF}[\underline{\mathsf{x}}]), \mathsf{load}(\mathsf{r0}, @\mathsf{rF}[\underline{\mathsf{x}}]), \mathsf{touch}(\mathsf{r0}), \mathsf{load}(\mathsf{r0}, \mathsf{r0})\ ]$$

where $\underline{\mathsf{x}} = \alpha(x)$

$$\mathbf{TE}[\![\mathtt{mFetch}(x)]\!]\ \alpha = [\ \mathsf{touch}(@\mathsf{rF}[\underline{\mathsf{x}}]), \mathsf{load}(\mathsf{r0}, @\mathsf{rF}[\underline{\mathsf{x}}]), \mathsf{take}(\mathsf{r0}, \mathsf{r0})\ ]$$

where $\underline{\mathsf{x}} = \alpha(x)$

Projections

The primitive projection function Prj^k in the source program returns the k-th field of a constructor. Given the representation of constructors we have chosen, this operation is straightforward to implement using SMT instructions. First, we load the value of identifier x, which is the address of the constructor. Then we use the address of the constructor with a second touch-load sequence to get the value of the field.

$$\mathbf{TE}[\![\mathtt{Prj}^k(x)]\!]\ \alpha = [\ \mathsf{touch}(@\mathsf{rF}[\underline{\mathsf{x}}]), \mathsf{load}(\mathsf{r0}, @\mathsf{rF}[\underline{\mathsf{x}}]), \mathsf{touch}(@\mathsf{r0}[\underline{\mathsf{k+1}}]), \mathsf{load}(\mathsf{r0}, @\mathsf{r0}[\underline{\mathsf{k+1}}])\ ]$$

where $\underline{\mathsf{x}} = \alpha(x)$

Lambda Abstractions

The rule describing the compilation of lambda abstractions is perhaps the most complicated of all. In our scheme, functions are responsible for allocating and initializing their own activation frames. Callers actually pass very little information to callees during the call sequence. The calling convention specifies the following contents for registers:

Register	On Entry	On Exit
r0	Argument location	Result
r1	Return instruction sequence	—
r2	Callee closure	—
rF	Caller's frame	Caller's frame
rB	Caller's barrier	Caller's barrier

The lambda compilation rule is composed of several parts. Each is described below. Figure 4 is crucial is to understand this translation rule:

- *body* : the body of the lambda is compiled using TE with a *modified* identifier map. This identifier map describes the locations of identifiers in the frame that will be created when the function is actually invoked.

- *savestate* : this code sequence allocates an activation frame. It saves the caller's activation frame pointer, the argument *location* (due to non-strictness), and the return instruction sequence in the new frame.
- *copyFV* : this code sequence copies the *locations*, not the values, of the free variables from the callee's closure to the activation frame.
- *locals* : this code sequence allocates space for the local variables of the function and copies these *locations* into the activation frame.
- *return* : this code sequence returns control to the caller. Notice that this code does not touch r0 since it assumes the result resides there.
- *closure* : this code sequence constructs the closure data structure which represents the lambda abstraction. It allocates a block of memory, inserts a reference to the function code, and copies the *locations* of all free variables of the function from the current frame to the closure. Remember that closures are non-strict in the value of free variables, so we must copy locations, not values, into the closure to capture the lexical environment. The location of this closure (the first location of the memory block allocated for it) is returned in r0 as the value of the lambda abstraction.

