Generating Multithreaded Code from Parallel Haskell
for Symmetric Multiprocessors

by

Alejandro Caro

S.B., Computer Science and Engineering, MIT 1990
S.M., Electrical Engineering and Computer Science, MIT 1993
Engineer in Computer Science, MIT 1998

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1999

© 1999 Massachusetts Institute of Technology. All rights reserved.
Generating Multithreaded Code from Parallel Haskell
for Symmetric Multiprocessors

by
Alejandro Caro

Submitted to the Department of Electrical Engineering and Computer Science on January 27, 1999, in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Abstract
This dissertation presents pHc, a new compiler for Parallel Haskell (pH) with complete support for the entire language. pH blends the powerful features of a high-level, higher-order language with implicitly parallel, non-strict semantics. The result is a language that is easy to use but tough to compile. The principal contribution of this work is a new code-generation algorithm that works directly on λ*, a terse intermediate representation based on the λ-calculus. All the power of the original language is preserved in λ*, and in addition, the new representation makes it easy to express the important concept of threads, groups of expressions that can execute sequentially, at a high-level.

Code generation proceeds in two steps. First, the compiler turns λ* programs into multithreaded code for the Shared-Memory Threaded (SMT) machine, an abstract symmetric multiprocessor (SMP). The SMT machine simplifies the code-generator, by providing special mechanisms for parallelism, and improves its portability, by isolating it from any particular SMP. Unlike previous compilers for similar languages, the SMT code generated by pHc makes use of suspensive threads for better use of sequential processors. Second, the compiler transforms SMT instructions into executable code. In keeping with the goals of simplicity and portability, the final product is emitted as a C program, with some extensions for compiler-controlled multithreading. This code is linked with a Run-Time System (RTS) that provides two facilities: a work-stealing scheduler and a memory manager with garbage-collection.

To evaluate the performance of the code, we generated instrumented programs. Using a small but varied set of benchmarks, we measured the components of run-time, the instruction mix, the scalability of code up to eight processors, and as a comparison, the single-processor performance against two of the latest Haskell compilers. We found encouraging speedups up to four processors, but rarely beyond. Moreover, we found substantial overhead in single-processor performance. These results are consistent with our emphasis on completeness and correctness first and on performance second. We conclude with some suggestions for future work that will remedy some of the shortcomings discovered in the evaluation.

Thesis Supervisor: Arvind
Title: Johnson Professor of Computer Science
Acknowledgments

I have been part of the Computation Structures Group for ten years, and in that decade, I have gone from queasy undergraduate to confident, some would say hardened, graduate. I cannot imagine time better spent. Through it all, Arvind has been the steady helm of CSG, and from his insights and his approach to hard questions, I have learned the course to good research. It follows a deceptively simple bearing: choose your work because you find it worthwhile, not because it is the fashion, and do it until you think it complete, not until others happen to judge it so. None of us can predict the long-term impact of our research, but work of high quality will not easily founder.

CSG originally attracted me because its dataflow projects involved a rethinking of computing from top to bottom. In Monsoon and StarT, we did it all, from programming language theory to high-performance hardware design. But none of this would have been possible without a superb crew. Together we tackled problems that were big, tough, unyielding, and we persevered in the face of more obstacles than we care to remember. I was especially lucky to have worked closely with several generations, in particular Shail Aditya, Boon Ang, Andy Boughton, Derek Chiou, James Hoe, Jamie Hicks, Chris Joerg, R. Paul Johnson, Jan-Willem Maessen, and Ken Traub.

I was fortunate too in my choice of thesis committee. In Martin Rinard and Vivek Sarkar, I found not only two experts in parallel compilers, which is rare enough, but also two enthusiastic and approachable colleagues, which was a real treat. This dissertation was markedly improved by their keen observations.

Finally, as this long journey comes to an end, my thoughts turn to my family. Their unwavering faith in its successful completion stayed mine during some crushing ebbs. It is because of them that I never felt alone, and it is to them that I dedicate this work.
A mi familia,
con mucho cariño.
Contents

1 Introduction .......................... 15
   1.1 pH: Parallel Programming Made Easy ............................................ 15
   1.2 Complications ................................................................. 16
   1.3 Contributions ........................................................................ 17
   1.4 Dissertation Overview .......................................................... 18

2 Parallel Haskell ................................ 19
   2.1 Origins and Design ................................................................. 19
   2.2 Type System ........................................................................... 22
   2.3 Higher-Order Functions .......................................................... 24
   2.4 Implicit Parallelism ................................................................. 26
   2.5 Lenient Evaluation ................................................................. 27
   2.6 Synchronizing Memory ............................................................. 30
   2.7 Barriers .................................................................................. 33
   2.8 The λ*-Calculus ...................................................................... 35
   2.9 Summary ................................................................................ 38

3 Compilation Challenges ................................ 41
   3.1 Polymorphism ........................................................................... 41
   3.2 Higher Order Functions ........................................................... 44
   3.3 Non-Strictness .......................................................................... 49
   3.4 Synchronizing Memory .............................................................. 55
   3.5 Barriers ................................................................................... 57
   3.6 Garbage Collection .................................................................... 58
   3.7 Summary ................................................................................ 59
# List of Figures

2-1 The Layered Design of $pH$ ........................................ 20  
2-2 Parallelism at All Levels in $pH$ ................................. 27  
2-3 I- and M-Structures ................................................ 31  
2-4 Barrier Synchronization ........................................... 35  
2-5 Syntax of the $\lambda^*$-Calculus ................................ 36  
3-1 Optimizing Higher-Order Functions via Inlining ............... 47  
3-2 Examples of Lambda-Lifting ...................................... 49  
4-1 The Shared-Memory Threaded (SMT) Machine .................... 62  
4-2 SMT Instruction Set Summary .................................... 64  
4-3 RTL Notation for SMT Instruction Descriptions ................ 64  
6-1 Original vs. Threaded Code for the Inner Letrec of $\text{fib}$ .... 122  
6-2 Codeblocks: A Control-Flow Abstraction ...................... 124  
6-3 Codeblock for Threaded $\text{fib}$ ................................ 125  
6-4 Algorithm to Find Defined Variables in Threads .......... 126  
6-5 Algorithm to Assign Task Structures to Threads .......... 129  
6-6 Threading Using the $\text{Always}After$ Map ...................... 138  
7-1 SMT Virtual Processor Implementation ......................... 148  
7-2 Work-Stealing Code in the Run-Time System .................. 153  
7-3 Local and External Access to SVP Work Queues ............... 155  
7-4 Traditional Tag Encoding ...................................... 156  
7-5 NaN Tag Encoding ............................................. 158  
7-6 Examples of Tagged Data ...................................... 159
Chapter 1

Introduction

This dissertation presents the design, implementation, and evaluation of pHc, a new compiler for Parallel Haskell (pH). The pH language blends the powerful features of a high-level, higher-order language with implicitly parallel, non-strict semantics. The product is a language that is easy to use but tough to compile.

1.1 pH : Parallel Programming Made Easy

A major theme in the work of the Computation Structures Group at MIT has been the development of general-purpose implicitly-parallel programming languages like Id [41] and pH [42]. pH was designed with great care so that a compiler can automatically extract all available parallelism from a program without intervention from the programmer. Parallelizing a program by hand, especially one with complex structure and data dependences, is a notoriously difficult task, so this property of pH is particularly attractive.

In contrast, efficient parallel programming systems in wide use today, such as PVM or MPI, follow a different design philosophy. These systems give low-level control of parallelism and machine resources to the programmer. With enough time and patience, it is possible to write fast, correct programs using PVM or MPI. But these pale besides pH programs when compared on the basis of portability, maintainability, elegance, and ease of development. Languages like Cilk [12] occupy the space between the two extremes. On the one hand, the Cilk programmer is still responsible for parallelizing the program explicitly, and that can be a heavy task. But on the other, the type of parallelism offered by Cilk is restricted to make compilation and implementation straightforward.
The tension between performance and productivity is not new. Early parallel machines required ultra-efficient programs to improve their poor cost/performance ratios. But the advent of symmetric multiprocessors (SMPs) based on commodity CPUs has made parallel machines common, if not downright cheap. This change forces a rethinking of the goals of parallel programming. The need for performance at all costs will give way to an emphasis on programmability, as large numbers of programmers become interested in taking advantage of parallelism within a flexible programming environment. The skeptical reader should note that a similar shift in priorities—productivity before performance—has certainly taken place in the uniprocessor arena, as evidenced by the popularity of Java and Visual Basic.

1.2 Complications

The assumption underlying the design of pH is that programmers are more valuable than CPU cycles. pH allows programmers to write correct parallel programs quickly by placing the burden of parallelization squarely on the compiler. Unfortunately, current pH compilers cannot generate code that even comes close to the efficiency of "hand-coded" PVM, MPI, or Cilk programs, so the productivity gains offered by pH are often overwhelmed by the performance penalties.

The non-strict, implicitly parallel semantics of the language allow the compiler to extract all available parallelism from a program. This property is useful when programs are executed on machines like Monsoon [45], which have built-in support for fine-grain threads. Standard processors, however, do not have the fast, low-latency synchronization and communication mechanisms to make them suitable for executing fine-grain parallel programs directly. Rather, the compilation strategy for pH on stock processors is tricky: pH programs must be chopped up carefully into sequential threads while preserving the correct semantics of the language. Synchronization and communication costs are better amortized as instructions are grouped into ever-larger threads.

In addition to improving parallelism, non-strictness also increases the expressive power of pH, but it introduces a host of problems for the compiler. Non-strictness forces many scheduling decisions to be performed dynamically at run-time, rather than statically at compile-time. Groups of instructions that could otherwise be combined into a single sequential thread often must be split into several threads to preserve the proper behavior of
many programs. Compiler writers know problems are solved most efficiently at compile-time, but ignoring non-strictness would lead to incorrect execution of pH programs.

An additional headache for the pH compiler involves data structures. Data structures in pH are implemented using synchronizing memory in the form of I-Structures or M-Structures. I-Structures provide single-assignment semantics and M-Structures provide mutual-exclusion semantics at the granularity of a single memory word. Such fine-grain synchronization makes it easy to write highly unstructured programs exhibiting complex data sharing patterns, such as producer-consumer parallelism, that cannot be determined at compile-time. But this power comes at a heavy price since every heap access in pH must follow a precise protocol to maintain the synchronizing memory invariants. Machines like Monsoon have dedicated hardware for implementing synchronizing memory. Machines based on standard CPUs must implement the synchronization protocol completely in software.

Implicit parallelism, non-strict semantics and synchronizing data structures are all features that make writing parallel programs in pH easy. But because all this drudgery of parallel programming is transferred to the compiler, the problem of generating good code for pH programs on stock machines is a difficult one.

1.3 Contributions

The work described in this dissertation has led to the following contributions:

- A compiler for pH with complete support for all language features.
- A new code-generation algorithm that works directly on $\lambda^*$, an intermediate representation based on the $\lambda$-calculus. The multithreaded code generated by the compiler makes use of suspensive threads, an approach that has been largely ignored since Traub's [69] original work on the subject.
- In joint work with Jan-Willem Maessen, a complete Run-Time System for pH including a work-stealing scheduler and garbage collector.
- A novel tagged data representation that supports the implementation of synchronizing memory and garbage-collection with minimal impact on the range of scalars.
1.4 Dissertation Overview

The next two chapters comprise an introduction to \( pH \). Chapter 2 shows how the different features of the language simplify the process of writing a parallel program. It concludes by introducing the \( \lambda^* \)-calculus, a simplified, de-sugared version of \( pH \) used as the intermediate language of the compiler. Chapter 3 examines the complexities of \( pH \) compilation in detail.

Chapter 4 introduces the Shared-Memory Threaded (SMT) abstract machine. SMT is an idealized symmetric multiprocessor model that insulates the code-generator for the quirks of any particular real machine. Chapter 5 then describes the first part of the code-generator: the compilation of a \( \lambda^* \) program into SMT machine language. The compilation is syntax-directed and is presented using three recursive rules, \( TE \), \( TS \), and \( TP \), for translating expressions, statements, and whole programs respectively.

The code generated in Chapter 5 is inefficient because it creates too many parallel tasks. Chapter 6 discusses partitioning and threading. The goal of partitioning analysis is to coarsen the granularity of parallelism in \( pH \) programs in order to reduce the costs of scheduling and synchronization. The threading transformation takes the results of partitioning to generate more efficient code.

Chapter 7 presents the translation of a program in SMT machine language into executable code. Our compiler uses the C language to emit low-level yet portable programs. This chapter also describes the implementation of the \( pH \) Run-Time System.

Chapter 8 gives measurements of the performance of code generated by the \( pHc \) compiler. Using a small but varied set of benchmarks, we discuss topics like the major components of run time, instructions mixes, and speedups.

Chapter 9 discusses the work of others that most closely influenced the design of \( pHc \). We also discuss alternative approaches to parallelizing the Haskell language.

Chapter 10 presents our conclusions and suggestions for future work. The current compiler encompasses a tremendous design and implementation effort, and it will serve as a sound framework for future \( pH \) research.
Chapter 2

Parallel Haskell

Parallel Haskell ($pH$) occupies a unique position in the space of programming languages, and its features and semantics are likely to be unfamiliar to many. We begin with an overview of $pH$ and its principal features. Our intention is to give readers a feel for the language, not to provide a formal description. Those interested in more details can consult Nikhil and Arvind's book on $pH$ [44]. We conclude with a description of the $\lambda^*$-calculus, a formal system with the power of the full language but with a much more concise syntax. The $\lambda^*$-calculus captures the essence of $pH$, and it is one of the cornerstones of our compilation strategy.

2.1 Origins and Design

$pH$ traces its origins to Id [41] and Haskell [46]. Its semantics and execution model come from Id, but its syntax and type system are based on those of Haskell. The goal of $pH$ is to serve as a bridge between these two language communities. Haskell programmers should be able to transition to parallel programming without sacrificing any of the high-level language features they depend on and without having to learn new syntax. In turn, Id researchers should be able to expose their work in parallel programming to a large audience of developers.

Id programs can be translated automatically into $pH$ programs with no changes in their behavior. The translation is made possible by the shared semantics and by the fact that the $pH$ type system is an extension of the simpler Id type system. Syntactically, all Haskell programs are legal $pH$ programs. However, differences in the semantics of the two languages,
explained below, may lead some Haskell programs that rely on lazy evaluation to fail to produce an answer or to produce an error. A Haskell program will not terminate with an incorrect result when compiled as a pH program.

pH was designed for general purpose parallel programming. The language offers a flexible multithreaded execution model that has been shown (in the context of Id) to be useful in writing scientific and symbolic applications [26, 60]. The features of pH are divided into three layers shown in Figure 2-1: the functional layer, the I-Structure layer, and the M-Structure layer. Each layer adds more expressive power to the language at the cost of more complex semantics. At its core, pH is a purely functional language that closely resembles Haskell. This layer of the language contains no operations for side-effects, and program execution is deterministic even on multiple processors. Programs written in this layer of the language are referentially transparent so sub-expressions with like values can be freely substituted without changing the value of the enclosing expression. Referential transparency makes functional languages amenable to optimization and automatic parallelization since the independence of expressions is easy to determine.

However attractive the pure semantics of functional languages, it is a fact of life that
side-effects are useful in writing certain algorithms, if only to simplify the incremental computation of data structures. The next layer of pH extends the functional core with single-assignment data structures called I-Structures. With the addition of I-Structures, the language ceases to be referentially transparent. In particular, the operation of allocating an I-Structure is a side-effect. In this layer of the language, similar expressions cannot be substituted always. Such a substitution might cause allocations to be duplicated, and a shared I-Structure might be transformed incorrectly into two independent I-Structures.

Nevertheless, programs written with I-Structures are guaranteed to be deterministic. If a program terminates, then the answer it produces is guaranteed to be unique for the given set of inputs. If a program produces an error (possibly after it returns an answer), all execution paths are guaranteed to produce the same error, or more generally, the same set of errors. On this point, however, the choice is left up to the implementation: it may choose to wait for all the errors to occur, but more commonly, the program will be terminated on the first error.

The final extension to pH consists of M-Structures, imperative, synchronizing data structures. Programs that use M-Structures are not always deterministic as are programs that stay within the functional and I-Structure layers of the language. Non-determinism makes debugging extremely difficult because the same program with the same inputs can produce different results on different runs due to changes in the run-time scheduling of tasks. It would seem that M-Structures are a poor addition to a language that is supposed to simplify parallel programming, but in fact, they are tremendously useful. Not surprisingly, M-Structures can be used to make programs more storage efficient because memory can be updated in place. What is a surprise, however, is that certain programs that use M-Structures can be more parallel and their specification more declarative than versions without M-Structures. Some systems programs, like memory managers, cannot even be written without M-Structures. In practice, M-Structures comprise a small fraction of the data structures in a program, so problems with non-determinism are often easy to isolate.

The original compilation targets for Id, the predecessor of pH, were the TTDA [8] and Monsoon [45] dataflow processors. These machines included hardware support for fine-grained parallelism down to the instruction level. Monsoon, for example, could synchronize two activities in a single cycle, and it could send a message on its network interface in a single clock cycle. To exploit this capability, Id was designed to expose all the parallelism
available in a program. The only limits to parallel execution were true data dependences.

2.2 Type System

Haskell and pH are typeful\(^1\) programming languages. Types are central to pH programming, and the programmer can use the capabilities of the type system to simplify the development process and to improve the robustness of his code. The Haskell type system that pH has inherited is an extended Hindley-Milner type system [17] with the following features:

Static Typing Programs are typechecked at compile-time. It is guaranteed that a type-correct program will not exhibit type errors at run-time.

Type Inference With some exceptions, the type system can infer the types of all expressions in a program automatically, without any declarations from the programmer. The exceptions occur due to the use of overloaded operators.

Polymorphism pH functions can be written to operate over arguments that can assume any type, and data structures can be instantiated to contain elements of any type. Polymorphism is particularly useful in combination with data structures like lists and arrays. We give examples below.

Overloading By extending the Hindley-Milner system with type classes, pH supports type checking of programs that use overloaded identifiers (or ad-hoc polymorphism). An overloaded identifier is one whose meaning is different according to context. Most languages limit overloading to identifiers like arithmetic operators (+, *, etc.), to represent both integer and floating point arithmetic with the same symbol. Type classes give the Haskell programmer the ability to create new overloaded identifiers.

Higher-Order Functions are first-class values. They can be created, passed as parameters, returned as results, and stored in data structures.

The user can define new types in pH, as in the following example of binary trees:

\[
data Tree a = Leaf a \mid Node (Tree a) (Tree a)
\]

The algebraic type Tree has two data constructors, Leaf and Node. The data constructors are used to create new values of the type. The type Tree is particularly interesting because

\(^1\)A term due to Cardelli [14].
it is recursive and polymorphic. It is recursive because the contents of a Node are themselves values of type Tree. It is polymorphic because it can be used to create trees that contain values of any type. The identifier Tree is itself a type constructor, parameterized by the type variable a. When applied to any type, as in Tree Int, the type constructor is used to create new types. In the case of Tree Int, the new type is a binary tree containing integers at the leaves. The expression Leaf 15 uses one of the data constructors to create a value of type Tree Int.

The type system also allows us to define polymorphic functions. Here is a trivial example:

```haskell
identity :: a -> a
identity x = x
```

The first line of the example is an optional type signature for identity. A signature is not necessary to define the function because it is almost certain that the type inference algorithm can deduce the type automatically. Still, it is good programming style to provide one. The signature states that identity takes a single argument of any type, represented by the type variable a, and returns a result of that same type. Though omitted by convention in the type signature, the use of type variables implies universal quantification: \( \forall a, \text{identity} :: a \rightarrow a. \)

The second line is the code for the function. We see that identity simply returns its argument as the result. Applications of the function give the expected results (we use \( \Rightarrow \) informally to mean “evaluates to”):

- \( \text{identity 15} \Rightarrow 15 \)
- \( \text{identity 2.0} \Rightarrow 2.0 \)
- \( \text{identity "hello"} \Rightarrow "hello" \)

Polymorphic functions derive their power from the fact that a single definition can be used for many different types, leading to very concise programs. To be precise, the type signature of a polymorphic function can instantiated to many different types. The code generated for the body of the function can be shared by all instantiations. This power is difficult to appreciate in a function as simple as identity, but it is interesting to note that even such a simple function cannot be properly typed in a monomorphic language like C, without resorting to unsafe type casts. Monomorphic languages can assign only a single type to

---

2 Unless otherwise noted, the terms function and procedure are used interchangeably, even for routines containing side-effects.
2.3 Higher-Order Functions

Another powerful feature of pH is its support for higher-order functions. Functions are first-class values in pH, so they can be created, passed as arguments to other functions, returned as results, and stored in data structures. Suppose we want to implement function composition: given two functions \( f \) and \( g \), we want to create a third function that computes \( f(g(x)) \). The pH function \texttt{compose} does exactly that:

\[
\texttt{compose \ f \ g = \ \lambda \ x \to f \ (g \ x)}
\]

The \texttt{compose} function creates a new "anonymous" function using the `\` pH construct. The `\` is typographical approximation for \( \lambda \)-abstraction. The new function takes a single argument \( x \), applies \( g \) to it and then applies \( f \) to the result.

What is the type signature of \texttt{compose}? It is:

\[
\texttt{compose :: (a \to b) \to (c \to a) \to (c \to b)}
\]

This type signature seems complicated, but it can be broken down into three parts. The \texttt{compose} function takes two arguments and returns one result. The type expression \( a \to b \) is the type signature of the first argument of \texttt{compose}, parameter \( f \). It represents the type of any function, since the type variables can be replaced with any actual type. The type expression for the second argument \( (g) \) is \( c \to a \). It can also represent any function, but we have introduced a constraint by repeating the use of type variable \( a \). We have stated that the result produced by \( g \) must be of the same type as the input to \( f \). The final type expression describes the result of \texttt{compose}. It is a function whose input argument has the same type as the input to \( g \) and whose output has the same type as the result of \( f \), exactly the purpose of function composition. A function like \texttt{compose} is higher-order because it takes functions as inputs or returns a function as a result. Moreover, \texttt{compose} is polymorphic because it can operate on an infinite number of input function types as long as the constraints expressed in the type signature are satisfied.