$$
\begin{array}{l}
\mathbf{TE}[\![\lambda\ x.E]\!]\ \alpha = \\
\quad \{\ \ \mathit{body} = \mathbf{TE}[\![E]\!]\ \alpha' \\
\qquad \mathit{savestate} = [\ \mathsf{allocate}(\mathsf{r3}, \underline{\mathsf{k}}), \mathsf{store}(\mathsf{r3}, \mathsf{rF}), \mathsf{move}(\mathsf{rF}, \mathsf{r3}), \mathsf{store}(\mathsf{rF}[\underline{2}], \mathsf{r0}), \mathsf{store}(\mathsf{rF}[\underline{1}], \mathsf{r1})\ ] \\
\qquad \mathit{copyFV} = [\ \mathsf{load}(\mathsf{r0}, \mathsf{r2}[\underline{1}]), \mathsf{store}(\mathsf{rF}[\underline{3}], \mathsf{r0}), \ldots, \mathsf{load}(\mathsf{r0}, \mathsf{r2}[\underline{\mathsf{m}}]), \mathsf{store}(\mathsf{rF}[\underline{2{+}\mathsf{m}}], \mathsf{r0})\ ] \\
\qquad \mathit{locals} = [\ \mathsf{allocate}(\mathsf{r4}, \underline{\mathsf{n}}), \mathsf{store}(\mathsf{rF}[\underline{3{+}\mathsf{m}}], \mathsf{r4}), \mathsf{plus}(\mathsf{r4}, \mathsf{r4}, \underline{1}), \mathsf{store}(\mathsf{rF}[\underline{4{+}\mathsf{m}}], \mathsf{r4}), \\
\qquad\qquad \ldots, \mathsf{plus}(\mathsf{r4}, \mathsf{r4}, \underline{1}), \mathsf{store}(\mathsf{rF}[\underline{2{+}\mathsf{m}{+}\mathsf{n}}], \mathsf{r4})\ ] \\
\qquad \mathit{buildframe} = \mathit{savestate} +\!\!+\ \mathit{copyFV} +\!\!+\ \mathit{locals} \\
\qquad \mathit{return} = [\ \mathsf{load}(\mathsf{r1}, \mathsf{rF}[\underline{1}]), \mathsf{load}(\mathsf{rF}, \mathsf{rF}), \mathsf{exec}(\mathsf{r1}, \_)\ ] \\
\qquad \mathit{func} = \mathit{buildframe} +\!\!+\ \mathit{body} +\!\!+\ \mathit{return} \\
\qquad \mathit{closure} = [\ \mathsf{allocate}(\mathsf{r0}, \underline{\mathsf{m}{+}1}), \mathsf{store}(\mathsf{r0}, \mathit{func}), \\
\qquad\qquad \mathsf{load}(\mathsf{r1}, \mathsf{rF}[\underline{\mathsf{y}_1}]), \mathsf{store}(\mathsf{r0}[\underline{1}], \mathsf{r1}), \ldots, \mathsf{load}(\mathsf{r1}, \mathsf{rF}[\underline{\mathsf{y}_m}]), \mathsf{store}(\mathsf{r0}[\underline{\mathsf{m}}], \mathsf{r1})\ ] \\
\qquad \mathtt{in} \\
\qquad\qquad \mathit{closure} \\
\qquad \}
\end{array}
$$

$$
\begin{array}{ll}
\text{where} & \underline{\mathsf{m}} = \mid FV(\lambda x.E) \mid \\
& y_1, \ldots, y_m = FV(\lambda x.E) \\
& \underline{\mathsf{y}_1} = \alpha(y_1), \ldots, \underline{\mathsf{y}_m} = \alpha(y_m) \\
& \underline{\mathsf{n}} = \mid BV(E) \mid \\
& b_1, \ldots, b_n = BV(E) \\
& \underline{\mathsf{k}} = \underline{\mathsf{m}} + \underline{\mathsf{n}} + 3 \\
& \alpha'(x) = \{\ x \mapsto 2, y_1 \mapsto 3, \ldots, y_m \mapsto 2 + \underline{\mathsf{m}}, b_1 \mapsto 3 + \underline{\mathsf{m}}, \ldots, b_n \mapsto 2 + \underline{\mathsf{m}} + \underline{\mathsf{n}}\ \}
\end{array}
$$

In the preceding description, *FV(E)* denotes the free variables of expression E; | *FV(E)* | denotes the number of free variables in E; | *BV(E)* | denotes the number of local variables and barriers bound in E, with the exception of those that are bound inside of lambda abstractions within E. *BV(E)* denotes the set of local variables.

Application

Most of the work involved in an application is actually performed by the callee, making the caller's side of a function call relatively simple. All that needs to be done is to load the correct values into the registers as required by the calling convention given in the lambda rule. In this rule, the value of f is expected to be a closure.

$$
\begin{array}{rl}
\mathbf{TE}[\![f\ \ x]\!]\ \alpha = [& \mathsf{touch}(\mathsf{@rF}[\underline{\mathsf{f}}]), \mathsf{load}(\mathsf{r2}, \mathsf{@rF}[\underline{\mathsf{f}}]), \mathsf{load}(\mathsf{r3}, \mathsf{r2}), \\
& \mathsf{load}(\mathsf{r0}, \mathsf{rF}[\underline{\mathsf{x}}]), \mathsf{exec}(\mathsf{r3}, \mathsf{r1})\ ]
\end{array}
$$

where $\underline{\mathsf{f}} = \alpha(f)$, $\underline{\mathsf{x}} = \alpha(x)$

Constructors

The code sequence to generate constructors is quite simple given their flat structure. A block of memory is allocated, the tag value is stored in the first slot, and the locations of the field values are stored in subsequent slots.

$$\mathbf{TE}[\![CN^k(x_1,\ldots,x_k)]\!]\ \alpha = [\ \mathsf{allocate}(\mathsf{r0},\underline{\mathsf{k{+}1}}),\mathsf{store}(\mathsf{r0},\underline{\mathsf{CN}}),\\ \mathsf{load}(\mathsf{r1},\mathsf{rF}[\underline{\mathsf{x}_1}]),\mathsf{store}(\mathsf{r0}[\underline{\mathsf{1}}],\mathsf{r1}),\ldots,\mathsf{load}(\mathsf{r1},\mathsf{rF}[\underline{\mathsf{x}_k}]),\mathsf{store}(\mathsf{r0}[\underline{\mathsf{k}}],\mathsf{r1})\ ]$$

where $\underline{\mathsf{x}_1} = \alpha(x_1),\ldots,\underline{\mathsf{x}_k} = \alpha(x_k)$
$\underline{\mathsf{CN}} = \textit{tag for } CN^k$

Case Expressions

Each branch of a *Case* expression is compiled separately. The *Case* expression is strict in the value of x, so this value is read into a register using the standard touch-load idiom. The primitive switch instruction takes care of managing the control flow based on the value in the register.

$$\mathbf{TE}[\![Case(x,E_1,\ldots,E_k)]\!]\ \alpha = \{\ \ \iota s_1 = \mathbf{TE}[\![E_1]\!]\ \alpha \\ \ldots \\ \iota s_k = \mathbf{TE}[\![E_k]\!]\ \alpha \\ \texttt{in} \\ [\ \mathsf{touch}(\mathsf{@rF}[\underline{\mathsf{x}}]),\mathsf{load}(\mathsf{r1},\mathsf{@rF}[\underline{\mathsf{x}}]),\mathsf{switch}(\mathsf{r1},\iota s_1,\ldots,\iota s_k)\ ] \\ \}$$

where $\underline{\mathsf{x}} = \alpha(x)$

Blocks

In the most general case, the semantics of λ_S prevent the compiler from statically determining the order in which the statements in a block will execute. Consequently, statements must be executed in a separate thread. This rule begins by incrementing the counter of the current barrier as a prelude to creating the thread that will execute the statements. The compilation rules maintain the invariant that for every statement in the program, there is a corresponding incbar or initbar instruction. In turn, the last instruction in the code sequence for every statement is decbar. Thus, proper barrier behavior is maintained.

$$\mathbf{TE}[\![\{S\ \texttt{in}\ x\}]\!]\ \alpha = \{\ \ \iota s = (\mathbf{TS}[\![S]\!]\ \alpha) +\!\!+ [\mathsf{schedule}] \\ \iota s_x = \mathbf{TE}[\![x]\!]\ \alpha \\ \texttt{in} \\ [\ \mathsf{incbar}(\mathsf{rB}),\mathsf{spawn}(\iota s,\_,\mathsf{rB},\mathsf{rF})\ ] +\!\!+ \iota s_x \\ \}$$

5.1.2 TS: Compilation of Statements

Parallel Statements

The order in which statements S_1 and S_2 execute cannot always be determined at compile-time. Therefore, the code to execute each statement is placed in its own thread. It is interesting to note that we only need to do include a single incbar instruction in the code sequence: we always "inherit" an incbar from a compilation rule at a higher level.

$$\mathbf{TS}[\![S_1\ ;\ S_2]\!]\ \alpha = \{\ \ \iota s_1 = (\mathbf{TS}[\![S_1]\!]\ \alpha) +\!\!+ [\ \mathsf{schedule}\ ] \\ \iota s_2 = (\mathbf{TS}[\![S_2]\!]\ \alpha) +\!\!+ [\ \mathsf{schedule}\ ] \\ \texttt{in} \\ [\ \mathsf{incbar}(\mathsf{rB}),\mathsf{spawn}(\iota s_1,\_,\mathsf{rB},\mathsf{rF}),\mathsf{spawn}(\iota s_2,\_,\mathsf{rB},\mathsf{rF})\ ] \\ \}$$

Sequential Statements

In this rule, the execution of statements S_1 and S_2 is controlled by the discharge of the barrier separating them. Notice that the barrier has a unique name 'b'. Like any other local variable, it has a slot in the activation frame associated with it. The code initializes barrier b with a counter value of 1, executes the code for S_1, waits for the barrier to discharge and executes the code for S_2. The barrier guarantees that all values computed and all side-effects performed by S_1 will complete before any part of S_2 executes.