Though languages like C and Pascal allow functions (more accurately "function pointers") to be passed as arguments, stored in data structures, and returned as results, they do not have a construct equivalent to `\` in pH. This means that new functions cannot be created at run-time, so these languages cannot be categorized properly as higher-order.
Another useful higher-order function is the `map` function on lists (analogous to `mapcar` in Lisp):

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x : xs) = (f x) : (map f xs)
```

The type signature indicates that `map` takes a function as its first argument, a list as its second, and produces a list as its result. The type signature includes two constraints. The elements of the argument list must be of the same type as the input to the argument function `f`, and the elements of the result list must be of the same type as the result of `f`.

This definition of `map` is given in two lines and uses pattern-matching. The definition of the list type (not shown here) creates two constructors: `[]` (read as “nil”) and `:` (used as infix and read as “cons”). If the input to `map` is the empty list, then `map` returns the empty list. If the input is a non-empty list, then `map` creates a new list cell whose head is the application of function `f` to the first element of the input list and whose tail is the recursive call of `map` on the tail of the input list. For example (note that `[x,y,z]` is shorthand for `x:(y:(z:[])))`:

```
map sqrt [1.0,4.0,9.0] => [1.0,2.0,3.0]
```

Like `compose`, `map` is both higher-order and polymorphic.

Finally, `pH` also supports currying, a shorthand way to create new functions via partial application of existing ones. For example, `double` is a one argument function that doubles its input. It is created from a two argument multiplier function `mult` by currying `mult` over a single argument, 2:

```
mult :: Int -> Int -> Int
mult x y = x * y
double :: Int -> Int
double = mult 2
```

```
double 4 => 8
map double [1,4,9] => [2,8,18]
```

Languages like Lisp [64], Scheme [52], and ML [40] have long supported higher-order functions because they help programmers write concise, robust programs. Programs are concise because higher-order functions, like `map`, can be used to capture common “patterns” of computation. It’s easy to specialize one of these generic patterns for a specific purpose: a subcomputation is simply passed in as a function argument. Programs become
robust because they can be composed out of small, well defined pieces. When using \texttt{map}, programmers need not worry about the details of iteration through lists. Abstractly, every list argument will be processed. Instead, programmers can concentrate on the smaller problem of the operation to be applied to each element.

2.4 Implicit Parallelism

We consider \texttt{pH} an implicitly parallel language because its semantics allow a compiler to uncover all parallelism in a program automatically, without any help from the programmer. The only limits on parallel execution are data dependences and the behavior of three control constructs: conditionals, \texttt{\lambda}-abstractions, and barriers. If data dependences cannot be uncovered statically by analyzing the program, the language expects them to be detected at run-time, often through the use of synchronizing memory (Section 2.6).

The following \texttt{pH} function determines if a triangle is a right triangle given the lengths of its three sides. A note about syntax: \texttt{pH} supports the \textit{off-side} rule so that indentation replaces braces (\{ \}) to delimit scopes.

\begin{verbatim}
rightTri :: Int -> Int -> Int -> Bool
rightTri a b c =
    let result = cc == aa + bb
        aa = a * a
        bb = b * b
        cc = c * c
    in result
\end{verbatim}

The bindings in the \texttt{let} block do not execute in the usual top-to-bottom, left-to-right order of other languages. Rather, they execute in an order that respects data dependences. The binding for \texttt{result} will execute after the bindings for \texttt{aa}, \texttt{bb}, and \texttt{cc}. These three bindings, in turn, can execute in any order, in series or in parallel, since they are independent of each other.

Standard sequential programming is hard enough, and parallel programming adds a whole new dimension of complexity to the software development process. In explicitly parallel programming languages, the programmer is responsible for identifying tasks that can execute concurrently, and often, is also responsible for scheduling tasks and for ensuring that shared data is accessed correctly. Implicit parallelism in \texttt{pH} vastly simplifies parallel programming because it places the burden of finding and exploiting parallelism squarely on
Implicit parallelism also allows the programmer to write more reliable programs and more portable programs. Programs are more reliable because the compiler manages notoriously tricky tasks like synchronization and scheduling. These are often the sources of extremely complex, non-deterministic bugs in explicitly parallel programs. Programs are more portable because they are not written to a particular machine configuration or network topology. Rather, they are written for an abstract, shared-memory, rewriting-based execution model that can exploit parallelism at every level of a program, as shown in Figure 2-2.

2.5 Lenient Evaluation

The semantics of pH are non-strict. This choice affects the language in three ways. First, the arguments of a procedure can be computed in parallel with the body of the procedure. Second, the result of a procedure can be returned immediately after it has been computed, even before all computation has ceased in the body, or in extreme cases before all arguments have been computed. Third, the allocation of a data structure and the computation of its
contents are decoupled. This means that partially computed data structures can be passed around the program and permits concurrent execution of both producers and consumers of data in a program. Non-strictness also makes it easy to create cyclic data structures.

The most common non-strict languages, like Haskell, follow a lazy evaluation strategy. A lazy language guarantees that only those expressions required to produce the final result of a program are evaluated. In theory, the lazy evaluation strategy will find an answer in the least number of abstract computational steps. In practice, lazy languages make it convenient to write programs that use conceptually "infinite" data structures. Such programs terminate because they need only a finite portion of the infinite data structure to compute a result. The lazy language implementation guarantees that exactly that portion, not the entire structure, is computed during the execution of the program.

$pH$ is not a lazy language but instead follows an eager evaluation strategy. Every expression that can be evaluated in a $pH$ program will be evaluated, even if it does not contribute to the final result. This means that some programs will execute forever in $pH$ but will terminate in Haskell. For example:

```haskell
forever :: Int -> Int
forever x = forever x

done :: Int -> String
done y = "done"

done (forever 5) ⟷ "done"  in Haskell
done (forever 5) ⟷ ???  in pH
```

The invocation of `done` terminates in Haskell because the result of the function, the string "done", does not depend on the argument to the function. The Haskell program never evaluates the infinite computation (`forever 5`). The $pH$ program, on the other hand, will evaluate the argument, so it will execute forever. Depending on the details of the particular $pH$ implementation, the program may still return the value "done". Why? Non-strictness says that functions can return results even before their arguments have been computed. If a $pH$ implementation schedules tasks fairly, so that all are guaranteed to make progress, then the task that returns the result of the function will terminate eventually, even though the task that computes (`forever 5`) and, consequently, the whole program will not.

Except for control constructs, there is never a question as to whether or not an expression should be evaluated in $pH$. This property is particularly useful when reasoning about
parallelism, but this is not the case for lazy languages. A central problem in compiling lazy
programs is limiting evaluation to exactly those expressions that are needed to produce an
answer. As a result, lazy programs tend to follow a highly serial execution path where only
one sub-expression is enabled for evaluation at any point in time.

The combination of non-strict semantics and eager evaluation, referred to as lenient by
Traub [69], is an intermediate step between lazy and strict languages in terms of expressiveness
and efficiency. Lenient languages are more expressive than strict languages but do not
require all the bookkeeping of lazy ones. Consider the following program in pH (adapted
from Traub) to create a doubly-linked list from a normal list. The cells of the doubly-linked
list are built from the Cell type. These are either an EndLink, to mark the beginning or
the end of the doubly-linked list, or a Link with three components: the element of the cell,
the previous cell, and the next cell.

```haskell
data Cell a = EndLink | Link a (Cell a) (Cell a)
makeDoublyLinked :: [a] -> Cell a
makeDoublyLinked [] = EndLink
makeDoublyLinked (x:xs) =
  let dblLink [] prev = EndLink
      dblLink (1:ls) prev =
        let this = Link 1 prev next
            next = dblLink ls this
        in this
      first = Link x EndLink rest
      rest = dblLink xs first
  in first
```
Non-strictness in pH allows the cyclic references in makeDoublyLinked between first and
rest and between this and next in dblLink. Without this sort of reference, the doubly-
linked list cannot be built, but this kind of data dependence cannot be expressed without
side-effects in a strict language like ML.

The lenient semantics of pH also expose the maximum amount of parallelism in any
program. Unlike a strict or lazy language (see Tremblay and Gao [71]), lenient evaluation
does not hinder parallelism in pH, so the compiler is able to discover fine-grain parallelism
directly, without the need for complex program analysis. The compiler does have to imple-
ment dynamic "safety-net" mechanisms for handling scheduling and synchronization that
cannot be detected at compile-time, as we will see in Section 3.3 and Chapter 6. This is an
important implementation challenge on non-dataflow machines.
2.6 Synchronizing Memory

One of the key elements of the pH programming model is a global heap for data structures shared among many tasks. One of the biggest challenges for parallel programmers has been to exploit the memory models offered by multiprocessor machines. Models that programmers find easy to understand, like sequential consistency [35], are not amenable to high-performance implementations. Models that can be implemented efficiently tend to provide weaker invariants than sequential consistency and often require significant effort to incorporate into programs.\(^3\) An effective memory model should be defined by the programming language, not the processor. This is the approach taken in pH with I-Structures and M-Structures, two forms of implicitly synchronizing memory.

2.6.1 I-Structures

I-Structures [9] are single-assignment data structures. During the lifetime of an I-Structure, every element can be defined at most once. As shown in Figure 2-3, a typical implementation divides I-Structure cells into two fields: data and presence. The data field holds the value of the I-Structure cell. On allocation, the contents of this field are undefined. The presence field is used to enforce the single-assignment semantics of the I-Structure cell, and it can take on one of two values, empty or full. On allocation, the presence field is initialized to empty to indicate that the contents of the data field have not been computed.

Two basic operations are defined on I-Structures, IStore and IFetch. These have the following type signatures, in pseudo-pH syntax:

\[
\begin{align*}
\text{IFetch} & : \text{ Address } \rightarrow \text{ Value } \\
\text{IStore} & : \text{ Address } \rightarrow \text{ Value } \rightarrow \text{ Void }
\end{align*}
\]

Both operations execute atomically with respect to the cell they access.

An IFetch operation on an empty I-Structure cell is said to defer. The task that issued the operation suspends and waits for the value of the cell to be computed. The task is placed on a deferred list associated with the cell. The deferred list identifies all tasks waiting for the value of the cell to be computed. In contrast, an IFetch operation on a full cell does not defer. It retrieves the contents of the data field and leaves both the presence and data fields unchanged.

\(^3\)See Frigo [19] for an interesting discussion of this issue as well as the weakest reasonable memory model.
IFetch 0 ⇒ 17.0
IStore 1 2.78 ⇒ side-effect only
IFetch 2 ⇒ task deferred
MFetch 3 ⇒ 8.69
MStore 4 3.14 ⇒ side-effect + task, enabled

<table>
<thead>
<tr>
<th>Memory State Before Operations</th>
<th>Memory State After Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 full 17.0</td>
<td>0 full 17.0</td>
</tr>
<tr>
<td>1 empty</td>
<td>1 full 2.78</td>
</tr>
<tr>
<td>2 empty</td>
<td>2 empty</td>
</tr>
<tr>
<td>3 full 8.69</td>
<td>3 empty</td>
</tr>
<tr>
<td>4 empty</td>
<td>4 empty</td>
</tr>
</tbody>
</table>

Figure 2-3: I- and M-Structures

An IStore operation on an empty I-Structure cell has two effects: the presence field changes to full and the contents of the data field are replaced with the Value parameter of the IStore. If the cell has a deferred list, those tasks are provided the new value of the cell and are restarted. An IStore operation on a full cell generates a “double store” error since it violates the single-assignment invariant of I-Structures.

The I-Structure protocol implemented by the IFetch and IStore instructions provides implicit, fine-grain synchronization for parallel tasks that share data structures. The pH programmer never needs to worry about coherence, atomicity, or any of the other complications associated with concurrent access to shared memory. The protocol specifies the behavior of each I-Structure cell independently and unambiguously. Furthermore, the single-assignment semantics of I-Structures guarantee that pH programs remain deterministic.

2.6.2 M-Structures

While I-Structures provide a great deal of functionality and account for the majority of data structures in pH programs, some applications require truly mutable state. The language provides M-Structures [10] for this purpose. The value of an M-Structure cell can change
many times during its lifetime, but the M-Structure protocol guarantees mutual exclusion between tasks accessing a particular cell. The protocol implicitly synchronizes any number of tasks accessing an M-Structure cell to prevent data races.

Freshly allocated M-Structures look a lot like I-Structures. The presence field of each cell is set to empty and the data field is undefined. Two basic operations are defined for M-Structures:

\[
\begin{align*}
\text{MFetch} & : \text{Address} \rightarrow \text{Value} \\
\text{MStore} & : \text{Address} \rightarrow \text{Value} \rightarrow \text{Void}
\end{align*}
\]

An MFetch operation on an empty M-Structure cell defers, just like an IFetch on an empty I-Structure. An MFetch on a full cell, however, changes the presence field to empty and returns the value of the data field. In this way, it forces future MFetch operations to wait for a new value of the cell to be written back.

An MStore to an empty cell first checks for the existence of a deferred list. If there is no deferred list, the MStore changes the presence field to full and updates the data field with the new value. If a defer list does exist, the MStore extracts only one task from the list, provides it with the new value, and resumes it. All other tasks remain deferred, and the presence and data fields of the cell remain unchanged. In this way, the MStore guarantees that only one task at time controls the value of the M-Structure cell. Notice that we leave unspecified exactly which task gets resumed on an MStore. It is up to the implementation to make this choice. A good implementation should strive for fairness, so that no particular reader is ever deferred indefinitely. Not surprisingly, an MStore to a full cell generates an error. It violates the mutual exclusion invariant that pairs each MFetch with an MStore.

It is important to note that the data structures in the core functional layer of pH are built from I-Structures and M-Structures. The I-Structure and M-Structure extension layers simply expose the mutable interfaces to data structures, while the core layer keeps them carefully hidden under a functional façade.

### 2.6.3 Producer-Consumer Parallelism

The non-strictness and implicit synchronization of I-Structures and M-Structures allow pH programs to exploit producer-consumer parallelism trivially. Suppose you are formatting your dissertation using the following pH functions:
latex :: File -> DVItree
dvips :: DVItree -> ()
mkDoc :: File -> ()
mkDoc f = dvips (latex f)

The first pass converts a \LaTeX\ input file into a device-independent (DVI) page description. The second pass generates PostScript output from the DVI description and prints the results.

Normally, the downstream pass, \texttt{dvips}, waits for its preceding pass to produce all its output before executing. But in \texttt{pH}, a pass can execute immediately as parts of the input data structure become available. Even a complex data structure like a \texttt{DVItree} can be produced and consumed in parallel. It is built out of basic I-Structure and M-Structure building blocks, so \texttt{pH} guarantees that parallel accesses to it will be synchronized. If the downstream pass gets ahead of the upstream pass, it will defer until the required data is available.

\section*{2.7 Barriers}

The execution model of \texttt{pH} is parallel by default. However, it is often useful to \textit{sequentialize} computations without introducing data dependences. \texttt{pH} provides barriers with the \texttt{seq} construct for this purpose. Conceptually, a barrier divides a computation into two regions, the \textit{pre-region} before the barrier and the \textit{post-region} after the barrier. The purpose of the barrier is to prevent execution from proceeding to the post-region until the pre-region terminates.

In most programming languages, barriers have global effect. Their scope extends to all parallel activities (threads, tasks, processes) in a program. Barriers are usually used for correctness, to divide programs into separate phases with well defined invariants, but they can also be used to improve performance as shown by Kuszmaul [34]. Barriers in \texttt{pH} are different because their scope can be controlled precisely to extend only to as many tasks as necessary. Suppose you want to print the coordinates of a point using the format \texttt{"(x,y)"} (below, \texttt{x} evaluates to 50 and \texttt{y} to 100). The values of \texttt{x} and \texttt{y} can be computed independently, so you write the following code:
let x = a big computation
    print "("
    print (asString x)
    print ","
    print (asString y)
    print ")"
    y = another big computation
...

In pH each statement in this let (letrec) block can execute in parallel if possible. Even if the computation of x terminates before the computation of y, the program might generate the output "),(10050", hardly the desired result. The real problem is that the print operation only performs a side-effect and does not return a value. Thus, data dependences cannot be used to sequentialize a series of print calls. The solution is to use seq:

let x = a big computation
    seq print "("
        print (asString x)
        print ","
        print (asString y)
        print ")"
    y = another big computation
...

The seq block forces the five statements within it to execute in serial order, but only those statements are affected. The entire seq block as a unit can execute in parallel with the computations of x and y. For instance, if x terminates before y, the printing of x can execute concurrently with the computation of y. The seq does not alter the lexical level of the statements it encompasses. These remain at the same lexical level as x and y.

A barrier provides a powerful way to express bulk synchronization. All computation in the pre-region will have terminated before any computation in the post-region executes. In Figure 2-4, we concentrate on a single barrier region. The pre-region consists of the statements S_1...S_3 while the post-region corresponds to statements S_4 and S_5. Statements S_1...S_3 can execute in parallel, but due to the barrier, the compiler will emit code that will wait for all the tasks they create to terminate before it schedules S_4 and S_5. The barrier guarantee holds regardless of the complexity of the computation in the pre-region. For instance, S_1, S_2, and S_3 might each invoke a different recursive function which might
lead to a deep call tree and lots of parallelism. The barrier synchronization applies to tasks created in function calls too, and they must all terminate before the post-region executes. This is exactly the same kind of guarantee provided by the `sync` statement in Cilk.

Barriers are rare in most pH code. Their presence is closely tied to the use of M-Structures or I/O. However, as we'll see in Chapter 5, the implementation of barrier synchronization has a pervasive influence on the code generation strategy for pH.

### 2.8 The λ*-Calculus

The previous sections have touched on the major features of pH. The full language combines a complex semantics with a rich syntax, making compilation a real challenge. To simplify the problem, we use the λ*-calculus, a syntactically simple language that captures the essence of pH. The compiler is designed so that only the parser and type-checking phases process the full source language. These are followed by a “desugaring” phase that converts a pH program into an equivalent λ* program. None of the power of pH is lost in the conversion, but the more restricted syntax of λ* vastly simplifies the structure of the rest of the compiler.
The $\lambda^*$-calculus is a minor modification of the $\lambda_S$-calculus used by Arvind et al. [7] to give a precise operational semantics for $pH$. It is based on the call-by-need $\lambda$-calculus and includes all the unique features of $pH$: parallel execution, I- and M-Structures, and barriers. Figure 2-5 presents the syntax of the $\lambda^*$-calculus.

Notice that instead of the off-side rule, $\lambda^*$ requires explicit braces to delimit scopes, and separators, like ‘;’, ‘—’, and ‘~’, to separate statements. The syntax of the calculus also restricts where full-fledged expressions may appear: only as the bodies of abstractions, in the arms of Case expressions, and in the right-hand sides of bindings. Everywhere else, only
identifier expressions can appear. This restriction does not curtail the power of the language, and it helps to simplify the compilation process. Finally, we assume that parentheses can be used to group non-terminals like $S$ and $E$.

As an example, the $pH$ expression in the previous section to print coordinates of a point is written in $\lambda^*$ as follows:

```lambda
{ 
  x = a big computation ;
  t1 = "("
  _ = print t1
  t2 = asString x
  _ = print t2
  t3 = ",
  _ = print t3
  t4 = asString y
  _ = print t4
  t1 = ")"
  _ = print t1
  y = another big computation ;

  ...

  in ... }
```

The $\lambda^*$ version of the expression includes additional variables (t1-t4) to name intermediate computations. Also, it uses the syntax `_ = ' for binding statements where the result of an expression need not be bound to an identifier. These statements usually result from expressions executed only for side-effects.

The $\lambda^*$-calculus makes two extensions to the original $\lambda_S$. First, $\lambda$-abstractions and application expressions can take multiple arguments. The original calculus only supported single-argument functions, but multiple arguments are needed to enable optimizations of currying. The second extension is the `$\sim$' syntax to group statements. The `$\sim$' indicates that the second statement (rightmost in Figure 2-5) does not need to execute in parallel with the first. In fact, the `$\sim$' notation asserts that the second statement can always execute after the first, regardless of how the procedure is called. The `$\sim$' statement conjunctions are inserted during the partitioning and threading phases of code generation. We discuss these in detail in Chapter 6.

Neither of these two changes affects the semantics of the original $\lambda_S$ calculus: multiple-argument functions behave exactly as nested single-argument functions, and the `$\sim$' notation can be treated just like `;'. Beyond these, the differences between the two calculi are trivial.
We give a summary of the semantics of $\lambda S$ in Appendix A.

The use of $\lambda^*$ as an intermediate program representation is unique to our $pH$ compiler. Previous $pH$ (and Id) compilers use dataflow program graphs (DFPGs) [68] rather than $\lambda$-calculus expressions as the intermediate representation. This is a legacy of the time when Id and $pH$ were compiled for dataflow machines like TTDA and Monsoon. We had three motivations for breaking the dependence of the compiler on dataflow graphs.

First, we wanted the intermediate form of the compiler to follow $\lambda S$, the framework used to give semantics to $pH$. Understanding of $pH$ has evolved significantly since the days when its behavior was directly linked to operations on dataflow graphs. Today, with the $\lambda S$ formalism, the semantics of $pH$ are still operational, but they are more abstract and much more precise. Indeed, important, though subtle, problems with the semantics of barriers were fixed only when $\lambda S$ was adopted. A close correspondence between the theoretical $\lambda S$ work and the language implementation gives us a better basis for writing a correct compiler.

Second, by using a representation more like the $\lambda$-calculus we can leverage from and contribute to the work of the higher-order, typed language community. The current front-end of the pHc compiler, for instance, was contributed by Lennart Augustsson and can do double duty as a Haskell front-end. Many of the high-level optimization passes applicable to Haskell, like deforestation [38], are also applicable to $pH$ because the two languages share a common functional core.

Third, the $\lambda^*$ representation seems well-suited to expressing threaded computations. Our compiler emits code for SMPs composed of standard von Neumann processors, so the quality of the code depends on how well we translate the implicitly parallel $pH$ programs into sequential code. Past experience has shown that expressing control flow in dataflow graphs is cumbersome. In contrast, we found that extending $\lambda S$ slightly with the ' $\sim$ ' construct made it easy to think about threading at a high-level.

### 2.9 Summary

The variety of parallel language implementations is surprisingly rich, yet the combination of features in $pH$ is unique. This is perhaps due to the fact that the designers of the language

---

4See Rinard [53, Ch.5] for a general overview of the subject or Hammond [23] for a focused on parallel functional languages.
made few compromises in their quest to make it easy to write correct parallel programs. For the foundation of the language, they chose to emulate existing strongly-typed, higher-order functional languages rather than imperative languages. A flexible type system makes it possible to write concise, well-structured programs, while static typing provides a strong guarantee about the correct run-time behavior of programs.

In the area of semantics, the decision to make the language non-strict, eager, and implicitly parallel places the burden of finding parallelism on the compiler, not the programmer. This relieves a significant amount of the workload involved in writing a correct parallel program. Unlike other languages that emphasize a particular kind of parallelism (e.g., data-parallelism), the semantics of pH also help make the language useful for general-purpose programming because they don't obscure any of the parallelism available in an algorithm, at any granularity.

One of the biggest headaches in parallel programming is coordinating tasks that access data structures in shared memory. Functional languages, like the core of pH, are well-suited for parallel programming precisely because they avoid side-effects. But side-effects can improve the efficiency and the ease of writing certain kinds of algorithms, so pH includes write-once I-Structures and mutable M-Structures. The key feature of I-Structures and M-Structures is that the memory model is defined by the language, not by the programmer nor by the machine. This substrate of memory with automatic synchronization at the granularity of a single memory word is a great aid in the development of programs with shared data structures. To simplify programming with side-effects even more, the language also includes fine-grain barriers that allow precisely-defined regions of a program to be serialized completely.

pH was designed to make parallel programming as natural and easy as sequential programming. The language relieves the programmer from tasks that are tedious or error prone and makes them the responsibility of the compiler. Throughout the rest of this dissertation, we see the implications of these design decisions on the process of generating efficient code for pH source programs.

39
Chapter 3

Compilation Challenges

Writing parallel programs in pH is easy. In fact, most novice programmers do not realize they are writing parallel programs because pH is parallel "by default." It requires explicit syntax to make a program sequential. In the last chapter, we showed how many features of the language help the programmer write concise, robust, and highly parallel programs, but unfortunately, simplicity for the programmer does not come for free. In the design of pH, the cost of a simple programming model is borne by the compiler writer, as the pH compiler must shoulder not just the usual burdens of generating code from a high-level language, but also of exploiting implicit parallelism.

In this chapter, we discuss how specific features of pH make it difficult to generate efficient code. The point we wish to make is simple: to achieve parity in code efficiency with a low-level imperative language like C or its parallel dialects like Cilk, the pH compiler writer must tackle a wide range of problems absent in those less complex languages. Some of these problems have to do with the expressiveness of the language and some with parallelism. Good work exists in the literature that addresses several of the problems discussed in this chapter, but incorporating all of it into a single compiler is a vast undertaking.

3.1 Polymorphism

We have already seen how the polymorphic type system of the language enables the programmer to write "generic" routines that operate over many different types. In general, the rich type system of pH is an asset to both the programmer and the compiler writer. For instance, due to compile-time type checking, code to detect type-errors at run-time does not
have to be generated, as is the case in dynamically-typed languages like Lisp or Smalltalk. So strong typing can improve the efficiency of the generated code. Polymorphism, on the other hand, tends to insert inefficiencies into the code by requiring complete uniformity in the calling conventions and in the data representation for the language.

Recall the identity function in the previous chapter:

\[
\text{identity } x = x
\]

We assume that arguments are passed and returned from functions in registers. In the case of identity which register should we use? The code for identity must work properly for arguments of all types: integers, floating-point numbers, pointers, etc. In general, values of these types cannot be stored in the same registers for two reasons. First, different data types have different bit-widths. Integers might be 32-bits wide while IEEE double-precision floating point values are 64-bits wide. Second, functional units in modern processors operate on different register sets. The integer ALUs might operate on the 32-bit integer register set and the floating-point ALUs might operate on the 64-bit floating-point register set. Transferring values between register sets often requires a memory store and load.

The standard calling conventions for operating systems like IBM AIX [27], which have been tailored to monomorphic languages like C, often require that integers be passed in integer registers and floats in floating-point registers. In a language like C, the prototype (type signature) of a function is enough to determine the correspondence of arguments to registers, but of course, this is not the case in pH.

The problems of polymorphism are not limited only to the function calling conventions. The format of aggregate data structures is also affected. Consider another trivial polymorphic function:

\[
\text{getArrayVal :: Array Int e } \rightarrow \text{ Int } \rightarrow \text{ e}
\]

\[
\text{getArrayVal } a i = a!i
\]

This routine simply fetches a value from an array \( a \) given the index \( i \). The ‘!’ is the infix array indexing operator in pH. Due to the type variable \( e \) in the type signature, getArrayVal is a polymorphic function that works for integer-indexed arrays containing elements of any type. But if different data types require different amounts of memory, how do we calculate the correct memory offset within the array for the element we require?

The simple solution to the problems presented by polymorphism is to use a uniform-width data representation. Many compilers for polymorphic languages choose to represent
all data as fixed 32-bit quantities because these can be handled conveniently in the integer registers of a processor. Data types larger than 32-bits are allocated on the heap ("boxed") and are represented by 32-bit heap pointers. Such a uniform-width data representation incurs extra overhead in both space and time. Obviously, boxed values consume an extra memory location per floating-point number, but even worse is the fact that an array of eight-bit characters consumes 32-bits per element. In addition, extra instructions must be included to box and unbox floating-point values every time they are used in a non-polymorphic way.

3.1.1 Compiling Polymorphism Efficiently

There is a body of interesting work that aims to improve the efficiency of polymorphic languages. Leroy [36] presents a scheme to mix both specialized (such as 64-bits for double-precision floats) and uniform data representations in a strict polymorphic language. In his system, functions which are known to be monomorphic at compile-time always work with specialized data representations. Polymorphic functions on the other hand use a uniform data representation. The problem is reduced to determining when to insert explicit boxing and unboxing code for communicating values between the monomorphic and polymorphic portions of a program. Leroy shows a systematic way to perform these conversions. Henglein and Jørgensen [25] generalize Leroy's work by providing a system to derive optimal solutions to the problem. In their sample programs, their system generally performs fewer boxing/unboxing operations than that of Leroy.

There are substantial differences in the approaches that have been taken between strict and non-strict (lazy) functional languages. Peyton Jones and Launchbury [50] extended the Core language (λ-calculus intermediate representation) of the Glasgow Haskell Compiler to support unboxed values directly. This allows them to describe unboxing optimizations in a well-specified form, not as some ad-hoc pass hidden in the code-generator. On top of this infrastructure, Hall et al. [22] describe a system that allows the programmer to specify, perhaps based on profiling information, the use of unboxed types in certain data structures. Since unboxed types cannot be used with polymorphic functions, their scheme takes care of generating specialized versions of polymorphic functions for each unboxed type declared. This relieves a lot of tedium for the programmer, but is clearly not a completely automatic process.
Why does the work in lazy languages not follow the path of automatic unboxing as in strict languages? As discussed in [22], boxed and unboxed values do not have the same semantics. In particular, unboxed values are strict and can only be used where they will not change the strictness properties of the program. This requires that boxing/unboxing and strictness analysis (see Section 3.3.1) be combined into the same optimization. Hall et al. are dubious of the efficacy of strictness analysis in this context, and so they leave it to the programmer to specify which values should go unboxed.

3.1.2 Approach Taken in pHc

In the implementation of pHc, we chose the expeditious solution of a uniform-width data representation. However, to avoid the overhead of boxing and unboxing, we chose to make all scalars 64-bits wide. Section 7.3 describes our data representation in detail. We pay overhead in terms of increased memory usage and bandwidth, but we achieve a simple implementation. In addition, the uniform-width representation fits well with garbage collection. In the future, we expect pHc will include some of the advanced techniques discussed above, though these will require changes in the middle-end of the compiler.

3.2 Higher Order Functions

Higher-order functions, in conjunction with polymorphism, are essential to writing compact, reusable code. In functions such as map, higher-order functions help programmers abstract control-flow and concentrate on per-element operations rather than on whole data structure traversals. Compiler writers, on the other hand, face several problems when generating code for higher order functions. First, the procedure calling conventions become significantly more complicated because even simple applications can be ambiguous at compile-time. For example, consider the use of parameter g in the following function, and assume the application of g returns an integer (pH type Int):

\[
\lambda g \cdot g \ 1 \ 2 \ 3
\]

To compile an application, we need to know the arity of the function. What is the arity of function g? The answer seems obvious. Since three integer arguments are being passed to g, then g must have arity three. Unfortunately, this reasoning is incorrect. It may come as a shock to discover that type information in a higher-order language does not provide
precise information about the arity of a function. The application of \( g \) above could in fact be satisfied by four kinds of functions with very different types:

1. \( g_1 : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \) A function of arity three that returns an integer.
2. \( g_2 : \text{Int} \rightarrow \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \) A function of arity two that returns a function of arity one that returns an integer.
3. \( g_3 : \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}) \) A function of arity one that returns a function of arity two that returns an integer.
4. \( g_4 : \text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})) \) A function of arity one that returns a function of arity one that returns a function of arity one that returns an integer.

For the purposes of type-checking, these four functions are equally suited to making the application a type-correct expression. For the purposes of code generation, however, these functions do not have the same behavior. An application of \( g_1 \) to three arguments consists of only a single function call, while an application of \( g_4 \) to three arguments consists of a sequence of three separate function calls. The end result of both computations is an integer, but the path to that result is different in each case.

How can the compiler determine at compile-time what kind of function is being passed as parameter \( g \)? The answer is, in general, that it cannot. Instead, the compiler must generate code to determine the arity and entry-point of a function at run-time. It does so by storing this information in the closure data structure representing the function. The extra instructions needed to interrogate the closure at run-time and to perform the correct action depending on the closure's remaining arity are an important part of the overhead of higher-order functions. In Section 5.4.2, we show exactly how we handle these sorts of applications in pHc.

The second and perhaps more challenging aspect of compiling higher-order functions is developing a good representation for closures. The purpose of a closure in a lexically-scoped, higher-order language is to capture the values of the free variables of a function when it is created. When the function is invoked, the closure is used to establish the correct environment for the body of the function to execute. A closure is a conduit for transferring data from the definition to the uses of a function. An efficient representation aims to minimize the number of operations needed to create and access closures as well as the amount of memory used for storing closures. In the general case, a closure must be
allocated on the heap, like any regular data structure, because the lifetime of the function it represents is unbounded.

3.2.1 Compiling Higher-Order Functions Efficiently

There are a variety of optimizations that can be applied to make higher-order function calls more efficient. The goal of these transformations is to remove some of the dynamic checks (of arity, etc.) that would be performed otherwise. The best-known of these transformations is inlining, and we illustrate its effects in Figure 3-1. In the figure, assume zipWith is a library function that takes a function and two lists and applies the function to consecutive elements of both lists producing a new list as a result. The use of \( f \) in the body of zipWith constitutes an application of a higher-order function. The function vectSum in part (a) is defined in terms of zipWith and adds the components of two vectors represented as lists.

The inlining transformation of the program in part (b) of the figure consists of copying the body of zipWith into a new function loop27 and of replacing the higher-order application of \( f \) with '+', the actual function at the call site. The application then becomes a full application of a known function (actually an operator) with no dynamic checks of the closure to be performed at run-time. Note that the original version of zipWith is not necessarily removed from the program unless the compiler can inline all uses of it.

This inlining transformation is essentially the same as the loop-preheader/invariant hoisting combination shown by Appel [4, pp.329-331]. Tarditi [66, p.216] demonstrates the effectiveness of inlining when whole programs are compiled. In his results, higher-order functions are completely removed from a variety of benchmarks with less than a factor of two increase in code size.

Much of the work on improving the performance of languages with higher-order functions has focused on closures. One of the cornerstones of Steele's Rabbit compiler for Scheme [62], a lexically-scoped dialect of Lisp, was a closure analysis phase. In the analysis, Steele identified fully-closed functions which required heap-allocated closures and non-closed functions which did not require closures because their environment was always accessible at all call-sites. Non-closed functions used registers to store the contents of the environment. Rabbit used the continuation-passing style (CPS) program representation, so in fact, all control-flow was expressed as function calls. Hence, the emphasis on compiling lambdas well.
The Orbit compiler of Kranz et al. [32] also expended significant effort in choosing the right closure representation for each function. Long-lived functions were represented on the heap while shorter-lived ones were stored in the stack or in registers. The compiler also included other optimizations like closure hoisting for reducing the memory consumption of closures: when a group of functions used a common set of variables, these were placed into a single run-time structure and were shared by the closures of all functions in the group. These and many other aggressive implementation techniques, like register allocation via trace-scheduling and native code generation, resulted in fast code that was competitive with imperative languages in some benchmarks.

In more recent work, Shao and Appel [58] make use of compile-time analysis and data flow information to make their closure representations both efficient and “safe for space complexity” (SSC). The SSC rule states that “any local variable binding must be unreachable after its last use within its scope.” It addresses a problem that arises with linked closures, closures that contain pointers to the closure of their enclosing function, due to the fact that dead variables within the enclosing function are kept alive by the links from the enclosed closures. Shao and Appel show how the space consumption of a program can jump easily from $O(N)$ to $O(N^2)$ when linked closures are used.

One can view some of the performance penalties of higher-order functions as mani-
festations of a general problem, namely that predicting the control-flow of programs with higher-order functions is difficult. Shivers [61] presents an analysis for Scheme programs that attempts to recover the control-flow graph via abstract interpretation. With the control-flow graph, he shows how to perform standard dataflow analyses like copy propagation on programs that include higher-order functions and side-effects. In the context of a lazy functional languages, Boquist [13] describes aggressive register optimization based on the analysis of Johnsson [29]. This analysis indicates the set of functions that might be invoked at any point in a lazy program, so it serves as an approximation of the control-flow at compile-time.

3.2.2 Approach Taken in pHc

The pHc compiler relies primarily on inlining to optimize higher-order function calls. Common functions in the pH Prelude (the set of standard functions available to all programs) such as map and fold are marked with the {-#INLINE#-} compiler pragma, so that their bodies will be copied directly into any call site. There is a penalty to be paid for overly aggressive inlining in that the size of a program increases. Currently, we believe the benefits of inlining outweigh these costs, though a more discerning inlining algorithm would be useful. The deforestation optimization pass [38] also helps reduce the cost of higher-order functions, but only for those that manipulate lists. Deforestation allows code to be written in the desirable higher-order, compositional style, but it removes the overhead of intermediate lists by turning them into loops or recursive functions.

In the area of closure representations, the compiler utilizes lambda-lifting [28] and flat vectors for curried arguments in partial applications. The essence of lambda-lifting is to convert free variables in a function into regular parameters. Lambda-lifting proceeds in three stages. First, all free variables in a function definition are identified and converted into parameters. Second, the body of the transformed function is lifted to the top-level of the program. It becomes a supercombinator, a function with no free variables. And third, the call-sites for the function are extended to include the extra parameters explicitly. Figure 3-2 illustrates the transformation on several small pH functions.

The great thing about lambda-lifting is that it makes the compiler much simpler. Environments are passed as arguments, and closures are created only by partial applications. In the low-level code-generation, lambda-lifting amounts to passing closures directly in reg-
\[
\begin{align*}
\text{f1}\ a\ b &= \text{let}\ f2\ c\ d = \\
&\quad \text{let}\ f3\ e\ f = a + b + c + d + e + f \\
&\quad \text{in}\ f3\ (4\cdot c)\ (5\cdot d) \\
&\quad \text{in}\ f2\ (2\cdot a)\ (3\cdot b) \\
\end{align*}
\]
(a) Simple Program

\[
\begin{align*}
\text{f1}\ a\ b &= \text{f2}\ a\ b\ (2\cdot a)\ (3\cdot b) \\
\text{f2}\ a'\ b'\ c\ d &= \text{f3}\ a'\ b'\ c\ d\ (4\cdot c)\ (5\cdot d) \\
\text{f3}\ a'\ b'\ c'\ d'\ e\ f &= a' + b' + c' + d' + e + f
\end{align*}
\]
(b) Lambda-Lifted Simple Program

Figure 3-2: Examples of Lambda-Lifting

isters. The price for this simplicity is increased register pressure in the generated code since free variables for functions originally deeply-nested have to pass through registers in the code of all enclosing functions. Lambda-lifted programs are also not compatible with optimizations like closure-hoisting.

3.3 Non-Strictness

In the previous chapter we noted that non-strictness increases the expressiveness and the parallelism of \( \text{pH} \) programs. Again in this instance, the advantage for the programmer is the headache for the compiler writer. Implementing non-strictness poses two important challenges: differentiating between values and expressions and determining which expressions can be executed together.

Any non-strict language implementation must distinguish between a value and the expression that computes that value. These distinctions are not apparent in the \( \lambda \)-calculus model of a language, where equational reasoning holds and like-terms can be substituted freely, but it is paramount in low-level code. Take for instance a lazy language like Haskell. As we mentioned earlier, the semantics of Haskell dictate that only expressions that contribute to the final result of a program can be evaluated. All other expressions cannot produce values.

To effect this behavior, Haskell compilers delay every sub-computation by wrapping a \( \lambda \)-abstraction around it. These sorts of \( \lambda \)-abstractions, which are implemented as heap-
allocated closures, are sometimes called “thunks”. When it is certain that the value of an
expression is required, the code *forces* the corresponding thunk. This amounts to invoking
the \( \lambda \)-abstraction. The value of the expression is computed, and the result is memoized in
the space previously occupied by the thunk. If the value of the expression is required again
later, the memoized version will be used, so that an expression is evaluated at most once in
a lazy language.

But the non-strict semantics of \( pH \) are lenient rather than lazy. This means that sub-
computations in \( pH \) do not have to be delayed because any expression that may be evaluated
will be evaluated, even if it does not contribute to the final result of a program. Still, the
language is non-strict, and this means that the schedule for evaluating expressions cannot be
determined statically. Thus, the need to distinguish a computed value from an uncomputed
“expression” also arises in \( pH \), but to implement this distinction, we do not have to use the
expensive thunks of a lazy language.

We implement non-strictness in two ways in our code generation strategy. The first way
is through the use of presence state in memory. This allows us to differentiate whether
or not the contents of memory are valid. The second way is through the use of a proxy
instead of a value in non-strict situations (Traub [69, p.63] uses the term *promise*). The
basic difference between a proxy and a value is that the proxy identifies a value but it does
not *depend* on the value it identifies. In other words, a proxy can always be generated
for a value if the value is not strictly required. If a task does require a value but is given
a proxy instead, it uses the proxy and the presence state of memory to determine if the
actual value has been computed. If so, the value replaces the proxy and the task proceeds.
If not, the task must wait until the value is computed. A good implementation of proxies
will strive to make the proxy-or-value test very fast since it may occur quite often. In
a parallel implementation, proxies also imply synchronization between concurrent tasks,
and synchronization often mandates the use of expensive atomic operations. To maximize
performance in these two areas, the implementation of proxies has to be done with great
care.

Certainly, the best code would result if proxies could be avoided altogether. This would
require the compiler to schedule all tasks in a \( pH \) program statically. We illustrate why this is
not possible in general with the function \texttt{mkPair} based on examples from [57]. The function
takes two arguments and returns a *tuple* data structure, a simple pair of values.
Consider several different invocations of `mkPair`. The variables on the left-hand side of the equal sign are bound to the components of the tuple returned by the function:

\[
\begin{align*}
(a_1, b_1) &= \texttt{mkPair} \ 8 \ 9 \quad \Rightarrow \quad (16, 12) \\
(a_2, b_2) &= \texttt{mkPair} \ 8 \ a_2 \quad \Rightarrow \quad (16, 19) \\
(a_3, b_3) &= \texttt{mkPair} \ b_3 \ 9 \quad \Rightarrow \quad (24, 12)
\end{align*}
\]

The first invocation is straightforward. The computations of `a` and `b` in the body of `mkPair` appear independent, and although they could be executed in parallel, it would be much more efficient to execute them sequentially.

The second invocation of `mkPair` is trickier. The result `a_2` appears on both the left- and right-hand sides of the binding. In a strict language, this would be an error, but in non-strict `pH`, this is perfectly correct code. The allocation of the tuple is decoupled from the computation of its contents, so `mkPair` can return an "empty" tuple immediately after allocating it. Of course, the tuple is not really empty. It contains proxies for its two components. The computation of `a` can also proceed since the first argument to the function, `p`, is available. Once `a` is ready, it is stored into the first element of the tuple. That value then flows back in as the second argument of the function and enables the computation of `b`, the second component of the tuple. In this invocation of `mkPair`, the multiplication must execute before the addition in the body of the function.

The third invocation of `mkPair` presents the reverse situation. Due to the way the arguments are passed to the function, the addition must execute before the multiplication. In general, the order in which computations are executed in a `pH` function depends not only on the body of the function but also on the way it is invoked. In the case of `mkPair`, `a` and `b` must be computed by independent tasks, and the order in which these tasks are scheduled can be determined only at run-time. We call this kind of dependence between values a context dependence. The scheduling of the two computations depends on the context in which the procedure call occurs.

Context dependences occur in conditionals or can flow from callees to callers, as in the following example also from [57].
\begin{align*}
  f_1(x, y, z) &= (y, z, x) \\
  f_2(x, y, z) &= (z, x, y) \\
  kt_2 \text{ func } z &= \text{ let } (a, b, c) = \text{ func } x \ y \ z \text{ in } \\
  &\quad x = a + a \\
  &\quad y = b \ast b \\
  &\quad c \\
  g(z) &= \text{ kt}_2 f_1(z) \\
  h(z) &= \text{ kt}_2 f_2(z)
\end{align*}

In an invocation of \(g\), which leads to an invocation of \(f_1\), the multiplication in \(kt_2\) will occur before the addition. In an invocation of \(h\), the reverse is true. Due to the context dependences introduced by \(f_2\), the addition will occur before the multiplication. Thus, the order in which tasks are executed in \(kt_2\) can be determined only at run-time and depends on the function passed as a parameter to \(kt_2\).