$$
\begin{array}{ll}
\mathbf{TS}[\![S_1 \text{ –}b\text{– } S_2]\!]\ \alpha = \{ & \iota s_1 = \mathbf{TS}[\![S_1]\!]\ \alpha \\
& \iota s_2 = \mathbf{TS}[\![S_2]\!]\ \alpha \\
& \texttt{in} \\
& \quad [\ \mathsf{initbar}(\mathsf{@rF}[\underline{\mathsf{b}}], 1, \mathsf{rB})\ ] +\!\!+ \iota s_1 +\!\!+ [\ \mathsf{touchbar}(\mathsf{@rF}[\underline{\mathsf{b}}])\ ] +\!\!+ \iota s_2 \\
& \}
\end{array}
$$

where $\underline{\mathsf{b}} = \alpha(b)$

Threaded Statements

The '$\sim$' separating S_1 and S_2 asserts it is safe but not necessary to execute some or all of S_1 (the pre-region) before executing S_2 (the post-region). Consequently, the code for the two statements can be combined into a single sequence. Again, only a single incbar is required to maintain parity with the decbar instructions executed at the end of each of the code sequences for the statements.

$$
\begin{array}{ll}
\mathbf{TS}[\![S_1 \ \sim \ S_2]\!]\ \alpha = \{ & \iota s_1 = \mathbf{TS}[\![S_1]\!]\ \alpha \\
& \iota s_2 = \mathbf{TS}[\![S_2]\!]\ \alpha \\
& \texttt{in} \\
& \quad [\ \mathsf{incbar}(\mathsf{rB})\ ] +\!\!+ \iota s_1 +\!\!+ \iota s_2 \\
& \}
\end{array}
$$

Simple Statements

A basic binding is the simplest statement to compile. The code that computes the value of expression E is compiled and combined with code that stores the value in the heap. Notice that the istore instruction assumes the value of the expression exists in register r0. Finally, the current barrier is decremented indicating the completion of the statement.

$$\mathbf{TS}[\![x = E]\!]\ \alpha = (\mathbf{TE}[\![E]\!]\ \alpha) +\!\!+ [\ \mathsf{istore}(\mathsf{@rF}[\underline{\mathsf{x}}], \mathsf{r0}), \mathsf{decbar}(\mathsf{rB})\ ]$$

where $\underline{\mathsf{x}} = \alpha(x)$

The compilation rules for the side-effecting statements `iStore` and `mStore` are identical. Both store a value on the heap. Once the value has been stored succesfully, both decrement the current barrier, indicating that the statement has completed. We only give the rule for `iStore`.

$$\mathbf{TS}[\![\texttt{iStore}(x, y)]\!]\ \alpha = [\mathsf{touch}(\mathsf{@rF}[\underline{\mathsf{x}}]), \mathsf{touch}(\mathsf{@rF}[\underline{\mathsf{y}}]), \mathsf{load}(\mathsf{r1}, \mathsf{@rF}[\underline{\mathsf{x}}]), \mathsf{load}(\mathsf{r2}, \mathsf{@rF}[\underline{\mathsf{y}}]), \mathsf{istore}(\mathsf{r1}, \mathsf{r2}), \mathsf{decbar}(\mathsf{rB})]$$

where $\underline{\mathsf{x}} = \alpha(x)$
$\underline{\mathsf{y}} = \alpha(y)$

5.1.3 TP: Compilation of Programs

The translation function TP is only called for top-level expressions, i.e. those that are to start the computation on the abstract machine. The TP rule builds the initial execution environment for the compiled code. This includes building the initial activation frame and creating an initial global barrier. The purpose of this global barrier is to serve as a synchronization point for all "top-level" activities in the program. Once the global barrier discharges, we can be certain that all threads in the program, except the one in the post-region of the global barrier, have terminated.

The code generated does the following:

- *init* : this code sequence allocates the global barrier and initializes its counter to 1. This barrier is used to keep track of all top-level activities in the program.
- *user* : this code sequence creates a closure out of the user expression E. We "wrap" the user code in a function so that by invoking this closure we can create the initial activation frame for the code. The parameter to the function is not used.
- *calluser* : this code sequence invokes the function containing the user expression, passing it a dummy argument. This operation has the effect of creating the initial activation frame and of executing the user's code.
- *terminate* : this code sequence executes after the user's code returns a value. The sequence prints the result value, decrements the global barrier, and waits for it to discharge. The discharge of this barrier guarantees that all computation has completed correctly. The code prints the message $\underline{\mathsf{done}}$, and forces the machine to halt. If the machine halts before this barrier is discharged, then the program encountered either a run-time error or a deadlock condition.