On dataflow machines like Monsoon, run-time scheduling for \(pH\) is supported directly in hardware through mechanisms for cheap synchronization and context switching. The costs are small enough that a dataflow compiler for \(pH\) can exploit parallelism down to the level of individual primitive operations. Standard von Neumann processors do not include these facilities, so the overhead of creating and synchronizing fine-grain parallel activities is great compared to the computation they perform. The best performance is obtained in standard processors with long runs of sequential code that compute with values in registers. The need for run-time scheduling of small tasks, which is a direct consequence of non-strictness in the language, becomes a significant obstacle to generating good code.

### 3.3.1 Optimizing Non-Strictness

The best way of optimizing non-strictness is to get rid of it. In lazy languages, \textit{strictness analysis} is used to reduce the number of thunks that are created. In lenient \(pH\), \textit{partitioning} is used to form groups of expressions that can be computed in a single code sequence.

Strictness analysis is used to determine which arguments of a function are always evaluated. In lazy languages, this information is particularly useful because it means that a caller does not have to use thunks for the strict arguments of a callee. The arguments can be evaluated ahead of time, before the callee is invoked, and can be passed directly as values.

Peyton Jones [48, Ch. 22] describes an approach to strictness analysis through abstract interpretation. The results of the analysis are a set of annotations placed on callers
and callees describing which arguments can be evaluated in advance. Peyton Jones and Partain [47] measure the effects of such a strictness analyzer within the Glasgow Haskell Compiler, and state that the savings per argument can amount to 25 instructions per call, most of them memory references. On a mixed set of realistic benchmarks, they found the overall savings on program run-time due to strictness analysis to be up to 30%, though savings of 10% to 20% were more typical.

To improve the performance of lenient languages like pH, researchers have developed partitioning analysis. The goal of partitioning is to identify fine grain threads, for instance individual bindings, that can be combined into a coarser-grained thread without violating the semantics of the language. This transformation constitutes a reduction in the parallelism of a program, so it might seem counterproductive. However, most pH programs exhibit theoretical parallelism beyond that which could be exploited profitably by an actual machine, so lower parallelism does not imply lower performance. In fact, performance is improved because fewer and longer threads mean less synchronization, fewer context switches, and better use of processor resources like registers.

Traub [69] was the first to cast the problem of compilation of lenient languages for sequential machines as one of partitioning into threads. His analysis yields a function dependence graph which encompasses the dependences among computations in a function in all possible contexts. Traub also showed how to generate suspensive threads based on the results of his analysis.

Subsequent research and implementations of partitioning concentrated on generating non-suspensive threads. Non-suspensive threads result in simpler code-generation algorithms and make explicit the cost of switching from one thread to the next. In suspensive threads like Traub's, context switching is implicit in the behavior of synchronizing primitives. In the framework of the Threaded Abstract Machine, Schauser et al. [56] studied dependence and demand set partitioning with merging rules for additional grouping. This work was extended by Traub et al. [70] to include interprocedural partitioning, though not for recursive functions. This line of research culminated in Schauser's dissertation [55] which considers all cases.

Concurrently with Schauser, Coorg [15] also studied the problem of partitioning lenient languages. Within procedures, he used demand and tolerance sets to get slightly better partitions than earlier approaches. Across procedures, he used path semantics to determine
dependences. Though powerful, this approach incurs significant complexity in implementation.

It is interesting to note that partitioning has also been used, with much better results, to coarsen parallelism in strict languages. Working with Sisal, Sarkar [54] developed a partitioning algorithm based on a cost function for execution time that included terms for both parallelism and overhead. These two measures, as we hinted above, are generally at odds with each other since a program resulting from a highly-parallel partition will incur great scheduling overhead, while one with low overhead will likely be completely sequential. His algorithm operated on Sisal programs expressed in the IF1 hierarchical, graphical intermediate form. At its core, the partitioner iterated over the body of a procedure while merging, at each step, different sub-graphs until only one was left. On each iteration, the cost of the partition, as given by the cost function, was recorded. When the partitioner terminated, the partition with the lowest recorded cost, thus being the “fastest”, was reconstructed, and executable code was generated from it.

Sarkar’s work was based on a macro-dataflow model of parallelism. For multithreaded code, Tang et al. [65] present a partitioning algorithm (again, for strict programs) that includes a more sophisticated cost function, namely one that considers the ability of a partition to tolerate memory latency. Their results show that their partitioner can come very close to generating “ideal” partitions, and that it is particularly adept in handling fine-grain code. It still remains to be integrated into an actual compiler.

Partitioners for pH use no such sophisticated cost functions. Rather, they work on the premise that “bigger threads (coarser partitions) are always better.” This is understandable because non-strictness in pH, in a sense, straightjackets the partitioner by drastically limiting the ways in which expressions can be merged into threads. In the context of Sarkar’s work, it is as if non-strictness would force the partitioner to exit early, before working its way down to a single graph (thread) for a procedure, because additional iterations would produce incorrect code. The end results for non-strict languages are finer-grain and less efficient partitions.

3.3.2 Approach in pHc

The mechanism in pHc to implement proxies that distinguish computed from uncomputed values is described in depth in Chapter 4 and later. For the purpose of this overview, it
can be summarized as follows. Every value accessed by more than one computation and requiring synchronization is allocated a location on the heap. This location is initialized with a known "empty" pattern that allows the code to determine whether or not the contents have been computed. When we must pass a proxy for a value, we pass a specially-tagged pointer (a reference) to the heap location that will contain the value. Code that is passed a reference checks the location's value against the empty pattern. If there is a match, the code suspends. If not, the code continues. References to values may be copied freely.

Clearly, if we reduce the number of values requiring proxies, we will improve the performance of the code. To accomplish this, we need to reduce the number of values that are accessed by different threads, so we need to partition the program well. We implement the demand-tolerance set partitioning algorithm of Coorg in pHc, but we omit interprocedural analysis. Schauser notes that interprocedural partitioning only produced "modest" speedups [55, p.152] beyond good intraprocedural partitioning. In his benchmarks, the improvement was a maximum of 30%. We return to the discussion of partitioning and thread-generation in Chapter 6.

3.4 Synchronizing Memory

Most pH programs create lots of heap-allocated data structures. This is a result of a mostly-functional programming style that tends to create new values instead of updating old ones. The heap in pH is built on top of an I- and M-Structure substrate, so a fast implementation of synchronizing memory is a first-order concern for the pH compiler writer.

The challenges of implementing synchronizing memory on a standard machine are not subtle. First, an implementation must augment regular memory locations with presence information. The implementor can choose to store presence information with the values or in a separate area of memory, but there is little flexibility as to granularity: every word of the heap must have independent presence information. Second, the presence information must be accessible quickly, and the presence tests (whether a location is empty or full) must be fast. Finally, presence information must be updatable in coordination with the associated value. On a multiprocessor implementation such updates must be atomic to ensure the integrity of the synchronizing memory protocol.
3.4.1 Fast Implementations of Synchronizing Memory

Fast implementations of synchronizing memory have traditionally relied on hardware support. In the Monsoon processor, such support was extensive and took the form of a specialized memory controller designed by Ken Steele [63]. The controller was essentially a simplified version of a Monsoon processor element, but it was still fully pipelined and included a microcoded protocol engine.

The Sparcle processor on the Alewife machine also includes support for synchronizing memory [33]. The designers of Alewife, however, implemented a small (though highly integrated) hardware component, to handle the common cases, and took care of the rest in software. The hardware consisted of a modified SPARC processor with 12 additional instructions that trapped if the low bit was set in a memory address. Even this limited support was found to contribute significantly to the performance of applications that rely on fine-grain synchronization. The supercomputer-class Tera MTA [2] follows a similar strategy, though it was not inspired by Alewife. Memory in the Tera MTA is augmented with four access state bits which can cause lightweight traps in the processor. Trap handlers can be used to implement synchronizing memory.

3.4.2 Approach in pHc

Synchronizing memory in pHc is implemented completely in software since standard SMPs include no hardware support for I- and M-Structures beyond basic atomic memory operations. We developed a novel 64-bit data representation, described in Section 7.3, that allows us to store presence information along with data with a minimal reduction in the range of data represented. The representation also provides for fast presence status checks and allows simultaneous update of presence and data.

While this software-only approach is slower than one with hardware support, it is still amenable to optimizations. For example, a compiler analysis that considers the use of data structures in conjunction with partitioning could determine that some data structures do not need to be synchronized at all. All elements are produced before they are consumed. Such data structures could be accessed with standard load and store instructions with much less overhead than the full I- or M-Structure protocols.
3.5 Barriers

Barriers in pH require substantial bookkeeping for two reasons. First, the granularity of the tasks controlled by a barrier can be very fine, as small as the simplest expression in pH. Second, the extent of a barrier is dynamic. The barrier guarantees that all computation in the pre-region, including computation in functions invoked by the pre-region, has terminated before the post-region executes. As we noted in the previous chapter, sync statements in Cilk provide the same kind of guarantee. In Cilk, however, only procedure calls can be spawned as parallel tasks, and these are much coarser-grained computations than pH expressions.

3.5.1 Barrier Implementations

Logically, the implementation of a barrier requires a counter to keep track of outstanding tasks. When a task is created, the counter is incremented, and when a task terminates, the counter is decremented. When the counter reaches zero, the barrier discharges.

In practice, this single logical counter is often implemented using a number of distributed counters that are linked together. This is the strategy used to implement barriers in Cilk. In Cilk, a sync statement in the body of a procedure forces execution to wait (using a counter) for all previously spawned procedures to return. Recursively, each spawned procedure returns only when all of its spawned callees have finished. To maintain this invariant, Cilk inserts an implicit sync at the end of every procedure. Thus, the logical barrier counter at the top of a Cilk call-tree is implemented by local counters within each procedure.

A similar scheme has been used in previous compilers for Id, but with an additional complication. In a strict language like Cilk, the return of a value from a procedure is enough to indicate termination of the callee, but this is not possible in a non-strict language. A non-strict procedure must be able to return a value as soon as possible, even before all computation terminates. Consequently, in Id compilers, every procedure returns two results: a value and a signal. The key point to remember is that due to non-strictness each result is returned independently and possibly at very different times during execution.

The signal is used to keep track of tasks which do not contribute to the value of a procedure. The value is dependent on the completion of some portion of all the tasks created in a procedure, so its computation implies their completion. But many tasks, such
as pure side-effects, do not contribute to the return value. In the executable code, the signal is made to depend on the completion of these other tasks. Within each procedure, the signal is implemented as a local synchronization counter, and each procedure includes within its own signal tree the returned signals of its callees. Only the return of both a value and a signal marks the termination of a callee, and this is the condition used to implement barriers.

3.5.2 Barriers in pHc

In pHc, we use a single counter for every barrier rather than a distributed counter. A barrier is represented as a heap-allocated data structure, which includes the counter and some additional state, and a pointer to it is passed from caller to callee in a dedicated register. This approach simplifies procedure calls. We trade the complexity of returning two independent results from a procedure, the value and the signal, for an additional argument to every procedure, the current barrier. In addition, the code within a procedure is cleaner since we don’t have to keep track of both values and signals.

On the other hand, this approach may lead to contention for the barrier counter if there is a lot of parallelism in the barrier pre-region. Currently, this appears to be a negligible source of overhead in our code. In addition, as we’ll see later, with this scheme we lose the ability to determine exactly when a callee terminates. This does not affect the ability of the compiler to generate correct code, but it does complicate the deallocation of procedure activation frames.

3.6 Garbage Collection

Garbage collection is a well-studied problem in sequential, distributed, and even parallel environments [30], and the issues in the pH compiler are not particularly different than in other systems. Still, garbage collection is hard to implement, so it merits a brief mention in this chapter.

The root of the problem is that precise (not conservative) garbage collection does not lend itself to a modular implementation. In pH, for instance, the data representation is self-identifying so that the garbage collector can trace memory and distinguish pointers from scalars; the code-generator has to remember to keep track of a correct root set at
all times when a garbage collection might occur, and it also has to maintain invariants like no pointers into the middle of objects (these complicate tracing of the heap); even the scheduler is affected, as it must check for a pending garbage collection after running out of work but before stealing work from other processors. Such a pervasive influence tends to weave intricate dependences between the components of a system, and it complicates the overall design and implementation effort.

3.7 Summary

pH was designed to make parallel programming easy, efficient, and robust. The consequence of these design choices is that the pH compiler writer must overcome a broad set of challenges to generate good code. In languages like Cilk, many of these problems do not even exist since such languages are often constrained in order to simplify compilation. In this chapter, we concentrated on these issues:

- **Polymorphism** in the type system favors uniformity in calling conventions and data structure representations.
- **Higher-order functions** force dynamic checks before applications and rely on efficient environments for free variables.
- **Non-strictness** requires a mechanism to distinguish expressions from computed values at run-time. In addition, non-strictness limits the ability of the compiler to coalesce expressions into sequential threads.
- **Synchronizing memory** requires fine-grain presence information in a form that is easy to access, provides for fast status checks, and can be updated atomically.
- **Barriers** demand careful bookkeeping of the termination of all spawned tasks, not just procedures.
- **Garbage collection** affects the entire system from data representation, to code generation, to scheduling.
Chapter 4

The SMT Machine

The process of converting a $pH$ source program into von Neumann machine code is an arduous one. As we have seen in the previous chapters, the language includes a number of powerful features to make parallel programming easy, but the efficient implementation of these features is not straightforward. Furthermore, there is a "semantic chasm" between the source and the target languages. Even the $\lambda^*$-calculus remains a high-level language whose semantics are far removed from von Neumann code.

To simplify the problem, the pHc code generator makes use of two key abstractions. The first is the $\lambda^*$-calculus described earlier. The second is the Shared-Memory Threaded (SMT) machine, the subject of this chapter. The SMT machine is an idealization of a modern symmetric multiprocessor. It reflects the important properties of this whole class of machines while insulating the code generator from the quirks of any particular instance.

It provides the bridge between $\lambda^*$ and executable code. Thus, a source $pH$ program is translated first into $\lambda^*$, then into SMT instructions, and finally into executable code.

Abstract machines have long been used to describe the evaluation model of functional languages and as compilation targets. In the case of Haskell, the Spineless-Tagless G-machine [49] is at the core of the Glasgow Compiler. For Id, P-RISC [18] and TAM [16] have served as abstract code-generator targets. The SMT machine itself is based on the work of Aditya et al. [1] who used an early version to give a semantics of $pH$. It should be clear, then, that the SMT machine was not designed in a vacuum. It includes mechanisms, such as support for non-strictness and parallelism, specifically intended to simplify the compiling strategy given in the next chapter. Still, these mechanisms were designed to be as lean as
possible, and their translation to executable code is not difficult.

4.1 Overview

The SMT machine is illustrated in Figure 4-1. It consists of a number of von Neumann processors each with independent register state but all sharing a global memory. The processors also share a global work queue. The work queue contains data structures describing tasks that are ready to run. The SMT machine has an instruction set that is more flexible than that of the typical von Neumann RISC machine. For instance, it offers indirect addressing, instructions that work with both general purpose and floating point registers, and instructions with a variable number of operands. More importantly, it offers instructions for implementing synchronizing memory, barriers, and task scheduling.

Each processor in the machine contains a register file for general purpose (GP) registers and a separate one for floating point (FP) registers. GP registers can hold integers, memory
addresses, or floating point values, but floating point operations only operate on FP registers. By allowing GP registers to contain FP values, we simplify the data representation and the implementation of polymorphic function calls. Each processor also contains three special registers: the program counter register rpc, the current barrier register rB, and the current task register rT.

SMT global memory is word-addressed and includes per-word presence information. A word is the widest value that can fit in a register. Instructions are also the size of a word, so that incrementing rpc by 1 causes it to point to the next instruction. Global memory is also coherent, but its consistency model is weak. Programs cannot assume that the order in which memory operations are issued on one processor will match the order they are detected by other processors.

In the design of the machine, we made a conscious decision to avoid specifying particular bit-sizes for basic data types like integers and floats. Our single assumption is that a GP register can contain any scalar quantity. An SMT implementation is free to define its own sizes for data types. To make particularly efficient use of memory, one might also define additional memory load and store instructions that operate at a finer granularity than a "word". However, this is not a trivial decision since the granularity of presence bits in memory might also have to change. For the purposes of our compiling strategy, word-based addressing is sufficient.

4.1.1 Instruction Descriptions

The entire SMT machine instruction set is summarized in Figure 4-2. In the following sections, the behavior of each instruction is described using a simple register transfer level (RTL) notation (Figure 4-3). A description has the following format:

\begin{verbatim}
Instruction

Description of instruction in RTL notation
\end{verbatim}

It is important to note that SMT instructions execute atomically, even when they access memory. The RTL description of the instruction constitutes a single, indivisible action.

A note about field selection syntax: the ' . ' is used to hide the details of accesses to multi-word data structures. In final code generation, field selection is replaced with a numeric offset.
<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>PURPOSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>spawn</td>
<td>Add task to the work queue</td>
</tr>
<tr>
<td>schedule</td>
<td>Schedule task from the work queue</td>
</tr>
<tr>
<td>jump</td>
<td>Conditional control transfer</td>
</tr>
<tr>
<td>switch</td>
<td>Computed jump, unconditional</td>
</tr>
<tr>
<td>allocate</td>
<td>Memory allocation</td>
</tr>
<tr>
<td>load</td>
<td>Read from memory, unsynchronized</td>
</tr>
<tr>
<td>store</td>
<td>Write to memory, unsynchronized</td>
</tr>
<tr>
<td>touch</td>
<td>Presence status check, synchronizing</td>
</tr>
<tr>
<td>take</td>
<td>Read from memory with mutual exclusion, synchronizing</td>
</tr>
<tr>
<td>istore</td>
<td>Write to memory, synchronizing</td>
</tr>
<tr>
<td>initbar</td>
<td>Barrier initialization</td>
</tr>
<tr>
<td>touchbar</td>
<td>Barrier status check, synchronizing</td>
</tr>
<tr>
<td>incbar</td>
<td>Barrier counter increment</td>
</tr>
<tr>
<td>decbar</td>
<td>Barrier counter decrement</td>
</tr>
<tr>
<td>plus, etc.</td>
<td>Primitive operations</td>
</tr>
<tr>
<td>move</td>
<td>Register-to-register move</td>
</tr>
</tbody>
</table>

Figure 4-2: SMT Instruction Set Summary

<table>
<thead>
<tr>
<th>SYNTAX</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>←</td>
<td>Assignment</td>
</tr>
<tr>
<td>.</td>
<td>Structure field selection</td>
</tr>
<tr>
<td>π</td>
<td>Presence field of memory cell: full or empty</td>
</tr>
<tr>
<td>δ</td>
<td>Data field of memory cell</td>
</tr>
<tr>
<td>ra</td>
<td>Some GP register</td>
</tr>
<tr>
<td>r1</td>
<td>Specific GP register</td>
</tr>
<tr>
<td>rfa</td>
<td>Some FP register</td>
</tr>
<tr>
<td>rf1</td>
<td>Specific FP register</td>
</tr>
<tr>
<td>rpc</td>
<td>Program counter register</td>
</tr>
<tr>
<td>( ra )</td>
<td>Contents of register ra</td>
</tr>
<tr>
<td>a</td>
<td>Memory address</td>
</tr>
<tr>
<td>Mem[a]</td>
<td>Contents of memory</td>
</tr>
<tr>
<td>WQ</td>
<td>The global work queue</td>
</tr>
<tr>
<td>cond</td>
<td>Boolean conditional expression</td>
</tr>
<tr>
<td>a, ..., z</td>
<td>Utility variables</td>
</tr>
</tbody>
</table>

Figure 4-3: RTL Notation for SMT Instruction Descriptions
The SMT instruction set supports two addressing modes for use in memory instructions. We describe these with RTL notation:

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Syntax</th>
<th>Address in RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Indexed</td>
<td>ra[i]</td>
<td>(ra) + i</td>
</tr>
<tr>
<td>Register Indexed Indirect</td>
<td>@ra[i]</td>
<td>Mem[(ra) + i]</td>
</tr>
</tbody>
</table>

If the index is omitted in an addressing expression, it is assumed to be zero.

### 4.1.2 Register State Across Suspensive Instructions

The suspensive instructions in the SMT instruction set are touch, take, and touchbar. When one of these instructions executes, there is the possibility that a task might have to suspend. If so, the instruction set implicitly saves all register state and restores it when the task resumes. In practice, only a limited number of registers actually contain live values, so saving the entire register set is a waste.

To remedy this problem, suspensive instructions include a live-state annotation such as "{r1, r3}". The annotation tells exactly which registers have to be saved and restored when the task suspends and resumes. The values of the registers are saved in the current task structure pointed to by register rT, and it is up to the code-generator to guarantee that enough memory is available in the task structure to store the registers. The identity of the registers saved is encoded in a register mask also stored in the task structure.

### 4.1.3 Compilation Preview

Before moving on to the description of the instruction set, we give a preview of some of the major features of the compilation strategy in the next chapter. It is useful to keep these in mind when trying to understand the behavior of each instruction.

- On invocation, each procedure is allocated an activation frame on the global heap. The activation frame is used for storing variables local to the procedure as well as barrier and task structures. A frame is allocated in the global heap rather than in local processor storage because different tasks from the same procedure might execute in parallel on different processors.
- The source language is non-strict, so the SMT code often needs to refer to values without actually computing them. As we explain in detail in Section 5.1.1, we use pointers to pass references to values. The use of references is important in preserving sharing since...
the transition from uncomputed expression to computed value is a change that must be
seen by all consumers of a value.

- The basic entity that can be scheduled on any processor is a task (thread). A task
  consists of a program counter address and a set of live registers. The state of each task
  is stored separately in a task structure in the activation frame of the enclosing procedure.
The maximum size of the state for a task is the entire register file.
- In addition to the special registers defined the by the machine model, the compiler also
  assigns special roles to some registers. For instance, a particular register rF always
  carries a pointer to the activation frame of the current procedure.

4.2 Arithmetic and Move: plus, etc., and move

The simplest instructions in the SMT instruction set are those that implement arithmetic
primitives. We use plus as an example:

\[
\begin{align*}
\text{plus}(ra, rb, rc) & \\
ra & \leftarrow (rb) + (rc) \\
rpc & \leftarrow (rpc) + 1
\end{align*}
\]

The leftmost register is the destination for the results of the plus instruction. The RTL
notation states that the contents of the rightmost registers are added and deposited in the
destination register. The same instruction is also used to represent floating-point addition
with the use of FP registers as operands. Constants are also valid operands.

The move instruction is also easy to explain: it moves a value into a register. The value
can be the contents of another register or a constant.

\[
\begin{align*}
\text{move}(ra, rb) & \\
ra & \leftarrow (rb) \\
rpc & \leftarrow (rpc) + 1
\end{align*}
\]

4.3 Control Flow: jump and switch

SMT offers two control flow instructions: jump and switch. The jump instruction is used for
both conditional and unconditional jumps. The first operand of the instruction is a boolean
test, the second is the target jump address, and the third is an optional register to store a
"return link", the address of the instruction following the jump. The return link is used to implement subroutine calls.

\[
\text{jump}(\text{cond}, a, rc)
\]

- If cond
- then \( rc \leftarrow (rpc) + 1 \)
- \( rpc \leftarrow a \)
- else \( rpc \leftarrow (rpc) + 1 \)

The conditions available for the cond operand include the usual comparison operators (<, >, ...) as well as additional ones to test whether or not the contents of a memory cell are full or empty (isFull, isEmpty). The condition Always is always true and is used to express an unconditional jump.

The switch instruction is used to implement multi-way jumps. The first operand of the instruction is a constant or register whose value is non-negative and determines the alternative address to jump to.

\[
\text{switch}(ra, a_0, ..., a_n)
\]

- If \( 0 \leq (ra) \leq n \)
- then \( rpc \leftarrow a_{(ra)} \)
- else Error!

If the value of the first operand does not identify one of the alternatives, the machine halts with an error. The switch instruction is particularly useful because it simplifies the compilation of the Case expression in \( \lambda^* \).

### 4.4 Scheduling: spawn and schedule

The next two instructions are used to access the global work queue of the SMT machine. The spawn instruction inserts a task structure into the work queue, \( WQ \). We treat the work queue as a list, so the operation of adding an element is represented as "::" (cons).

\[
\text{spawn}(a_{task})
\]

- \( WQ \leftarrow a_{task} : WQ \)
- \( rpc \leftarrow (rpc) + 1 \)

Below, we illustrate a task structure. The core components are the next field and the pc field.
Task Data Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>full</td>
<td>Pointer to next task structure in list</td>
</tr>
<tr>
<td>pc</td>
<td>full</td>
<td>Program counter for task</td>
</tr>
<tr>
<td>regmask</td>
<td>full</td>
<td>Register mask: encodes identity of saved registers</td>
</tr>
<tr>
<td>taskstate₁</td>
<td>full</td>
<td>Saved register</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>Saved register</td>
</tr>
<tr>
<td>taskstateₙ</td>
<td>full</td>
<td>Saved register</td>
</tr>
</tbody>
</table>

The *next* field is used when the task structure forms part of a list (as in the work queue). The *pc* field is used to store the address of the instruction where execution is to proceed in the task. Initially, this is the address of the first instruction in the task. Later, if the task suspends, it contains the address of the instruction where execution is to resume. The remaining fields are used to store the values of registers that are live when the task suspends.

The schedule instruction extracts a task data structure from the work queue. It initializes the program counter with the contents of the *pc* field and the rest of the registers based on the register mask and the contents previously saved in the task structure. To simplify the description of the instruction, we ignore the register mask and assume all registers are restored. If the queue is empty, then the instruction effectively waits until some new task is placed onto the work queue.

```
loop:
if (nil WQ)
then goto loop
else rT ← head WQ
    WQ ← tail WQ
    rpc ← Mem[rT.pc]₆
    r1 ← Mem[rT.taskstate₁]₆
    r2 ← Mem[rT.taskstate₂]₆
    ...
    rn ← Mem[rT.taskstateₙ]₆
```

It is important to note the following points. First, access to the work queue in both of these instructions is atomic. Second, *spawn* does not affect control flow. After the instruction is finished, control passes to the following instruction. Finally, the symbolic names, like *pc* and *taskstate₁*, corresponding to task structure fields free us from having to use numeric
field offsets.

4.5 Memory: allocate, load, touch, store, istore, and take

The allocate instruction reserves fresh storage from the shared global memory. The register \( rb \) contains the number of adjacent cells to allocate, and after the instruction executes, the \( ra \) register contains the address of the first of these. The presence fields of the memory cells are initialized to empty.

\[
\text{allocate}(ra, rb)
\]

\[
\text{Mem}[a] \leftarrow \text{empty} \\
\text{Mem}[a + 1] \leftarrow \text{empty} \\
\ldots \\
\text{Mem}[a + (rb) - 1] \leftarrow \text{empty} \\
ra \leftarrow a \\
rpc \leftarrow (rpc) + 1
\]

The load instruction reads a value from memory into a register. The instruction pays no attention to the presence field of the memory cell. The structure of the SMT machine code must guarantee that a valid value is available.

\[
\text{load}(ra, a)
\]

\[
ra \leftarrow \text{Mem}[a] \\
rpc \leftarrow (rpc) + 1
\]

The touch instruction is used to check the presence status of a memory cell. A touch instruction issued against a full memory cell does nothing; control passes immediately to the next instruction. However, if the memory location is empty, the touch instruction forces the current task to suspend, saves all registers in the live-state annotation (only two in this example) in the current task structure, places the current task structure on the deferred list associated with the memory cell, and schedules a new task from the global work queue, just like the schedule instruction. Remember that the deferred list is used to record all the tasks that are waiting for the data field of the memory cell to be computed. A pointer to the head of the deferred list is stored in the data field of the memory cell while the presence field is empty.

69
touch(a) \{ra,rb\}

\[
\begin{align*}
\text{if} & \quad (Mem[a], == \text{empty}) \\
\text{then} & \quad Mem[rT.pc]_\delta \leftarrow (rpc) + 1 \\
& \quad Mem[rT.taskstate_1]_\delta \leftarrow (ra) \\
& \quad Mem[rT.taskstate_2]_\delta \leftarrow (rb) \\
& \quad Mem[rT.regmask]_\delta \leftarrow ra \oplus rb \\
& \quad Mem[a]_\delta \leftarrow (rT) : Mem[a]_\delta \\
& \quad \text{do schedule} \\
\text{else} & \quad rpc \leftarrow (rpc) + 1
\end{align*}
\]

Note that a task that suspends due to a touch instruction always resumes at the instruction following the touch since that is the value placed in the \(pc\) field of the task structure. Also, we note again that the touch instruction, and all other SMT instructions, execute atomically. This means that no other instruction can modify the memory locations the touch is accessing while it is executing.

The address \(a\) in the description of touch is called a reference (or location pointer). References play a central role in the implementation of non-strictness because, as we’ll see in the next chapter, they can be used to identify an expression regardless of whether or not the expression has been computed.

The store instruction writes the contents of a register into memory and sets the presence field to full. It ignores the old values of the presence and data fields of the memory cell.

\[
\text{store}(a, ra)
\]

\[
\begin{align*}
Mem[a]_\delta & \leftarrow (ra) \\
Mem[a]_\pi & \leftarrow \text{full} \\
rpc & \leftarrow (rpc) + 1
\end{align*}
\]

In contrast, the behavior of istore depends on the value of the presence field. If the cell is empty, then the istore updates the data field of the cell, changes the presence field to full, and places all the tasks on the deferred list \((d)\), which could be an empty list, back onto the work queue. We use the list append operation \((++\) to accomplish this task. If the memory cell is full, then the istore signals an error, and the machine halts.
istore(a, ra)

if (Mem[a] == empty)
then d ← Mem[a]
    Mem[a] ← (ra)
    Mem[a] ← full
    VQ ← d-H
    WQ ← d++ WQ
    rpc ← (rpc) + 1
else Error!

It is easy to see that load, touch, and istore provide the basic operations for implementing I-Structures. But M-Structures require an additional instruction, take, since they offer mutable, not single-assignment, semantics. The take instruction is used to extract a value from an M-Structure cell. If the presence field of the cell is full, then the take changes it to empty and extracts the data field. If the cell is empty, then take causes the current task to suspend.

take(ra, a) {ra, rb}

if (Mem[a] == empty)
then Mem[rT.pc] ← (rpc)
    Mem[rT.taskstate1] ← (ra)
    Mem[rT.taskstate2] ← (rb)
    Mem[rT.regmask] ← ra ⊕ rb
    Mem[a] ← (rT): Mem[a]
    do schedule
else ra ← Mem[a]
    Mem[a] ← nil
    Mem[a] ← empty
    rpc ← (rpc) + 1

A task that suspends due to a take resumes when an istore is issued against the memory cell. The task resumes at the same take instruction that suspended, so it tries grab the value of the memory cell again. Many tasks may be trying to do so concurrently, but only one will succeed at a time. This somewhat cumbersome protocol guarantees mutual exclusion and prevents any single task from dominating access to the M-Structure cell.
4.6 Barrier Instructions: initbar, touchbar, incbar, and decbar

The sequential statement conjunction (---) of λ* is implemented in SMT using a barrier data structure. The barrier, illustrated below, consists of three fields: a counter field (count) containing an integer; a previous barrier field (prev) pointing to the old barrier in effect when this one was created; and post-region field (post) possibly containing a pointer to a task. Barrier structures are used to keep track of the termination of tasks created by spawn instructions. The role of these instructions in the λ* compilation strategy is explained in the next chapter, so here we focus only on their individual behavior.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Count of outstanding activities in pre-region</td>
</tr>
<tr>
<td>prev</td>
<td>Pointer to previous barrier</td>
</tr>
<tr>
<td>post</td>
<td>Pointer to task structure for barrier post-region</td>
</tr>
</tbody>
</table>

Barrier structures and the barrier instructions described below should be regarded as the definition of an abstract data type. It is entirely possible to define barriers using standard synchronizing memory instructions, but we found that the use of separate instructions simplified the presentation.

The initbar instruction is used to initialize barrier structures. We assume the space for the barrier has been created using the allocate instruction and its address is aBarrier. The count field is initialized with register ra, the prev field is initialized with the current barrier structure from register rB, and the post field is left untouched. It is important the save the value of the rB register. The fine-grain barriers of pH can be nested, and this means that different threads, possibly scheduled one after the other, might belong to different barrier regions. In the case of the initbar instruction, the current value of rB must be restored once the post-region is ready to execute. Once initialized, the new barrier is made the current barrier by storing its address in rB.

\[
\text{initbar(aBarrier, ra)}
\]

\[
\begin{align*}
\text{Mem}[\text{aBarrier}.\text{count}]_5 &\leftarrow (\text{ra}) \\
\text{Mem}[\text{aBarrier}.\text{prev}]_5 &\leftarrow (\text{rB}) \\
\text{rB} &\leftarrow \text{aBarrier}
\end{align*}
\]

The key component of the barrier structure is the counter. The idea is that if the barrier counter field is zero, then the pre-region has terminated (the barrier is said to discharge),
and the post-region can execute.

The touchbar is designed to be the instruction that separates the pre-region from the post-region. It checks the condition that the barrier counter is zero. If the barrier has discharged, then the instruction updates the rB register and continues. If the barrier has not discharged, the instruction suspends the current task before control flows into the post-region. The instruction is designed so that only one touchbar should execute against a particular barrier structure.

\[
touchbar(abarrier) \{ ra, rb \}
\]

\[
\begin{align*}
\text{if} & \quad (\text{Mem}[abarrier.count]_δ == 0) \\
\text{then} & \quad \text{rB }\leftarrow \text{Mem}[abarrier.prev]_δ \\
& \quad \text{rpc }\leftarrow (\text{rpc}) + 1 \\
\text{else} & \quad \text{Mem}[rT.pc]_δ \leftarrow (\text{rpc}) + 1 \\
& \quad \text{Mem}[rT.taskstates_1]_δ \leftarrow (ra) \\
& \quad \text{Mem}[rT.taskstates_2]_δ \leftarrow (rb) \\
& \quad \text{Mem}[rT.regmask]_δ \leftarrow ra \oplus rb \\
& \quad \text{Mem}[abarrier.post]_δ \leftarrow (rT) \\
& \quad \text{Mem}[abarrier.post]_π \leftarrow \text{full} \\
& \quad \text{do schedule}
\end{align*}
\]

The incbar instruction increments the value of the count in a barrier. In the compilation strategy shown in the next chapter, an incbar instruction is issued every time a new task is spawned.

\[
\text{incbar(abarrier)}
\]

\[
\begin{align*}
\text{Mem}[abarrier.count]_δ \leftarrow \text{Mem}[abarrier.count]_δ + 1 \\
\text{rpc }\leftarrow (\text{rpc}) + 1
\end{align*}
\]

The decbar instruction is used to decrement the count of a barrier structure. It complements the use of incbar to mark the termination of a task. If the value of the counter is one, then the instruction checks the post field of the barrier. If the post field is empty, then the counter is decremented. This configuration of the barrier structure arises when the last decbar instruction for a particular barrier occurs before the single touchbar associated with the barrier.

On the other hand, if the field is full, it contains a suspended task placed there by the touchbar instruction. This configuration of the barrier structure arises when the touchbar instruction executes before the last decbar. This indicates that at the time control-flow
reached the end of the pre-region (immediately before the post-region), some tasks spawned in the pre-region were still executing. According to the sequential semantics of barriers, the post-region must wait until those tasks terminate. When those tasks terminate, they execute decbar instructions, so the last decbar must restart the post-region.

\[
decbar(abarrier)
\]
\[
c \leftarrow Mem[abarrier\_count]_\delta \\
if (c > 1) \\
then Mem[abarrier\_count]_\delta \leftarrow c - 1 \\
else Mem[abarrier\_counter]_\delta \leftarrow 0 \\
\quad if (Mem[abarrier\_post]_\pi == full) \\
\quad then WQ \leftarrow Mem[abarrier\_post]_\delta : WQ \\
rpc \leftarrow (rpc) + 1
\]

4.7 Summary

In this chapter, we have described the SMT machine and its instruction set. The SMT machine models a basic symmetric multiprocessor, but it includes extra instructions to support multithreaded execution and non-strictness. With instructions like touch and schedule, we capture simple mechanisms for synchronization and scheduling that are not available in real machines.

After the λ*-calculus, the SMT machine is the second major abstraction upon which our compiler is built. Its use has simplified the development of the code generator in two ways. First, the SMT machine allows us to focus on the important problems of code generation, emitting correct and efficient code, without worrying about the quirks of any particular target machine. Second, it simplifies retargetting the back-end to different platforms since, in principle, no modules that use SMT need to be rewritten when the target architecture changes.
Chapter 5

Compiling $\lambda^*$ to SMT

This chapter discusses the compilation of a $\lambda^*$ program into SMT instructions. This involves the transformation of a program where parallelism is implicit and operations are only loosely ordered into one where parallelism is explicit and control-flow follows the serial von Neumann model. In addition, the compilation takes a program from a high-level representation into a low-level one.

The simple SMT instruction set forces the compiler writer to take a position on many issues that are not relevant to $\lambda^*$, like data structure representations and procedure calling conventions. This chapter opens with a discussion of these issues which together determine the “execution environment” of SMT code. The rest of the chapter describes how each $\lambda^*$ expression is compiled into SMT code. This style of compilation has its roots in the work of Aditya et al. [1] and Arvind et al. [7, 6]. The SMT code generated by compilation rules presented in this chapter is true to the semantics of $\lambda^*$, but has not been optimized for efficiency. Later passes in the compiler for partitioning and threading, discussed in the next chapter, are designed to squeeze out some of the overhead introduced here.

5.1 Execution Environment

This section describes how compiler-internal and user data structures are organized. User data structures result from the application of constructors defined in type definitions. The internal data structures include references to represent values that may not have been computed, activation frames to store local procedure values, closures to represent $\lambda$-abstractions, tasks, and barriers. Tasks and barriers were discussed earlier along with the SMT instruc-
tions to manipulate them. Internal data structures are not visible in the user’s program, but rather, are used only by the compiler to implement objects outside the language model.

5.1.1 References to Values

An implementation of $\lambda^*$ on a von Neumann machine (like SMT) requires a mechanism by which programs can refer to values without needing to compute them. Consider, for instance, the problem of creating a closure. A closure contains a vector of free variables that captures those pieces of the lexical environment that might be used at the invocation of the closure. What object should be stored in the closure for a free variable? One option is to compute and store the value of the free variable. However, if that value is not used when the closure is invoked (or if the closure is never invoked), then the value will have been computed prematurely. This approach is incorrect in many contexts because closures are supposed to be non-strict with respect to free variables.

The correct strategy is to store the name of the free variable in the closure. A concrete implementation of “the name of a variable” is a pointer to the memory location that may eventually hold the value of the variable. We say “may” because the variable might never be computed if, for instance, the task that uses it appears on the branch of a conditional that is not taken. This location pointer (reference) uniquely identifies the variable, yet its use does not require the variable to be computed. In the compilation rules in this chapter, references are often used to preserve the non-strict semantics of $\lambda^*$, especially in the presence of barriers, conditionals, closures, and function calls.

Other non-strict languages, like Haskell, also need to represent values without computing them. Because it is lazy, a Haskell implementation that computes a value prematurely risks violating the semantics of the language. Instead, the implementation passes a proxy for a value until the value is used in a strict construct (e.g., primitive operation or case expression). That proxy is usually a pointer to a closure (thunk). In pH, we don’t use thunks because we can dissociate computations from the values they produce. A reference simply names a value that may or may not be computed, but it need not tell us anything about the task that computes it.
5.1.2 User Data Structures

User data structures in $\lambda^*$ are created using constructors. For instance, the `Cons` and `Nil` (`:` and `[]` in `pH` syntax) constructors are used to create lists, and `Pair` (``,` ` in `pH` syntax) is used to create 2-tuples. In addition, the user can define new constructors like `Leaf` and `Node`, as we showed in Section 2.2.

The most important thing to remember about constructors is that they are non-strict with respect to their arguments. This allows the “shell” of a new data structure to be created and passed around a program even while its contents are being computed. To preserve non-strictness, the implementation of constructors relies heavily on references. For example, the result of the expression `Node(x,y)` is the following data structure:

<table>
<thead>
<tr>
<th>Node(x,y) Data Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tag</strong></td>
</tr>
<tr>
<td><strong>left subtree</strong></td>
</tr>
<tr>
<td><strong>right subtree</strong></td>
</tr>
</tbody>
</table>

The structure consists of three memory cells: the first is a `tag` used internally by the compiler to distinguish different constructors for the purposes of `Case` expressions. Tags are discussed in detail in Section 5.3. The next two cells contain references to the components of the `Node` data structure. In general, the compiler cannot be certain that the values corresponding to the contents of the `Node` have been computed when the `Node` is created. The compiler can be certain, however, that references to the contents will be available, so it uses these instead. In this way, the `Node` constructor executes non-strictly: it allocates memory for the data structure, fills it with references, and returns it without having to wait for any other computations to terminate.

The discussion of user data structures in this dissertation is limited to algebraic types. Arrays are also available in `pH`, and they are implemented as abstract types using the basic I- and M-Structures we have presented. We omit arrays from the presentation because they do not require any additional compilation machinery.

5.1.3 Activation Frames

The remaining sections describe data structures that are not visible directly in `pH` or $\lambda^*$ programs. They are implicit in the high-level representation of the program, but they must
be defined explicitly at the SMT level. The most important of these is the *activation frame*. An activation frame is created for each invocation of a function. It provides space for parameters, local variables, tasks, and barriers. An activation frame for a function can be deallocated when all the tasks within the body of the function have terminated.

Activation frames in \( \lambda^* \) are managed differently than in most languages. In C, for example, a simple stack, separate from the heap, can be used to allocate and deallocate activation frames because the language is strict and sequential. Callers and callees execute in a LIFO order since a callee always terminates before its caller. In pH and most other parallel languages, function calls constitute the coarsest-grain parallelism in a program. A function and its callees can run in parallel, and callees can terminate out of order. With a stack for activation frames, this would prevent callee frames from being deallocated until all parallel calls have completed, and this might prove to be extremely inefficient. Instead, what is commonly done is to allocate frames on the heap, just like any other data structure, and to link them into a general activation frame tree that follows the call tree of the program.

A pH activation frame is divided into five areas though frames for simple procedures might include only a subset of these:

1. The *caller link* area contains information about the caller of a procedure: a pointer to the caller’s activation frame, the return address into the caller’s code, and a pointer to the calling task structure.

2. The *arguments* area contains references to the arguments of the procedure. The storage for the argument values is created earlier in the call tree. Arguments are passed as references to maintain non-strictness since they allow a function to be invoked, to execute, and to return even if no arguments have been computed.

3. The *locals* area contains references to local variables. The memory cells where the values of local variables are stored are allocated separately from the frame because the lifetime of this storage may outlive the frame. How? A procedure can pass a reference to a local variable as an argument to a child procedure, but due to non-strictness, the child procedure may outlive the parent. Thus, the memory cell the reference points to cannot be deallocated along with the caller’s frame.

4. The *barrier pointers* area contains pointers to the barrier data structures required by the procedure. Barriers are allocated separately at the time of function invocation. In theory, barriers could be allocated directly as part of the activation frame since the
lifetime of the barriers exactly matches the lifetime of the frame. In practice, it is problematic in a garbage-collected environment to deal with pointers into the interior of objects.

5. The task pointers area contains pointers to the task structures for the procedure. Remember that task structures are used to build up defer lists when tasks suspend on empty memory cells. The global work queue also contains task structures for tasks that are ready to execute. Like barriers, the space for task structures could be allocated directly as part of the frame since the lifetimes of the structures match that of the frame, but for the same reason, to avoid pointers into the interior of objects, we allocate them separately at the time of function invocation.

<table>
<thead>
<tr>
<th>Activation Frame Data Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Caller Link</strong></td>
</tr>
<tr>
<td>full</td>
</tr>
<tr>
<td>full</td>
</tr>
<tr>
<td>Pointer to caller’s activation frame</td>
</tr>
<tr>
<td>full</td>
</tr>
<tr>
<td>Return address</td>
</tr>
<tr>
<td>full</td>
</tr>
<tr>
<td>full</td>
</tr>
<tr>
<td>Pointer to caller’s task structure</td>
</tr>
<tr>
<td>Argument</td>
</tr>
<tr>
<td>full</td>
</tr>
<tr>
<td>Argument 1</td>
</tr>
<tr>
<td>References</td>
</tr>
<tr>
<td>full</td>
</tr>
<tr>
<td>full</td>
</tr>
<tr>
<td>Argument a</td>
</tr>
<tr>
<td>Local</td>
</tr>
<tr>
<td>full</td>
</tr>
<tr>
<td>Local variable 1</td>
</tr>
<tr>
<td>References</td>
</tr>
<tr>
<td>full</td>
</tr>
<tr>
<td>full</td>
</tr>
<tr>
<td>Local variable 1</td>
</tr>
<tr>
<td>Barrier</td>
</tr>
<tr>
<td>full</td>
</tr>
<tr>
<td>Barrier 1</td>
</tr>
<tr>
<td>Pointers</td>
</tr>
<tr>
<td>full</td>
</tr>
<tr>
<td>full</td>
</tr>
<tr>
<td>Barrier b</td>
</tr>
<tr>
<td>Task</td>
</tr>
<tr>
<td>full</td>
</tr>
<tr>
<td>full</td>
</tr>
<tr>
<td>Task 1</td>
</tr>
<tr>
<td>Pointers</td>
</tr>
<tr>
<td>full</td>
</tr>
<tr>
<td>full</td>
</tr>
<tr>
<td>Task t</td>
</tr>
</tbody>
</table>

5.1.4 Closures

Closures are data structures used to represent functions. They are created directly using $\lambda$-syntax or via currying. In memory, a closure consists of at least three slots: one to hold the *arity* of the closure (number of arguments required for invocation), one to hold the entry point address to the code, and one to hold the number of arguments stored in the
closure. The rest of the closure contains the curried arguments. In the following examples, the expression on the top line results in the closure below it:

<table>
<thead>
<tr>
<th></th>
<th>$f = \lambda a \ b \ c. \ E_{body}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>arity</td>
<td>full 3</td>
</tr>
<tr>
<td>entry point</td>
<td>$a_{body}$</td>
</tr>
<tr>
<td>argument count</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$f \times y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>arity</td>
<td>full 1</td>
</tr>
<tr>
<td>entry point</td>
<td>$a_{body}$</td>
</tr>
<tr>
<td>argument count</td>
<td>2</td>
</tr>
<tr>
<td>argument</td>
<td>Reference to $x$</td>
</tr>
<tr>
<td>argument</td>
<td>Reference to $y$</td>
</tr>
</tbody>
</table>

The expression and closure on the left represent a function $f$ of arity 3, where the address of the entry point is $a_{body}$. The expression on the right is a curried application of $f$ to two arguments. A new closure is created as the value of the expression. The arity field of the new closure is computed by subtracting the number of arguments (2) from the original arity (3), the entry point remains unchanged, and references to the arguments (in left-to-right order) are placed in the subsequent slots. This new closure only needs one more argument in order to invoke the body of the function. The original closure still needs three.

### 5.2 Compiling Simple, Primitive, and Data Expressions

This section describes how each $\lambda^*$ expression is compiled into SMT instructions. The compilation rules are given in terms of two translation functions: $TE$ (Translate Expression) and $TS$ (Translate Statement). A third translation function $TP$ (Translate Program) is given in Section 5.6. The $TP$ function drives the compilation process in a syntax-directed fashion, based on the syntax of $\lambda^*$ in Figure 2-5. The input to a translation function is a syntactic program element: an expression for $TE$, a statement for $TP$, and an entire $\lambda^*$ program to $TP$. The output of a translation function is a list of SMT instructions. These lists of instructions are appended to form longer lists for larger syntactic elements, and in fact, the final result of the compilation process is a single list of SMT instructions.

The translation functions are specified in an informal pseudo-code that borrows from the syntax of pH. For example, we borrow "[ ]" for list notation and ++ for list concatenation. In addition, we give an example of the code generated for each translation function in the following format:
The compilation scheme sets aside several GP registers for special purposes. These registers do not conflict with any other registers, and they are given the following names:

- **rV** The *value* register. The instructions generated for an SMT expression, when executed, always place the value of the expression (never a reference) in rV.

- **rF** The *frame pointer* register. Contains the address of the activation frame of the procedure currently executing on the processor.

- **rB** The *current barrier* register. Contains the address of the barrier structure corresponding to the current pre-region.

- **rL** The *link* register. Used to store return addresses on procedure calls.

- **rS** The *scratch memory* register. Points to a private per-processor area of scratch memory that can be managed without the use of the allocate instruction. This register is initialized before code starts executing on a processor.

- **rSS** The *scratch memory size* register. Contains an integer indicating the number of scratch memory locations in use.

Though not apparent in the syntax, identifiers in a λ* program are either local or top-level. Local identifiers are those defined within a procedure while top-level identifiers are defined outside the scope of any procedure. As we'll see in Section 5.6, a program consists of a group of functions bound to top-level identifiers. The translation functions make use of identifiers to represent the addresses of references to values. For a local identifier, the reference is stored in the procedure activation frame. For a top-level identifier, the reference is a statically generated global data structure. A pass in the compiler following SMT code generation converts symbolic local identifiers into integer offsets in the activation frame. For a top-level identifier, the symbol is resolved into an address at program link-time.
5.2.1 Constants

A constant is the simplest $\lambda^*$ expression to compile to SMT. We simply move the value of the constant into register $rV$.

$$\text{T}_{E}[c] = [ \text{move}(rV, c) ]$$

The following code results for the compilation of the expression “1”:

$$\begin{align*}
1 \\
\text{move}(rV, 1)
\end{align*}$$

5.2.2 Identifiers

To fetch the value of an identifier, we must make sure that the value has been computed using a touch instruction. Remember that the symbol for the identifier represents the address of a reference to the identifier. This is the rule to generate code for an identifier expression:

$$\text{T}_{E}[x] = [ \text{touch}(\@x) \{ rF, rB \}, \text{load}(rV, \@x) ]$$

The touch instruction in this sequence tests the presence status of the memory cell for $x$. When the touch succeeds, then the value is guaranteed to be present, so the load fetches into register $rV$. Notice how the use of indirect addressing keeps the code sequence very compact. Also, notice the live-state annotation on the touch instruction. This particular annotation with $rF$ and $rB$ is so common that to avoid clutter, we'll abbreviate it with $\ast$.

Here is the compilation of the expression “$x$”. We assume that symbolic identifiers correspond to local variables and that they are converted into activation frame offsets in some later pass in the compiler.

$$\begin{align*}
x \\
\text{touch}(\@rF[x]) \ast \\
\text{load}(rV, \@rF[x])
\end{align*}$$

5.2.3 Primitive Functions

Primitive functions represent machine operations (arithmetic, logic, etc.), not user-defined $\lambda$-abstractions. Unlike user functions, primitive functions are strict in all arguments. This
means that all arguments must be computed before the primitive is executed. The compiler implements this behavior using touch instructions. The rule for compiling primitives, where $PF^n$ is a primitive of arity $n$, is:

$$\text{TE}_{PF^n(x_1, \ldots, x_n)} = [\text{touch}(x_1) \ast, \ldots, \text{touch}(x_n) \ast, \text{load}(r_1, x_1), \ldots, \text{load}(r_n, x_n), pf^n(r_V, r_1, \ldots, r_n) ]$$

The following code results from compiling a binary integer addition expression. Identifiers $x$ and $y$ are local variables within the frame.

\[
\begin{align*}
x + y \\
touch(@\text{rF}[x]) \ast \\
touch(@\text{rF}[y]) \ast \\
\text{load}(r_1, @\text{rF}[x]) \\
\text{load}(r_2, @\text{rF}[y]) \\
\text{plus}(r_V, r_1, r_2)
\end{align*}
\]

The code is structured so all the touch instructions execute first, before the value of any identifier is loaded into a register. This approach limits the amount of live state that accumulates in the registers, so that we don't have to preserve it on a suspension. After the touch instructions execute, it is certain that $x$ and $y$ have been computed, so we simply load their values from the frame, execute the plus primitive, and deposit the result in $r_V$.

### 5.2.4 Memory Expressions: allocate, iFetch, and mFetch

In the $\lambda^*$ syntax, allocate, iFetch, and mFetch are treated as separate expressions, not lumped with the primitive functions, because their behavior depends on side-effects in the program. The argument to allocate is the number of memory cells to allocate. The result is the address of a fresh block of memory. This expression maps almost directly to the SMT instruction we have defined for this purpose:

$$\text{TE}[\text{allocate}(x)] = [\text{touch}(x) \ast, \text{load}(r_V, x), \text{allocate}(r_V, r_V) ]$$

As expected, we access the value of the identifier, the number of cells to allocate, with a touch/load combination.

The argument to an iFetch or mFetch expression is an identifier whose value is the
address of a memory cell. The purpose of these expressions is to extract the value of the cell, so they must synchronize twice. The first synchronization ensures that the value of the identifier has been computed (i.e., that the memory cell has been allocated). The second ensures that the contents of the memory cell have been computed. The following rule achieves this effect for iFetch:

$$\text{TE}[\text{iFetch}(x)] = [\text{touch}(\@x) \ast, \text{load}(rV, \@x), \text{touch}(rV) \{rV, rF, rB\}, \text{load}(rV, rV)]$$

Notice that the second touch makes sure to preserve $rV$ (as well as the usual $rB$ and $rF$). This allows the contents of the register to be used in the subsequent instructions and avoids the recalculation of the address in $rV$. For mFetch, only the last two instructions change. They are be replaced by a single take($rV, rV$). To implement the M-Structure protocol properly, take combines a synchronization operation and a memory load.

An example of the code generated by this rule follows:

\[
i\text{Fetch}(x) \\
touch(\@rF[x]) \ast \\
\text{load}(rV, \@rF[x]) \\
touch(rV) \{rV, rF, rB\} \\
\text{load}(rV, rV)
\]

5.2.5 Constructors

Constructors are non-strict with respect to their contents. A constructor in memory consists of as many slots as it has fields, plus one extra slot: the tag used to distinguish one constructor from another. The tag is simply an integer and is computed when the constructor is created. Its value depends on the static structure of the constructor's type. For example, the type Tree from Section 2.2,

\[
data \text{Tree } a = \text{Leaf } a \mid \text{Node } (\text{Tree } a) (\text{Tree } a)
\]

contains the constructors Leaf and Node. The compiler assigns a tag of 0 to the Leaf and a tag of 1 to the Node. The tag slot is considered the 0th field of the constructor.
The rule to compile a constructor with $k$ fields is the following:

$$\text{TE}[CN^k(x_1, \ldots, x_k)] = [\text{allocate}(rV, k + 1), \text{store}(rV[0], \text{tag}),$$

$$\text{load}(r1, x_1), \text{store}(rV[1], r1), \ldots, \text{load}(r1, x_k), \text{store}(rV[k], r1) ]$$

The code generated for a constructor expression does not include any synchronizing (suspensive) instructions like touch, so it is certainly non-strict. The size argument to the allocate instruction is $k + 1$ to accommodate the extra slot required for the tag of the constructor. Also, note that references, not values, are stored in the body of the constructor. We cannot store values because it would make the computation of the constructor dependent on its contents. It is important to note that unlike function applications, constructor applications are not curried.

Here is the code that is generated for `Node`, a constructor with two fields:

```
Node(x, y)
allocate(rV, 3)
store(rV[0], 1)
load(r1, rF[x])
store(rV[1], r1)
load(r1, rF[y])
store(rV[2], r1)
```

### 5.2.6 Projections

The primitive projection function $\text{Prj}^k$ is used to extract the $k^{th}$ field of a constructor. Like `iFetch` and `mFetch`, this implies that projections must synchronize twice: once to make sure that the constructor has been computed and once to make sure that the field of the constructor has been computed. The only exception is when fetching the tag of the constructor (the $0^{th}$ field). The tag is always known to be available because it is computed when the constructor is allocated. We give these two cases of the projection rule separately.

For the tag field:

$$\text{TE}[\text{Prj}^0(x)] = [\text{touch}(@x) \ast, \text{load}(rV, @x), \text{load}(rV, rV[0]) ]$$

85
For data fields, where $k > 0$:

$$\text{TE}[\text{Prj}^k(x)] = [\text{touch}(\@x) \ast, \text{load}(rV, \@x), \text{touch}(\@rV[k]) \{rV, rF, rB\}, \text{load}(rV, \@rV[k])]$$

Here is an example of a projection on the first data field:

$$\text{Prj}^1(x)$$

- touch(\@rF[x]) \ast
- load(rV, \@rF[x])
- touch(\@rV[1]) \{rV, rF, rB\}
- load(rV, \@rV[1])

### 5.3 Compiling Case Expressions

Unlike the previous expressions, **Case** requires the implementation of control-flow. The first argument of a **Case** is an integer expression (the *discriminant*) that determines which branch to execute. The rest of the arguments are the expressions for the branches. In the SMT code, we represent code addresses symbolically with *labels*. As in most machine languages, a label can appear in an instruction sequence or as a parameter to an instruction. Labels are eventually resolved into addresses by the linker when we generate an executable program. We generate fresh labels in the compiler with the **genLabel** routine.

In the rule below, the expressions for the branches are compiled using recursive calls to **TE**. Labels are added to the beginning of each branch. A jump to the end of the **Case** code sequence is added to all but the last branch since control can flow directly from the last branch to the end of the code sequence. Then, the discriminant code and the code for all the branches are concatenated into a single instruction sequence.

$$\text{TE}[\text{Case}(x, E_0, \ldots, E_k)] =$$

- \{ $l_{\text{done}}, l_0, \ldots, l_k = \text{genLabel}, \text{genLabel}, \ldots, \text{genLabel}$
- $\iota_{s_0} = [l_0 : ] ++ \text{TE}[E_0] ++ [\text{jump}(\text{Always}, l_{\text{done}})]$
- $\iota_{s_1} = [l_1 : ] ++ \text{TE}[E_1] ++ [\text{jump}(\text{Always}, l_{\text{done}})]$
- ...
- $\iota_{s_k} = [l_k : ] ++ \text{TE}[E_k]$
- in
- [touch(\@x) \ast, \text{load}(rV, \@x), \text{switch}(rV, l_0, \ldots, l_k)] ++$
- \iota_{s_0} ++ \iota_{s_1} ++ \ldots ++ \iota_{s_k} ++ [l_{\text{done}} : ]$
- \}

86
Too keep the example simple, the discriminant in the following Case expression chooses between only two expressions, y or 151.

\[
\text{Case}(x, y, 151)
\]

\[
touch(@rF[x])
\]
\[
load(rV, @rF[x])
\]
\[
\text{switch}(rV, 1.0, 1.1)
\]

1.0:

\[
touch(@rF[y])
\]
\[
load(rV, @rF[y])
\]
\[
\text{jump}(Always, 1.\text{done})
\]

1.1:

\[
\text{move}(rV, 151)
\]

1.\text{done}:

The sequence is structured so that when control flow reaches label 1.\text{done}, the value of the Case expression will be in register rV.

### 5.4 Compiling λ-Abstractions and Function Calls

In this section, we focus on λ-abstractions and function applications exclusively. The two compilation rules are closely related since they generate code for the two halves of the function calling convention. The rule for λ-abstractions handles the callee side of the calling convention while the rule for application expressions handles the caller's responsibilities.

#### 5.4.1 λ-Abstractions

The code generated for λ-abstractions is complex because the calling convention must include full support for currying. Due to currying, a function of n arguments may eventually execute as result of an application ranging from n arguments down to a single argument. For efficiency, we also want to support passing a limited number of arguments in registers, but we must support function calls that exceed that limit. We divide the presentation below into three parts. First, we present the calling convention for pH functions. Then we present the rule for generating code based on this convention. Finally, we present the code generated for a particular λ-expression.
Calling Conventions

We use the following registers to pass information from caller to callee:

- \( r1 \ldots r8 \) Argument registers (GP registers)
- \( rV \) Pointer to callee closure
- \( rL \) Caller return address
- \( rF \) Caller frame pointer
- \( rT \) Caller task structure
- \( rS \) Pointer to overflow area
- \( rSS \) Number of overflow arguments

The choice of eight registers for the arguments is common in the calling conventions of real systems. All registers in the SMT register files are considered *caller-saved* by the compilation strategy. Code in the caller must explicitly save and restore the values of registers that are live across procedure calls. This convention is essential due to the possibility of suspension in a callee. If a callee suspends, there is no guarantee that the next task to be scheduled will be in the caller function.

Arguments are placed in registers in an interesting way to simplify the structure of the code. Currying proceeds from left-to-right, so arguments at a call site are placed in registers in right-to-left order beginning at register \( r1 \). This guarantees that the right-most argument, the last argument needed to invoke the body of the function, will always appear in \( r1 \). Recall the function \( f \) in Section 5.1.4:

\[
 f = \lambda a \ b \ c \cdot \ E_{body}
\]

If this function is invoked as "\( f \ x \ y \ z \)" , then \( r1 \) will contain a reference to \( z \) , \( r2 \) a reference to \( y \) , and \( r3 \) a reference to \( x \).

If there are more than eight arguments in an application, the excess arguments are placed in the scratch memory of the processor, accessed via \( rS \). Given the way arguments are assigned to registers, all arguments except the right-most eight will be passed in memory. Notice that we must take into account the possibility of currying in all cases. Suppose a closure is created from a function of arity 12 by currying it with 2 arguments. Later the resulting arity 10 closure is applied to 10 arguments. When the body of the function is invoked, its arguments appear in three different places. The first and second arguments (counting from the left) appear in the closure data structure; the next two arguments are in the scratch memory; and the last eight arguments are in the argument registers.
Code Generation

The rule to generate code for lambda abstractions follows. To understand its behavior, it is important to keep in mind the calling convention just presented and the structure of activation frames from Section 5.1.3. The compilation rule is divided into three sections:

1. The first section generates the code to allocate and initialize the activation frame for the function. Recall that frames are divided into five areas: caller link, arguments, local references, barrier pointers, and task pointers. Each of these areas is initialized separately, and this is what consumes the majority of the rule. References for arguments, for instance, are copied from registers, scratch memory, or the incoming closure into the frame. Locations for local variables are allocated independently, and references to these locations are stored into the frame. Likewise, space for barriers and tasks is allocated and pointers to these structures are stored in the frame.

2. The second section generates code for the body of the function. This is particularly easy since it only involves a recursive invocation of the TE rule.

3. The last section links the frame setup code, the function body code, and a small code fragment to return control to the caller into a single code sequence. This code sequence constitutes the full code for the lambda abstraction.
\[ TE[\lambda x_{1} \ldots x_{n}.E_{\text{body}}] = \]

> The entire code for the function.

\[
\{ \text{fullCode} = \text{setupFrame} + \text{body} + \text{return} \}
\]

> Generate code to initialize the activation frame.

\[
\begin{align*}
\text{setupFrame} &= \text{callerLink} + \text{argsMemRegs} + \text{argsClosure} + \text{setupStructures} \\
\text{callerLink} &= [\text{allocate}(r9, fsize), \text{store}(r9[0], rF), \text{move}(rF, r9), \text{store}(rF[1], rL), \text{store}(rF[2], rT)]
\end{align*}
\]

> Get the arguments from scratch memory and registers.

\[
\begin{align*}
\text{argsMemRegs} &= \text{testArity} + \text{argCode}_{n} + \ldots + \text{argCode}_{1} \\
\text{arg}_{1}, \ldots, \text{arg}_{n} &= \text{genLabel}, \ldots, \text{genLabel} \\
\text{testArity} &= [\text{load}(r9, rV[0]), \text{switch}(r9, err, arg_{1}, \ldots, arg_{n})] \\
\text{argCode}_{n} &= [\text{arg}_{n} : \text{load}(r10, rS[n-8]), \text{store}(rF[3], r10)] \\
\text{argCode}_{n-1} &= [\text{arg}_{n-1} : \text{load}(r10, rS[n-9]), \text{store}(rF[4], r10)] \\
\ldots \\
\text{argCode}_{0} &= [\text{arg}_{0} : \text{load}(r10, rS[1]), \text{store}(rF[n-9+3], r10)] \\
\text{argCode}_{0} &= [\text{arg}_{8} : \text{store}(rF[n-8+3], r8)] \\
\ldots \\
\text{argCode}_{1} &= [\text{arg}_{1} : \text{store}(rF[n-1+3], r1)]
\end{align*}
\]

> Get the arguments from the closure.

\[
\begin{align*}
\text{argsClosure} &= \text{testArity2} + \text{clCode}_{n-1} + \ldots + \text{clCode}_{0} \\
\text{cl}_{0}, \ldots, \text{cl}_{n-1} &= \text{genLabel}, \ldots, \text{genLabel} \\
\text{testArity2} &= [\text{switch}(r9, err, cl_{n-1}, \ldots, cl_{0})] \\
\text{clCode}_{n-1} &= [\text{cl}_{n-1} : \text{load}(r9, rV[2]), \text{store}(rF[3], r9)] \\
\text{clCode}_{n-2} &= [\text{cl}_{n-2} : \text{load}(r9, rV[3]), \text{store}(rF[4], r9)] \\
\ldots \\
\text{clCode}_{1} &= [\text{cl}_{1} : \text{load}(r9, rV[n]), \text{store}(rF[n+1], r9)] \\
\text{clCode}_{0} &= [\text{cl}_{0} :]
\end{align*}
\]

> Initialize additional structures: local variables, barriers, tasks.

\[
\begin{align*}
\text{setupStructures} &= \text{makeLocals} + \text{makeBarriers} + \text{makeTasks} \\
\text{makeLocals} &= [\text{allocate}(r9, locals), \text{store}(rF[n+3], r9), \text{plus}(r9, r9, 1), \text{store}(rF[n+4], r9), \ldots, \text{store}(rF[n+locals+2], r9)] \\
\text{makeBarriers} &= [\text{allocate}(r9, barriers*3), \text{store}(rF[n+locals+3], r9), \text{plus}(r9, r9, 3), \text{store}(rF[n+locals+4], r9), \ldots, \text{store}(rF[n+locals+barriers+2], r9)] \\
\text{makeTasks} &= [\text{allocate}(rT, sizetasks), \text{store}(rF[n+locals+barriers+3], rT), \text{plus}(r9, rT, sizetasks), \text{store}(rF[n+locals+barriers+4], r9), \ldots, \text{plus}(r9, r9, sizetask_{tasks-1}), \text{store}(rF[n+locals+barriers+tasks+2], r9)]
\end{align*}
\]

> Generate code for the body of the function.

\[
\begin{align*}
\text{body} &= TE[E_{\text{body}}] \\
\end{align*}
\]

> Generate code to return from the function.

\[
\begin{align*}
\text{return} &= [\text{load}(rT, rF[2]), \text{load}(rL, rF[1]), \text{load}(rF, rF[0]), \text{jump}(Always, rL)] \\
in \\
\text{fullCode} \\
\end{align*}
\]

\[
\begin{align*}
\text{where} \ n &= \text{number of arguments for } \lambda\text{-expression} \\
\text{locals} &= \text{number of local variables in } E_{\text{body}} \\
\text{barriers} &= \text{number of barriers in } E_{\text{body}} \\
fsize &= \text{size of activation frame, } n + \text{locals} + \text{barriers} + 3 \\
tasks &= \text{number of tasks in } E_{\text{body}} \\
sizetask_{n} &= \text{size of structure for task}_{n} \\
sizetasks &= \sum_{i=1}^{tasks} sizetask_{i} \\
\text{err} &= \text{standard error handling code (not shown)}
\end{align*}
\]
The rule shows how to generate code for the most general case, where \( n > 8 \) and the function body contains several tasks and barriers. Obvious optimizations are made when, for example, there is only one argument: the switch instructions are unnecessary. The values of \( n \), locals, etc., are all compile time constants and are computed by analyzing the \( \lambda \)-expression.

**Example**

The example presented below is more straightforward than the most general case of a \( \lambda \)-abstraction because it consists of only two arguments, no local variables, no barriers, and a single task-structure. We present the code in fragments following the structure of the compilation rule. The first fragment corresponds to the callerLink code. This code is used to allocate the activation frame and to save three pieces of information from the caller: the caller's frame pointer in register \( rF \), the return address in register \( rL \), and the caller's task structure pointer in \( rT \).

\[
\begin{align*}
\lambda \ a \ b. \ & a + b \\
allocate(r9, 6) \\
store(r9[0], rF) \\
move(rF, r9) \\
store(rF[1], rL) \\
store(rF[2], rT)
\end{align*}
\]

The allocate instruction allocates free space for the frame on the heap. The size of the frame for this function is 6 because it consists of three slots for the caller link information (caller frame pointer, return address, and task structure), two slots for the references to \( a \) and \( b \), and one slot for a pointer to a single task structure. Notice that we use register \( r9 \) as a temporary register since no parameter is ever passed in that register. The code stores the caller's frame pointer into the new activation frame, and updates the value of the frame pointer register \( rF \) with the pointer to the new frame. Subsequent accesses to the new frame are made through \( rF \).

The next sequence of instructions transfers the argument references from the argument registers and closure into the frame. It corresponds to the code fragments argsMemRegs and argsClosure in the compilation rule. The code is a little tricky here due to currying. As we noted, the code for the function might have been invoked as result of an application
to one or two arguments, so the arguments can be distributed between the registers and
the closure in \( rV \) in two ways. If the function is invoked with two arguments directly, then
all arguments will be in registers and none will be in the closure. If the function is curried
over one argument and then applied to another argument later, then one argument will be
in the registers, and the other in the closure.

The piece of information that describes how the arguments are organized is the arity
field of the closure in \( rV \). Located at offset 0 in the closure, the arity can range in value
from 1, for a maximally curried closure, to \( n \), the arity of an uncurried closure. We use this
value as follows:

\[
\begin{align*}
\text{load}(r9, rV[0]) \\
\text{switch}(r9, \text{err}, \text{arg}.1, \text{arg}.2) \\
\text{arg}.2: \\
\quad \text{store}(rF[a], r2) \\
\text{arg}.1: \\
\quad \text{store}(rF[b], r1) \\
\quad \text{switch}(r9, \text{err}, c1.1, c1.0) \\
\text{c1.1:} \\
\quad \text{load}(r9, rV[3]) \\
\quad \text{store}(rF[a], r9) \\
\text{c1.0:}
\end{align*}
\]

The first switch uses the arity to determine how many arguments are in registers. If the arity
of the closure is 1, for instance, then control passes to label \( \text{arg}.1 \), and the value of register \( r1 \)
is moved into the activation frame. The second switch uses the arity to determine how many
arguments are in the closure. Again, if the arity is 1, then control passes to label \( c1.1 \) since
only one argument needs to be moved from the closure to the frame. Notice how the code
is organized so that a transfer of control to one of the earlier switch alternatives causes the
code for all the succeeding alternatives to be executed. In this way, we can make the code
more compact since the case for extracting seven registers from arguments also makes uses
of the code for extracting six registers, five registers, etc. Often, the behavior of a function
is simple (leaf and non-suspensive) so arguments can remain in registers throughout an
invocation. To keep the code generator simple, we leave optimizations like this to other
passes of the compiler.

The next piece of code (the \texttt{makeLocals} code fragment in the compilation rule) constructs
the area of the activation frame for local variables. First it allocates enough storage for all

92
the locals, and then it puts references to these individual memory cells in the frame. Our example has no local variables, so we skip this code. We also skip the makeBarriers code since our example has no barriers. Finally, we reach the code that initializes the pointers to the task structures. This code corresponds to the makeTasks fragment in the compilation rule. The λ-expression in our example only needs a single task structure, to be used when the code synchronizes with the values of a and b in the computation of the primitive function application a + b. This task structure needs to store only three pieces of information: a program counter, the value of rF, and the value of rB. Only two instructions are required to allocate the task structure and to link it to the activation frame:

```
allocate(rT, 3)
store(rF[task1], rT)
```

Notice that as a side-effect of the allocation, a pointer to the first task structure is placed in rT. This ensures that rT is initialized by the time the code for the body of the function starts executing.

The code for the body of the function, a + b, follows the frame initialization code. It's simply a primitive application:

```
touch(@rF[a]) *
touch(@rF[b]) *
load(r1, @rF[a])
load(r2, @rF[b])
plus(rV, r1, r2)
```

After the body produces a value in register rV, the function returns to its caller with the return sequence of instructions:

```
load(rT, rF[2])
load(rL, rF[1])
load(rF, rF[0])
jump(Always, rL)
```

The first instruction restores the caller’s task structure. The second loads the return address into rL. The third restores the caller’s frame pointer, and finally, the last jumps back into the caller’s code.
Lambda-Lifting

The syntax of $\lambda^*$ allows functions to be bound to identifiers at any level of the lexical hierarchy. As discussed in Section 3.2, the front-end of the pHc compiler performs a lambda-lifting transformation [28], so that $\lambda$-abstractions are only bound to identifiers at the top-level of a program. This means that curried applications are the only expressions that result in the creation of closures at run-time. Lambda-lifting also simplifies the management of lexical scopes because free variables are passed as explicit function arguments. We discuss function application in the next section.

Summary

The code for a $\lambda$-abstraction needs to support the language features of currying and parallelism. Due to currying, a function may be invoked in many different ways. In some cases, all arguments may be in registers while in others, arguments might be spread out over a closure, the SMT processor scratch memory, and registers. Due to parallelism, a function must allocate and create links to storage separate from the activation frame for local variables, task structures, and barriers. Despite the complexity of pH function calls compared to other languages, like C, there are opportunities for optimization. For example, variables that are not live across suspensions might always be stored in registers, and task structures might be shared by computations that are disjoint, like the branches of a conditional. In the code generator, however, we use a compilation rule that handles the most general case of a $\lambda$-abstraction and worry about optimizations separately.

5.4.2 Function Applications

The code for a $\lambda$-abstraction performs much of the work needed for a function call, but higher-order functions and currying still make it tricky to compile all but the simplest applications. An application expression consists of a function identifier and a number of argument identifiers. As mentioned earlier, there are two types of function identifiers: top-level and local. Both kinds of identifiers are bound to closures, but the contents of the closure of a top-level identifier are known at compile-time (they are generated by the TP rule in Section 5.6). This is important because the code to compile an application expression needs the arity and entry-point information from the closure.
Application expressions also differ depending on the number of arguments present at the call site. There are three cases. If the number of arguments at the call site is less than the arity of the function being applied, then the application is a curried application and results in the creation of a closure. If the number of arguments is exactly equal to the arity of the function, then the application is a full application, and the function is invoked. Last, if the number of arguments at the call site exceeds the arity of the function, then the application is a compound application. In a compound application, an unknown number of intermediate closures will be created and immediately applied, incrementally consuming the arguments at the call site. We discuss compound applications in more detail later in this section.

The code generated by the compiler to perform a function application proceeds in three steps. The first step is to determine the arity of the function being applied. Based on the arity, the compiler can determine whether the application is curried, full, or compound. If the function being applied is a top-level function, the arity can be determined statically, and the compiler can generate a curried, full, or compound calling sequence directly. However, if the function is not a top-level function, the arity cannot be known statically. In this case, the compiler must generate code to determine the arity at run-time using the first slot of the closure for the function. In addition, all three different code sequences to handle the curried, full, and compound application cases must be emitted. The compiler generates code that uses the arity from the closure to dispatch to the appropriate sequence dynamically.

The next step in the calling sequence is to package the arguments. In the case of a full application, the calling conventions described in the compilation of λ-abstractions (Section 5.4.1) describe exactly how to pass parameters. Briefly, the rightmost eight parameters are passed in registers r1 to r8, and the rest are passed in scratch memory. If the application is a curried application, and the callee is top-level, then code to create a new closure is emitted directly. If the callee is not top-level, then the compiler emits code that invokes an auxiliary run-time system routine called makeClosureRTS. This RTS routine expects the current closure in register rV, a return address in rL, and the arguments of the application in scratch memory. It returns to address rL with a freshly allocated closure in rV.

A similar strategy is used for all compound applications. The compiler delegates the work of performing the compound application to the RTS routine compoundClosureRTS. Like makeClosureRTS, this routine expects the current closure in rV, a return address in rL,
and the arguments at the call site in scratch memory. Curried and compound applications are much less common than full applications, so we expect that the cost of invoking the RTS will not affect performance significantly. This is a small price to pay for substantially reducing the amount of code at a call site.

The final step in the calling sequence is transferring control to the callee. In a full application of a top-level callee, the name of the entry point is known statically, so the compiler emits a jump instruction with the correct symbolic address. In a full application of a non-top-level callee, the entry point must be extracted from the closure for the callee and the compiler emits a jump instruction that performs an indirect control transfer.

The following compilation rule handles all six different kinds (the product of curried, full, and compound with top-level and local callees) of applications. It uses the compiler routines `topLevel` and `arity` to discover static properties of the callee.

$$\text{TE}[f \ x_1 \ldots x_n] =$$

```plaintext
if (topLevel(f))
  then if (n < arity(f))
    then [ allocate(rV, n + 3), store(rV[0], arity(f) - n), store(rV[1], f.entry_label),
           store(rV[2], n), load(r1, x1), store(rV[3], r1), ..., load(r1, xn), store(rV[n + 2], r1) ]
    else if ((n == arity(f)) and (n ≤ 8))
      then [ load(rV, f), load(r1, xn), ..., load(rn, x1), jump(Always, f.entry_point, rL) ]
    else if ((n == arity(f)) and (n > 8))
      then [ load(rV, f), load(r1, x1), store(rS[0], r1), ..., load(r1, xn-8),
             store(rS[n - 8 - 1], r1), load(r1, xn), ..., load(r8, xn-7),
             jump(Always, f.entry_point, rL) ]
      else [ load(rV, f), load(r1, x1), store(rS[0], r1), ..., load(r1, xn), store(rS[n - 1], r1),
               move(rSS, n), jump(Always, compoundCallRTS, rL) ]
  else [ load(rV, f), load(r1, xi), store(rS[0], r1), ..., load(r1, x), store(rS[n - 1], r1),
             move(rSS, n), jump(Always, compoundCallRTS, rL) ]
```

where $n$ = number of arguments at call site

$\text{arity}(f) =$ compiler function giving arity of $f$

$\text{topLevel}(f) =$ compiler predicate indicating if $f$ is known statically
Of the six varieties, curried and full applications involving top-level function identifiers are the simplest to compile. The following examples are based on a top-level function \( f \) of two arguments. The compiler knows the arity of \( f \) statically, so the following curried application generates a closure directly. As usual, we assume \( x \) is a local identifier:

\[
\begin{align*}
\text{allocate}(rV, 4) \\
\text{store}(rV[0], 1) \\
\text{store}(rV[1], f\text{-entry\_point}) \\
\text{store}(rV[2], 1) \\
\text{load}(r1, rF[x]) \\
\text{store}(rV[3], r1)
\end{align*}
\]

This code corresponds to the code generated by the compilation rule when the conditions \( \text{topLevel}(f) \) and \( n < \text{arity}(f) \) are both true. The \text{allocate} instruction creates a new closure of four slots: arity, entry point, curried arguments count, and reference to \( x \). The rest of the instructions initialize the closure. It is important to note that because closures are non-strict, the parameter \( x \) is represented with a reference, not its value.

When the application expression corresponds to a known, full-arity application—that is when \( \text{topLevel}(f) \) and \( n == \text{arity}(f) \) are true—the following code is emitted:

\[
\begin{align*}
\text{load}(rV, f) \\
\text{load}(r1, rF[y]) \\
\text{load}(r2, rF[x]) \\
\text{jump}(\text{Always}, f\text{-entry\_point}, rL)
\end{align*}
\]

The first load instruction is used to get a pointer to the closure for \( f \). The closure for \( f \) is known to be computed (it is emitted as a static data block by the \text{TP} rule), so there is no need for a touch instruction. The next two load instructions initialize the argument registers. Since procedure calls are non-strict, only references are passed in the argument registers. Hence, there is no indirection in the addressing. Finally, the jump instruction transfers control directly to the label \( f\text{-entry\_point} \), while storing the return address in \( rL \). The name of this entry point label can be derived, by convention, from the name of the function \( f \). As we shall see, the entry point is generated by the \text{TP} rule.

The final case to consider for top-level functions is a compound application. We discuss
this case for both top-level and local function identifiers in a separate section below.

When the arity of a function is not known at compile-time, the code must determine it dynamically. This is the code generated for an application of \( z \), a local variable, to two arguments:

\[
\begin{align*}
z \ x \ y \\
touch(@rF[z]) & \ast \\
load(rV, @rF[z]) & \\
load(r1, rV[0]) & \\
jump((r1 == 2), 1\_full) & \\
load(r2, rF[x]) & \\
store(rS[0], r2) & \\
load(r2, rF[y]) & \\
store(rS[1], r2) & \\
move(rSS, 2) & \\
move(rL, l\_done) & \\
jump((r1 > 2), makeClosureRTS) & \\
jump(Always, compoundCallRTS) & \\
\end{align*}
\]

1\_full:
\[
\begin{align*}
load(r1, rF[y]) & \\
load(r2, rF[x]) & \\
jump(Always, @rV[1], rL) & \\
\end{align*}
\]

1\_done:
\[
\begin{align*}
load(r1, rF[y]) & \\
load(r2, rF[x]) & \\
jump(Always, @rV[1], rL) & \\
\end{align*}
\]

First, the code makes sure that the value of \( z \), a closure, has been computed. An application is strict only in the function being applied. Then, the code loads the value of the arity slot of the closure into \( r1 \). This value is compared to the number of arguments in the application expression (2). If the arity of the closure equals the number of arguments, then the application is a full application. The code jumps to the 1\_full label, loads the registers with references to \( x \) and \( y \) according to the calling convention, and jumps to the entry point of the function, extracted from offset 1 in the closure.

If the arity of the closure is greater than the number of arguments in the application expression, the application is a curried application and a new closure must be created. Creating this new closure is more complicated than in the case of a top-level function because the size of the existing closure is not known statically. The process involves additional
code to compute the new size, copy the old closure, etc. We use the RTS to to perform this operation. The makeClosureRTS routine expects the current closure in rV and the application arguments in the scratch memory. It returns and new closure in rV to the address in rL, which the code sets to 1-done.

The last case to consider is that the arity of the closure is less than the number of arguments at the call site. We handle this case with the RTS primitive compoundCallRTS.

**Compound Applications**

A compound application arises when the number of arguments at a call site is greater than the arity of the function being applied. In this section, we describe such compound applications, and we sketch the behavior of the RTS routine used to handle them, compoundCallRTS. It might seem incorrect to provide more arguments to a function than it can accept, but it can be perfectly reasonable and type-correct in a higher-order language. Suppose a function with the type signature `Int -> Int -> Int -> Int` is passed as argument to the following function:

\[
\lambda g . \ g \ 1 \ 2 \ 3
\]

The compiler can determine that the application `g 1 2 3` is type-correct, but it cannot determine the kind of function passed for `g`. It could be any one of `g1`, `g2`, `g3`, or `g4` from Section 3.2. So it must determine the kind of function being applied, repeatedly. Given a set of arguments and an initial closure, the core of compoundCallRTS is a loop that checks the arity of a closure, calls it with an equal number of arguments from the arguments at the call site, and repeats the process with the resulting closure. The routine terminates with a value or with a new closure when there are no more arguments to consume.

For example, if `g1` is passed, the code generated for the application expressions will determine that this instance of the application is a full application. But if `g2`, `g3`, or `g4` are passed, then the code will determine that it is a compound application. At this point, compoundCallRTS steps in. For `g2`, compoundCallRTS will call `g2` with `1` and `2`, and then will call its result function with `3`. The result of the second call is the result of the entire application expression.

For `g3`, compoundCallRTS will call `g3` with `1`, will call its result with `2` and `3`, and finally, will return the result of the second application as the value of the expression. For `g4`, compoundCallRTS performs three applications: first it calls `g4` with `1`, then its result
with 2, and finally, the result of the second application with 3. The result of the third application is the result of the original application expression.

**Summary**

Full applications of top-level functions are easy to compile. The only subtlety is that references instead of values are passed as arguments. But this is only one of the six cases of application expressions. When all the features of $\lambda^*$ are considered, function applications become surprisingly complex, and in the most general cases, require a great deal of code which we have segregated into the run-time system. In the code sequences above, the cost of higher-order functions is particularly evident: dynamic mechanisms must be invoked because the properties of callees are not known at compile-time. As we mentioned earlier, it is often possible to get rid of these inefficiencies via aggressive inlining at the cost of larger programs.

### 5.5 Compiling Letrec Expressions

A letrec consists of statements and an in expression. The statements are used to bind identifiers to values or to perform side-effects and the in expression is used to give a value to the entire block. The language semantics require that the letrec be strict only in the value of the in expression. The key feature of letrecs is that they are the source of parallelism in $\lambda^*$ programs, via parallel execution of statements.

#### 5.5.1 Parallel Statements

In this section, we consider the compilation of parallel and sequential blocks separately. Though the $\lambda^*$ syntax allows parallel and sequential statements to be intermingled, the front-end of the compiler applies a normalization pass that groups statements into nested parallel and sequential groups. At the outermost level, the statements of letrec blocks execute in parallel. Thus, all letrec blocks follow the pattern

\[
\{ \begin{array}{l}
S_1 \\
S_2 \\
\vdots \\
S_n \\
in \ x
\end{array} \}
\]
where the statements $S_1$ through $S_n$ always execute in parallel. Each statement individually may be a simple statement, like a binding, or it can be a group of sequential statements. For example, $S_2$ might expand to $(S_{10} - S_{11} - S_{12})$. The statements $S_{10}$ through $S_{12}$ are separated by barriers (—). In turn, each statement can be a simple statement or a group of parallel statements.

The following rule is used to compile a letrec whose outermost statements are to execute in parallel. It relies on $\text{TS}$, the rule to compile statements, explained later. First, the statements and the in expression are compiled into separate code sequences. Then, these are linked together using “glue” code to spawn each sequence as a separate task. For instance, before the code for $S_1$, we insert instructions to spawn $S_2$. At the end of the code for $S_1$, we insert a few instructions to terminate $S_1$ and to request a new thread from the RTS. The code for the in expression is the exception since it is the last task in the letrec. It does not need to spawn any other tasks nor does it request work from the RTS. Rather, control flows out of it to the rest of the program.

$$
\text{TE}[S_1 ; S_2 ; \ldots ; S_{n-1} ; S_n \text{ in } x] = \\
\{ \text{label}_1, \ldots, \text{label}_n, \text{label}_\text{in} = \text{genLabel}, \ldots, \text{genLabel}, \text{genLabel} \\
S_1 \text{Code} = \text{TS}[S_1] \\
S_2 \text{Code} = \text{TS}[S_2] \\
\ldots \\
S_{n-1} \text{Code} = \text{TS}[S_{n-1}] \\
S_n \text{Code} = \text{TS}[S_n] \\
in \text{Code} = \text{TE}[x] \\
\text{finishTaskCode} = [ \text{ decbar}(rB), \text{ schedule } ] \\
\text{spawn} S_2 = [ \text{ label}_1 : , \text{ load}(rV, a_{\text{task}_2}), \text{ store}(rV[I], \text{ label}_2), \text{ incbar}(rB), \text{ spawn}(rV) ] \\
\text{spawn} S_3 = [ \text{ label}_2 : , \text{ load}(rV, a_{\text{task}_3}), \text{ store}(rV[I], \text{ label}_3), \text{ incbar}(rB), \text{ spawn}(rV) ] \\
\ldots \\
\text{spawn} S_n = [ \text{ label}_{n-1} : , \text{ load}(rV, a_{\text{task}_n}), \text{ store}(rV[I], \text{ label}_n), \text{ incbar}(rB), \text{ spawn}(rV) ] \\
\text{spawnIn} = [ \text{ label}_n : , \text{ load}(rV, a_{\text{task}_{\text{in}}}), \text{ store}(rV[I], \text{ label}_{\text{in}}), \text{ incbar}(rB), \text{ spawn}(rV) ] \\
in \\
\text{spawn} S_2 \text{ ++ } S_1 \text{Code} \text{ ++ } \text{finishTaskCode} \text{ ++ } \\
\text{spawn} S_3 \text{ ++ } S_2 \text{Code} \text{ ++ } \text{finishTaskCode} \text{ ++ } \\
\ldots \\
\text{spawn} S_n \text{ ++ } S_{n-1} \text{Code} \text{ ++ } \text{finishTaskCode} \text{ ++ } \\
\text{spawnIn} \text{ ++ } S_n \text{Code} \text{ ++ } \text{finishTaskCode} \text{ ++ } \\
[ \text{ label}_{\text{in}} ] \text{ ++ } in \text{Code} \text{ ++ } [ \text{ decbar}(rB) ] \}
$$
5.5.2 Simple Statements: Bindings and Side-Effects

There are different versions of TS, one for each kind of statement in the syntax of \( \lambda^* \). Here we present the rules to compile the simplest statements, beginning with the rule to compile bindings:

\[
TS[x = E] = TE[E] \oplus [ \text{istore}(\@x, rV) ]
\]

The rule emits the code for the right-hand side of the binding, to compute its value, and appends an istore instruction to write the resulting value in the cell allocated for the variable, the left-hand side of the binding. We use istore rather than store to record the value of the variable because other computations may be waiting for it already. Nothing in our compilation scheme guarantees that the value of a variable will be produced before it is first requested, so we must use the synchronizing version of the store instruction. Notice that the istore instruction depends on the convention followed by the TE rule that expression values are computed in register \( rV \).

The other kind of simple statements are side-effects: iStore and mStore. In these, the first argument is the address of a memory cell and the second is a value to store in the cell. The code generated for iStore and mStore is identical since the real difference in the protocols is embodied in the iFetch and mFetch primitives, which are in turn based on the touch and take SMT instructions. In the rule below, we show iStore as an example. Notice that the code makes certain that both values (the cell pointer and the value to store) have been computed by issuing two touch instructions:

\[
TS[iStore(x,y)] = \\
[ \text{touch}(\@x) \,* , \text{touch}(\@y) \,* , \text{load}(r1, \@x), \text{load}(r2, \@y), \text{istore}(r1, r2) ]
\]

In the following example, we use the letrec compilation rule and the two simple-statement rules to generate code for a letrec expression consisting of three statements: two bindings and an iStore. According to the semantics of the language, all the statements and the in expression might run in parallel limited only by data dependences.
\{ x = \text{allocate}(1) ; \\
y = 151 ; \\
i\text{Store}(x, y) \\
in y \} \\

\text{l.x:} \\
\quad \text{load}(rV, rF[\text{task}_y]) \\
\quad \text{store}(rV[1], \_y) \\
\quad \text{incbar}(rB) \\
\quad \text{spawn}(rV) \\
\quad \text{allocate}(rV, 1) \\
\quad \text{istore}(@rF[x], rV) \\
\quad \text{decbar}(rB) \\
\quad \text{schedule} \\

\text{l.y:} \\
\quad \text{load}(rV, rF[\text{task}_\text{iStore}]) \\
\quad \text{store}(rV[1], \_\text{iStore}) \\
\quad \text{incbar}(rB) \\
\quad \text{spawn}(rV) \\
\quad \text{move}(rV, 151) \\
\quad \text{istore}(@rF[y], rV) \\
\quad \text{decbar}(rB) \\
\quad \text{schedule} \\

\text{l.iStore:} \\
\quad \text{load}(rV, rF[\text{task}_\text{in}]) \\
\quad \text{store}(rV[1], \_\text{in}) \\
\quad \text{incbar}(rB) \\
\quad \text{spawn}(rV) \\
\quad \text{touch}(@rF[x]) \ast \\
\quad \text{touch}(@rF[y]) \ast \\
\quad \text{load}(r1, @rF[x]) \\
\quad \text{load}(r2, @rF[y]) \\
\quad \text{istore}(r1, r2) \\
\quad \text{decbar}(rB) \\
\quad \text{schedule} \\

\text{l.in:} \\
\quad \text{touch}(@rF[y]) \ast \\
\quad \text{load}(rV, @rF[y]) \\
\quad \text{decbar}(rB)
The label 1.x marks the code for the first statement in the letrec expression, the binding of variable x. The first thing done by the code is to initialize the task structure for the task that computes the next binding, the binding for y. The initialization consists of setting the pc field of the task structure, at offset 1, to the address of the entry point of the task at label 1.y. Task structures are mapped statically at compile-time to each independent computation in the program. The compiler determines exactly the number of live variables per task and generates the appropriate allocation code at the beginning of the procedure containing the task as we saw in Section 5.4. Different tasks can share the same task structure if it can be guaranteed that they do not execute concurrently.

The next instruction in the code for statement x is an incbar. This instruction is used to increment the current barrier counter before spawning the task structure for y onto the global work queue. The increment is used to keep track of the new thread created by the spawn instruction.

After the spawn, the code for computing the value of x begins. The allocate(1) expression is compiled directly to an SMT allocate instruction using TE, an istore instruction stores the pointer to the allocated space in the cell for x. Finally, the decbar instruction decrements the current barrier counter, and the schedule terminates the current thread and causes a new thread to be executed. The purpose of the decbar instruction is to assert that one less thread exists in the current barrier region. It complements an incbar that was executed outside the code for the letrec when the current thread was spawned.

The code for the second binding at label 1.y follows a structure similar to the first. The core of the code computes the value of the right-hand side of the binding, the value 151, using a single move instruction according to TE in Section 5.2.1. This is followed by an istore instruction to record the value in the memory cell for y. The four instructions preceding the move are used to spawn the task to perform the last statement in the letrec. The last two instructions after the istore consist of a decbar and schedule pair. The decbar marks the current task as terminated in the current barrier region and the schedule grabs a new task to execute from the global work queue. The code for the last statement spawns the task to compute the in expression of the letrec, performs the iStore side-effecting statement, and terminates with the usual decbar and schedule instructions.

The code for the in expression is slightly simpler than that for the bindings. First of all, since the in expression computes the final value of the letrec, it does not spawn any
other tasks. It anchors the chain of spawns in the code for the letrec statements. The
in code consists of a touch and load for the expression \( y \) followed by a decbar. As in the
case of the statements, the decbar is used to indicate that the task for the in expression,
spawned by last statement, has terminated. Control flows directly from the decbar to the
next instruction of the program, outside of the letrec expression, as the letrec is only a piece
of a larger expression. Notice, however, that when control does flow out of the code, the
value of the letrec will be in register \( rV \) as required by our compiling conventions.

The scheme for compiling letrecs exposes parallelism incrementally, not all at once as in
our previous work [6]. Statement \( x \) spawns \( y \), \( y \) spawns the iStore, and the iStore spawns
the in expression. If enough processors request work between successive spawns, this scheme
allows all statements and the in expression to execute in parallel, with data dependences
as the only restriction. In other words, the scheme does not artificially restrict parallelism.
This property is not important just for performance reasons but also for correctness. On
the other hand, the scheme also allows the in expression to execute even if all statements
suspend on missing data. The letrec block is strict only in the value of the in expression,
so this behavior is again important to implement the correct semantics of the language.

It may seem odd to include barrier instructions like incbar and decbar in the code for
parallel statements. After all, barriers are supposed to be used to sequentialize statements.
The scope of barriers, however, is dynamic, so even though the letrec block in isolation
contains no barriers, it may still execute within the context of a barrier. In our particular
implementation, a global barrier, initialized by the run-time system, encloses the execution
of the entire program. As result, all statements run in the context of some barrier. The
global barrier is used by the run-time system to detect termination of the entire program
since in a non-strict program, simply returning a result does not imply that computation
has ceased. Remember that a \( pH \) program may perfectly well return an “answer” while it
is still performing side-effecting operations like output to the screen.

5.5.3 Sequential Statements

The front-end of the compiler arranges letrec expressions so that parallel and sequential
statement groups alternate, with a parallel group at the outermost level of every letrec.
The previous rule showed how to compile a parallel group of statements which in general,
may be followed by an in expression. In the following rule, we show how to compile a group
of sequential statements. A sequential group is never followed by an in expression since it is always enclosed within a parallel group.

The statements in a sequential group are always separated by barriers (—). In the rule below, each barrier has been given a unique local name \((b_1, b_2, \ldots)\) by the compiler since each is implemented with a separate barrier structure. The local name represents the slot in the activation frame that holds the pointer to the barrier structure. These are the same barrier structures initialized at the beginning of the procedure after the activation frame is allocated (Section 5.4).

Like the rule for compiling parallel statement groups, the rule for sequential groups generates the code for each statement separately and then links the code sequences together with "glue" instructions for synchronization.

\[
\begin{align*}
\text{TS} &\left[ S_1 \rightarrow b_1 S_2 \rightarrow b_2 \cdots \rightarrow b_{n-2} S_{n-1} \rightarrow b_{n-1} S_n \right] = \\
&\begin{cases}
S_1 \text{Code} = \text{TS}[S_1] \\
S_2 \text{Code} = \text{TS}[S_2] \\
\vdots \\
S_{n-1} \text{Code} = \text{TS}[S_{n-1}] \\
S_n \text{Code} = \text{TS}[S_n] \\
\text{finishStatement} = \left[ \text{decbar}(rB), \text{touchbar}(rB) \star \right] \\
in \\
\left[ \text{initbar}(@b_1, 1) \right] ++ S_1 \text{Code} ++ \text{finishStatement} ++ \\
\left[ \text{initbar}(@b_2, 1) \right] ++ S_2 \text{Code} ++ \text{finishStatement} ++ \\
\vdots \\
\left[ \text{initbar}(@b_{n-1}, 1) \right] ++ S_{n-1} \text{Code} ++ \text{finishStatement} ++ \\
S_n \text{Code}
\end{cases}
\end{align*}
\]

The synchronization code for each statement is divided into two parts. The first part is a single initbar instruction at the beginning of each statement. This instruction initializes a new barrier object on entry to the statement. The new barrier becomes the current barrier object in register \(rB\) and is used to keep track of all activities created by the statement code.

The second part of the synchronization code consists of a pair of instructions after the code for the individual statements (except the last). These two instructions are a decbar and a touchbar of the current barrier. The decbar is used to decrement the initialization count of the barrier. If no other tasks are outstanding when the decbar executes, then the subsequent touchbar will also execute without suspending. However, if additional tasks
were created by the code for the statement and some of these have not completed, then the decbar will not decrement the barrier counter to zero and the touchbar will suspend. When the last decbar executes in one of the pending tasks, the code following the touchbar will resume executing. This code corresponds to the next statement in the sequence. Thus, the barrier prevents the code for the next statement from executing before all tasks created in the previous statement have terminated. The last statement in a sequential group does not create its own barrier. Rather, it forms the post-region of the last barrier. Control flows out of this last statement to the rest of the letrec expression surrounding the sequential statement group.

In the following example, we show the code generated for a letrec with both parallel and sequential statement groups.