The TP rule assumes that the top-level expression E contains no free variables.

$$
\begin{array}{ll}
\mathbf{TP}[\![E]\!] = \{ & \mathit{init} = [\ \mathsf{allocate}(\mathsf{rB}, \underline{1}), \mathsf{initbar}(\mathsf{rB}, 1, \mathsf{rB})\ ] \\
& \mathit{user} = \mathbf{TE}[\![\lambda x.E]\!]\ \alpha_0 \\
& \mathit{calluser} = [\ \mathsf{move}(\mathsf{r2}, \mathsf{r0}), \mathsf{load}(\mathsf{r3}, \mathsf{r2}), \mathsf{move}(\mathsf{r0}, \underline{\mathsf{dummy}}), \mathsf{exec}(\mathsf{r3}, \mathsf{r1})\ ] \\
& \mathit{terminate} = [\ \mathsf{print}(\mathsf{r0}), \mathsf{decbar}(\mathsf{rB}), \mathsf{touchbar}(\mathsf{rB}), \mathsf{print}(\underline{\mathsf{done}}), \mathsf{halt}\ ] \\
& \mathit{main} = \mathit{init} +\!\!+\ \mathit{user} +\!\!+\ \mathit{calluser} +\!\!+\ \mathit{terminate} \\
& \mathtt{in} \\
& \qquad \mathit{main} \\
& \}
\end{array}
$$

where x is a unique identifier
$\alpha_0(x) = \{\ \}$

5.2 Improvements and Optimizations

The compilation of λ_S programs to SMT instructions as embodied in the TE, TS, and TP translation functions was designed to serve as a "proof of concept." The translation functions handle the entire λ_S language (and thus the entire pH language) while maintaining correct non-strict semantics in the presence of barriers, closures, and data structures. We do not claim that the translation functions, as given, produce efficient code. The code can be improved in many ways, as the attentive reader is sure to point out. We discuss some of these optimizations in the following sections. We do claim, however, that the SMT model embodies the characteristics of a real machine that need to be exposed to a compiler in order to generate good code. In other words, we can improve the translation rules significantly, but we do not expect to make major changes to the SMT model in order to produce code that will run well on real SMPs.

The compilation process presented is characterized by its syntax-directed, top-down decomposition of the problem. This structure imposes a number of limitations on the amount of analysis that can be performed on the program source. In particular, "global" analyses are difficult to express in a syntax-directed translation since each translation rule has a narrow, local view of the entire program. Most of the optimizations described below require global analysis of the program. We do not describe how to perform the analyses, so we appeal to the reader's intuition that they are indeed possible.

5.2.1 Partitioning

In the context of λ_S, the goal of partitioning analysis is to replace parallel statement separators ';' with threaded separators '$\sim$'. It should be clear from the compilation rules that threaded statements should execute much more efficiently than parallel statements. Threaded statements avoid the thread creation and scheduling overheads associated with parallel statements. In addition, machine-level optimizations can take advantage of the larger instruction sequences that result from threading in order to allocate resources, like registers, more efficiently. λ_S and SMT have been designed carefully to incorporate the results of partitioning analysis, when available.

5.2.2 Expanded Use of Registers

The current compilation rules use SMT registers sparingly. In particular, local variables are always read from the heap through the frames. The SMT machinery does not preclude a more aggressive use of registers, so that, for example, values computed early in a thread could be used directly, without going through the frame, by computations later on in the thread. This use of registers is particularly important once a partitioning transformation has been performed.

The only limitation imposed by the machine on the use of registers is that the code must be careful to save live register values across suspension points. Remember the only general purpose register saved by suspensive instructions like touch and take is r0. Any other live registers must be stored in the activation frame of the thread explicitly. This type of analysis is quite natural for a compiler and straightforward to incorporate into a more aggressive compilation scheme.

5.2.3 Removal of Unnecessary Barrier Instructions

The translation functions maintain a very simple invariant to guarantee the proper behavior of barriers: every statement is preceded by an incbar or initbar instruction, and the last instruction in every statement is a decbar. In some cases, this simple-minded approach introduces instructions which are clearly redundant, as in the case of threaded statements. The initbar instruction before the instruction sequence for S_1 can be eliminated along with the last instruction in the sequence, a decbar. We could introduce a more complicated translation rule that would avoid the superfluous instructions, but this would add complexity to the compilation scheme.

Another simple code improvement would "lift out" many of the incbar instructions in an expression. These could be absorbed by the initbar instruction of the corresponding barrier through the use of an initialization count greater than one. Of course, this optimization cannot be performed with all incbar instructions; those inside a conditional cannot be lifted out. Likewise, those inside a function inheriting a barrier from its caller cannot be lifted out.

5.2.4 More Efficient Activation Frame Structure

We use flat activation frames mainly to simplify the exposition of the compilation rules. To maintain this structure, however, we must perform many load and store operations when an activation frame is created: some of these are used to copy the free variable locations from the closure into the frame; others are used to copy the locations of local variables into the frame. A more clever structure for activation frames would make use of a two level structure. At one level, the frame would store information about the caller and about the function argument; at a second level, the frame would contain the location of the vector of free variables in the closure. In addition, the frame would also contain the location of the block of storage allocated for local variables. This two level structure would reduce the number of load and store instructions in the code, but it would require slightly different code for loading the value of a free variable than for loading the value of a local variable.

5.2.5 Deallocation of Frames

Our compilation rules do not deallocate activation frames explicitly. We posit the existence of some automatic system for garbage collection, and we believe our rules do not lead to any systematic storage leaks. Nevertheless, it is possible to make a minor extension to SMT to introuce a deallocate instruction for frames and general heap structures. A possible scheme for deallocating frames involves the use of a barrier per function invocation. The function code is compiled so that that once this barrier is discharged, no more references will be made through the activation frame. Thus, when the barrier is discharged, the frame can be deallocated. Aggressive analysis of the code might even be able to eliminate the barrier altogether, as in the case of fully strict procedure calls.

6 Conclusions

We have presented a complete scheme for compiling the non-strict language pH (in its kernel form λ_S) into the instruction set of an abstract machine that behaves much like a real SMP. The compilation process is based on the direct semantics of barriers, produces suspensive thread code, and avoids the pitfalls of compiling through dataflow graphs.

We are currently designing a basic implementation of the compiler described here. We plan to generate C code for suspensive, user-level threads that are scheduled by a fast, work-stealing run-time system. We are also examining the implementation of a broad range of optimizations in this framework, and we believe that by abandoning the restrictions of syntax-directed translation we will generate much better code. We are also working to formulate partitioning algorithms—based on the use of '$\sim$'—in this framework. Existing partitioning algorithms are designed exclusively around dataflow graphs. We believe a more "language-centric" approach to partioning will yield code that will execute more efficiently on standard processor architectures.

References

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

[2] Shail Aditya, Arvind, and Joseph E. Stoy. Semantics of Barriers in a Non-Strict, Implicitly-Parallel Language. In *Proceedings of the 7th ACM Conference on Functional Programming and Computer Architecture*, La Jolla, CA, 1995.

[3] Zena M. Ariola and Arvind. Compilation of Id. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing Semantics Based Program Manipulation*, Santa Clara, CA, August 1991.

[4] Arvind, R.S. Nikhil, J.E. Stoy, and J. Maessen. λ_S: A λ-calculus with Letrec-blocks, Constants, Barriers, and Side-effects. Technical report, Computation Structures Group, MIT Laboratory for Computer Science, 1996. In preparation.

[5] Satyan R. Coorg. Partitioning Non-strict Functional Languages for Multi-threaded Code Generation. In *Proceedings of Static Analysis Symposium '95*, Glasgow, Scotland, UK, September 1995.

[6] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Santa Clara, CA, April 1991.

[7] Cormac Flanagan and R.S. Nikhil. pHluid: The design of a Parallel Functional Language Implementation. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, PA, May 1996. To appear.

[8] P. Hudak, S. Peyton Jones, and P. 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] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International, London, 1987.

[10] K. E. Schauser. *Compiling Lenient Languages for Parallel Asynchronous Execution.* PhD thesis, University of California, Berkeley, Computer Science Div., May 1994. appears as UCB CSD-94-832.

[11] Kenneth R. Traub. *Implementation of Non-Strict Functional Programming Languages.* Research Monographs in Parallel and Distributed Computing. The MIT Press, 1991.

[12] K.R. Traub, G.M. Papadopoulos, M.J. Beckerle, J.E. Hicks, and J. Young. Overview of the Monsoon Project. In *Proceedings of the 1991 IEEE International Conference on Computer Design*, Cambridge, MA, October 1991.