```plaintext
{ z = 1;
    ( x = allocate(z) —
        y = 151 —
        iStore(x,y) )
in y }
```

```
1_z:
    load(rV,rF[task_seq])
    store(rV[1],1_seq)
    incbar(rB)
    spawn(rV)
    move(rV,1)
    istore(@rF[z], rV)
    decbar(rB)
    schedule

1_seq:
    load(rV,rF[task_in])
    store(rV[1],1_in)
    incbar(rB)
    spawn(rV)
    initbar(@rF[barrier1], 1)
    touch(@rF[z]) *
    load(r1, @rF[z])
    allocate(rV,r1)
    istore(@rF[x], rV)
    decbar(rB)
```
According to the semantics of the language, the statement \( z \), the entire sequential statement group, and the \texttt{in} expression can run in parallel. The code for \( z \) spawns the sequential statements as a parallel task. In turn, the first thing done in the sequential statements is to spawn the \texttt{in} expression. Within the sequential statement group, execution flows serially except at touchbar instructions. These instructions prevent execution from proceeding until all activities in the pre-region (the code before the touchbar) have terminated. At the end of the sequential group, we find the usual \texttt{decbar} and \texttt{schedule} pair. These mark the end of the task comprised by the entire sequential statement group.

The barriers only guarantee that \( x \), \( y \), and the \texttt{iStore} will execute in order. Outside of the sequential statement group the only restrictions on execution order arise from data dependences. Since the \texttt{in} expression depends on \( y \), we can be certain the values of \( z \), \( x \) and \( y \) will be computed before the value of the \texttt{in} expression is computed. The execution of the \texttt{iStore} is less certain. It may execute before or after the value of the \texttt{in} expression is computed.

5.5.4 Summary

A \texttt{letrec} expression consists of a parallel statement group and an \texttt{in} expression. Code for each statement is generated separately, and statement code sequences are linked together.
with instructions to synchronize their execution. In a parallel statement group, the synchronization code performs two functions. It spawns tasks onto the global work queue, and it keeps track of the termination of tasks. To avoid creating excessive work, parallel work is made available incrementally. The letrec code is structured so that execution will always reach the in expression even if all statements suspend. Letrecs are strict only in the value of the in expression, and the SMT code must preserve these semantics.

A sequential statement group may be nested within a parallel statement group in a letrec. Sequential statements are also compiled individually and stitched together with synchronization instructions. In this case, the synchronization code makes use of barrier structures to keep track of termination at the statement granularity. All the tasks created by the code for one statement must terminate before the next statement begins to execute.

5.6 Compiling Programs

In this section we present the rule TP for compiling whole programs. This rule drives the SMT code generation process and calls TE as a subroutine. First, it is important to discuss the form of a pH program at the time it reaches the compiler back-end. A complete pH program consists of a number of independently-compiled modules one of which must be named Main and must contain a function main. The function main is the entry-point into the program. Each module is specified in a single file and consists of a set of top-level definitions. These definitions are similar to the binding statements in a letrec expression, except that they occur outside the scope of any expression. The left-hand side of a definition is a top-level identifier, and the right-hand side is always a lambda expression.

The TP rule produces a single long sequence of instructions corresponding to the code for all top-level definitions. This single sequence is constructed by concatenating the subsequences of code for each definition. Before the code for each individual definition, the TP rule adds the entry-point label for the function. The name of this label is derived from the name of the function's top-level identifier, and the label is used to generate the code for application expressions in Section 5.4.2.

In addition, the TP rule generates a static closure data structure for each top-level function using the genClosure routine. The closure is labeled with the top-level identifier bound to the λ-expression used to generate the closure. Remember from Section 5.1.4 that
a closure consists of at least three slots: the arity of the closure, the entry point to the code, and the number of arguments curried in the closure (zero for top-level functions). The \texttt{genClosure} routine takes two arguments to generate the data structure: the entry point to the code and the lambda expression represented by the closure. From the lambda expression, \texttt{genClosure} can determine the arity value to place in the first slot of the closure. The \texttt{genClosure} routine returns a sequence of SMT pseudo-instructions to describe the structure of the closure. Pseudo-instructions are common in most low-level machine languages and are used to specify compilation directives rather than code. The SMT pseudo-instructions are described informally below.

\begin{align*}
\text{TP}\left[x_1 = \lambda_1 \ldots E_1 \right. \\
 & x_2 = \lambda_2 \ldots E_2 \\
 & \ldots \\
 & x_n = \lambda_n \ldots E_n \right] = \\
\left\{ \begin{array}{l}
\lambda_1 \text{Code} = [\ x_1 \text{entry\_point}: \ ] + \text{TE}[\lambda_1 \ldots] \\
\lambda_1 \text{Closure} = [\ x_1 : \ ] + \text{genClosure}(\lambda_1 \ldots) \\
\lambda_2 \text{Code} = [\ x_2 \text{entry\_point}: \ ] + \text{TE}[\lambda_2 \ldots] \\
\lambda_2 \text{Closure} = [\ x_2 : \ ] + \text{genClosure}(\lambda_2 \ldots) \\
\ldots \\
\lambda_n \text{Code} = [\ x_n \text{entry\_point}: \ ] + \text{TE}[\lambda_n \ldots] \\
\lambda_n \text{Closure} = [\ x_n : \ ] + \text{genClosure}(\lambda_n \ldots) \\
\end{array} \right. \\
\text{in} \\
\lambda_1 \text{Code} + \lambda_2 \text{Code} + \ldots + \lambda_n \text{Code} + \\
\lambda_1 \text{Closure} + \lambda_2 \text{Closure} + \ldots + \lambda_n \text{Closure}
\end{align*}

In the following example, the \textbf{TP} rule is used to compile a \textit{pH} module consisting of the single function \textit{identity}.
\[ \text{identity} = \lambda \ a \ a \]

**identity.entry.point:**
- allocate(r9, 5)
- store(r9[0], rF)
- move(rF, r9)
- store(rF[I], rL)
- store(rF[2], rT)
- store(rF[a], r1)
- allocate(rT, 3)
- store(rF[task1], rT)
- touch(@rF[a]) *
- load(rV, @rF[a])
- load(rT, rF[2])
- load(rL, rF[I])
- load(rF, rF[0])
- jump(Always, rL)

**.datablock 4**

**identity:**
- .datapointer identity_closure

**identity.closure:**
- .integer 1
- .codepointer identity.entry.point
- .integer 0

The code following the `identity.entry.point` is generated by the `TE` function applied to a single-argument \( \lambda \)-abstraction. It contains the instructions to allocate and initialize a frame, compute the value of \( x \), and return from the function. The frame for `identity` contains five things: the caller’s frame address, the return address, the caller’s task structure, a reference to the argument \( x \), and a task structure for the body of the lambda expression.

The data block following it is generated by the `genClosure` compiler routine. The `.datablock` pseudo-instruction defines the beginning a four-cell static data structure. The first cell in the structure is a pointer to the closure for the `identity` function and is used in applications of `identity`. The next cell is the first cell of the closure. It contains the integer 1, representing the arity of the closure since `identity` has one argument. The
second cell of the closure contains the entry point label of *identity*, by convention the label *identity_entry_point*. This label is resolved into an address by the linker when the SMT code is transformed into an executable program. The last cell contains zero since the top-level closure for *identity* contains no curried arguments.

### 5.7 Summary

In this chapter, we have presented the translation of *λ* programs to SMT code. We began by specifying the execution environment for the SMT code, including the structure of procedure activation frames, closures, user data structures, and the use of references to implement non-strictness. Then, we described the translation of each expression and statement in *λ* using the functions **TE** and **TS**. The whole syntax-directed translation process is driven by the rule to compile programs **TP**.

The translation process involves a major change in the representation of programs, from a high-level, implicitly parallel language to a low-level, explicitly-parallel machine instruction set. The most involved expressions to translate are *λ*-abstractions, function applications, and letrec blocks. Both *λ*-abstractions and function calls are complex for two reasons: the parameter passing convention must accommodate higher order functions and currying; and the activation frames must be initialized properly to support synchronization of parallel activities. In turn, letrec blocks are the source of parallelism in *λ* programs. The compiler must generate instructions to manage parallelism and to track termination of concurrent tasks, all while maintaining the correct semantics of the source language.
Chapter 6

Better Threaded Code

An SMT thread is a sequence of instructions that is enabled by a spawn instruction and that terminates when it encounters a schedule instruction. It is the basic schedulable entity in the SMT programming model. In the previous chapter, the compilation of letrec blocks embodied a trivial algorithm for dividing a $\lambda^*$ program into threads: every statement in a parallel block was spawned separately. Though always correct, this strategy produces too many threads and incurs too much scheduling overhead. Often, several statements could be computed together, in the same thread, without affecting the non-strictness or termination properties of the program.

In this chapter we examine how the compiler creates larger threads for more efficient programs. The key to larger threads is partitioning analysis, whose history we discussed in Section 3.3. Using the results of partitioning analysis, the compiler performs a threading transformation. Threading is a source-to-source transformation in the $\lambda^*$ representation of a program. Essentially it involves replacing parallel statement conjunctions ($;$) with threaded statement conjunctions ($\sim$) in letrecs. The resulting threaded program is then compiled to SMT code that is significantly faster than that resulting from the non-threaded version of the program.

In this chapter, we begin by comparing suspensive and non-suspensive threads. Then, we discuss the last stage of the threaded compilation process, using fib as a running example. We assume that a threaded $\lambda^*$ program has been given to the compiler back-end, and we discuss the changes that have to be made to generate better SMT code. We emphasize code generation because the back-end is essentially independent of the partitioning analysis.
and threading transformation. Better partitioners could be implemented with little or no change to a back-end that knows how to generate SMT code for threaded letrecs.

After discussing the code generator, we take up the subject of partitioning. We describe the information produced by partitioning analysis, and we show how the threading transformation uses it, along with additional heuristics, to produce a threaded $\lambda^*$ program. Interestingly, the threading transformation can be too successful and can remove all parallelism from a program. We show what can be done in this situation. We conclude with a discussion of how scheduling has affected the strategy we use to spawn parallel threads.

6.1 Suspensive versus Non-Suspensive Threads

In existing compilers for Id and pH, like the Berkeley Id compiler [16, 20, 55] and pHluid [18], a thread is a non-suspensive sequence of instructions with strictly linear control-flow, i.e., no jumps or loops. The thread is non-suspensive because all the local (frame-based) values that it requires are guaranteed to be available before the thread begins to execute. Non-suspensive threads are organized so that they synchronize “up-front” and will not have to suspend mid-stream. To access I-Structures and M-Structures, they use split-phase memory transactions. The first part of a split-phase transaction is a non-blocking request for data from memory. The second part of the transaction is a reply from memory with data that activates the execution of a new thread. Since the request and reply in a split-phase memory transaction can be separated by an arbitrary amount of time (e.g., when an I-Structure is empty), a thread must terminate, and yield control to the scheduler, after issuing the first part of the split-phase transaction. Otherwise, the thread would violate the non-suspensive assumption. Even if the value happened to be available at run-time, the thread must still terminate because the synchronization status of the memory location cannot be known at compile time. In other words, non-suspensive threads imply a pessimistic synchronization strategy.

Threads in the pHc compiler are more substantial sequences of code than in Berkeley Id or pHluid. According to the definition, an SMT thread can encompass rich control-flow, including loops and conditionals. In addition, an SMT thread is suspensive: synchronization is done “just-in-time”. If a local variable or the value of an I/M-Structure cell is not available, the thread suspends until the value is computed. But if a value is available, the
thread continues on. Thus, I-Structure and M-Structure operations do not force a thread to terminate. In this way, a suspensive thread is optimistic about synchronization, and the code can be structured so that the common case, where a value is available before it is used, can run quickly.

The code generated by the Berkeley Id and pHluid compilers is not always pessimistic. In certain restricted situations, it can also apply the optimistic assumptions that are the default in pHc. In Berkeley Id, the compiler often inlines the inlets that receive the split-phase response from memory with the code that issues the split-phase request when the value is known to be available. However, this optimization is applied only when the compiler is emitting single-processor code. In the case of pHluid, which is targeted at distributed memory machines, the response handler for the split-phase transaction is executed immediately after the request when the following conditions are true for a synchronizing load instruction: the instruction following the load is a halt (the last instruction in a pHluid thread), the source address is in memory local to the current processor, and the value is available.

The results of Schauser and Goldstein [57] seem to indicate that the optimistic synchronization policy is good for many Id (pH) programs. They studied non-strictness in Id programs, and found that of 45 programs, only eleven required non-strictness, and of those, five were designed specifically to be non-strict. If a program does not require non-strictness to execute properly, then in practice, most values will be computed before they are used. Our results in Section 8.4 bear this out.

Compilers like Berkeley Id and pHluid use non-suspensive threads because their internal program representations are variations on the dataflow program graph (DFPG). In a DFPG, a graph node (operator) executes atomically after all its inputs are available. It is natural to implement such an operator as a sequence of instructions that synchronizes up-front and then runs to completion in a bounded amount of time.

Non-suspensive threads and split-phase memory operations are also a natural fit for these compilers since they originally targeted distributed memory machines. In these machines, memory latencies can be enormous, so it is not feasible to check the presence field of a memory cell before every access, as we do in the SMT touch instruction. Split-phase transactions allow the overlap of communication and computations, so a processor can do useful work while fetching data from remote memory. In an SMP, however, the situation is different. Overlapping computation and communication is not as useful since the time to
access a single-cell of memory is roughly comparable to the time needed to switch to a new thread.

We chose to use suspensive threads in pHc for two reasons. First, it seems to be easier to generate long suspensive threads than non-suspensive threads since suspensive threads do not have to terminate on access to an I-Structure / M-Structure. Long threads run more efficiently than short threads, as long as suspensions are rare, because they can make more effective use of processor state. Long threads can accumulate more state in registers and can amortize scheduling overhead over more instructions. The second reason for using suspensive threads is that they can yield better code when programs run serially in the common case. We wanted to be able to put structures like loops and procedure calls into a single thread whenever possible.

Suspensive threads, however, don’t come for free. Statically, a suspensive thread must include conditional tests to check the status of presence fields and code to save state should a suspension occur. These operations are embodied in suspensive SMT instructions like touch. Dynamically, the most significant cost of suspensive threads is the execution of these conditional tests, not the code to save state. The common case is that threads do not suspend—since programs are rarely non-strict—so branch prediction hardware on modern processor should make short work of the extra conditionals.

6.2 SMT Code from Threaded Programs

In this section we describe how to generate SMT code from programs that have been threaded already. We begin by introducing non-threaded and threaded versions of fib, a program for computing Fibonacci numbers. Then we see how to modify the current compilation rule for letrecs to generate code that is more efficient because it creates fewer threads.

6.2.1 A Non-Threaded fib

A non-threaded version of fib looks like this in \( \lambda^* \):
fib = λ x .
   { p = x >= 2 ;
     t = Prj⁰(p);
     r = Case(t, x, { x₁ = x - 1 ;
                     f₁ = fib x₁ ;
                     x₂ = x - 2 ;
                     f₂ = fib x₂ ;
                     r' = f₁ + f₂
                     in r' } )
   in r }

Though the algorithm is well known, it is worth explaining the λ* code. First, the code tests if the argument is greater than or equal to 2. The result is an object of type Bool, a type consisting of the two disjuncts False and True. Next, the primitive function Prj⁰ is used to extract the tag of the object. As discussed in Section 5.3, tags are an implementation technique that uses integers to distinguish algebraic disjuncts. The tag for the boolean False is 0 and for True is 1.

The tag is used in the Case expression as an index into the alternatives. If x is less than two, in which case the tag will be 0, the first alternative will be executed. This expression is simply the base case for the standard recursive Fibonacci algorithm. If x is greater than or equal to 2, in which case the tag is 1, then the code enters the recursive case of the Fibonacci algorithm. The code makes two recursive calls to fib, with parameters x-1 and x-2, sums the results, and returns that as the result of the function call.

How many threads are created by the current letrec compilation strategy for this version of fib? In the outer letrec block, the compiler will spawn the bindings for t, r, and in r. In the inner letrec block inside the True side of the Case expression, the compiler will spawn the threads for f₁, x₂, f₂, r', and in r'. So in total, eight threads are created for every invocation of fib beyond the base case. Most of these threads, except for the recursive calls to fib, contain only a tiny amount of work.

6.2.2 A Threaded fib

Now suppose the back-end is given the following threaded version of fib:
\( \text{fib} = \lambda x . \{
\quad p = x \geq 2 \\
\quad t = \text{Prj}^0(p) \\
\quad r = \text{Case}(t, x, \{ x1 = x - 1 \\
\quad x2 = x - 2 \\
\quad f1 = \text{fib} x1 \\
\quad f2 = \text{fib} x2 \\
\quad r' = f1 + f2 \\
\quad \text{in } r' \}
\} \)

In this threaded version of \texttt{fib}, the compiler, after careful analysis, has seen fit to replace every parallel statement conjunction (;) with a threaded statement conjunction (\(~\)). In the process, it has also reorganized the order of some statements, but this is not particularly important.

As we mentioned in Section 2.8, the ‘\(~\)’ notation is used to assert that two statements do not need to execute in parallel and that the code for the second statement can always follow the code for the first, regardless of context. In other words, the two statements can execute as a single thread. This threaded version of \texttt{fib} agrees with our intuition that the whole computation should be able to fit into a single thread while maintaining the same termination behavior as the original non-threaded version.

For convenience, the compiler has also inserted ‘\(~\)’ between the last statement in a letrec and the \texttt{in} expression. This is an extension of the syntax of \texttt{\(\lambda^*\)}, and it asserts the threading relationship between these two parts of the expression. It is important to note that any ‘\(~\)’ can be ignored and treated as a ‘;’ without affecting the semantics of the program, but the opposite is not true. A ‘;’ cannot be replaced by ‘\(~\)’ arbitrarily.

### 6.2.3 Compiling Threaded Letrec Expressions

The ‘\(~\)’ notation allows the compiler to make strong assertions about control-flow and synchronization which in turn lead to much better code generation. The compiler knows that statements in the same thread do not have to execute independently. The following simple compilation rule handles groups of threaded statements. The rule compiles the code for each statement in the thread separately, like the other thread compilation rules. But now, it does not need to add any “glue” code to link the sequences together! The ‘\(~\)’ asserts that control-flow can pass directly from one sequence to the next, so they are simply
concatenated into one.

\[
\text{TS}[S_1 \sim S_2 \sim \ldots \sim S_n] = \\
\{ S_1 \text{Code} = \text{TS}[S_1] \\
S_2 \text{Code} = \text{TS}[S_2] \\
\ldots \\
S_n \text{Code} = \text{TS}[S_n] \\
\text{in} \\
S_1 \text{Code} ++ S_2 \text{Code} ++ \ldots ++ S_n \text{Code} \\
\}
\]

The nesting of parallel and sequential regions remains the same as before. Letrec blocks are assumed to be parallel at the outermost level, but now, their component statements can be sequences of statements connected by threading conjunctions or by barriers.

Contrasting the rule for compiling threaded statements with the rule for compiling sequential statements (Section 5.5.3) makes clear the distinction between the ‘\(S_1 \sim S_2\)’ and ‘\(S_1 \sim S_2\)’. The synchronization guarantee offered by a barrier is absolute: even if control-flow reaches \(S_2\), execution cannot continue until all previous activities created by \(S_1\) have completed. It is the responsibility of the programmer who has inserted the barrier to ensure that the program will not deadlock given this strong synchronization. The ‘\(\sim\)’ statement conjunction, which is inserted by the compiler not the programmer, provides a different guarantee. It assures that no value consumed by any task in \(S_1\) is produced by any task in \(S_2\). In other words, there is no chain of dependences from computations in \(S_2\) to computations in \(S_1\). Thus \(S_2\) can follow \(S_1\) without the possibility of deadlock, in all execution contexts. By itself, the ‘\(\sim\)’ guarantees nothing about the termination of tasks created by \(S_1\) when control reaches \(S_2\).

There is one minor modification that needs to be made to the compilation scheme for letrec expressions. When a threaded program indicates that the \texttt{in} expression of a letrec can come in the same thread as the last statement, we omit spawning the \texttt{in} expression and simply concatenate it to the code for the last statement. We show this new rule below, and use it only when there is a ‘\(\sim\)’ between the last statement and the \texttt{in} expression.
\[ \text{TE}[S_1; S_2; \ldots; S_{n-1}; S_n \sim \text{in } x] = \\
\begin{array}{l}
\{ \text{label}_1, \ldots, \text{label}_n = \text{genLabel}, \ldots, \text{genLabel} \\
S_1\text{Code} = \text{TS}[S_1] \\
S_2\text{Code} = \text{TS}[S_2] \\
\ldots \\
S_{n-1}\text{Code} = \text{TS}[S_{n-1}] \\
S_n\text{Code} = \text{TS}[S_n] \\
in\text{Code} = \text{TE}[x] \\
\text{finishTaskCode} = [\text{decbar}(rB), \text{schedule}] \\
\text{spawnS}_2 = [\text{label}_1:, \text{load}(rV, atask_2), \text{store}(rV[1], \text{label}_2), \text{incbar}(rB), \text{spawn}(rV)] \\
\text{spawnS}_3 = [\text{label}_2:, \text{load}(rV, atask_3), \text{store}(rV[1], \text{label}_3), \text{incbar}(rB), \text{spawn}(rV)] \\
\ldots \\
\text{spawnS}_n = [\text{label}_{n-1}:, \text{load}(rV, atask_n), \text{store}(rV[1], \text{label}_n), \text{incbar}(rB), \text{spawn}(rV)] \\
\text{lastStmt} = [\text{label}_n:] \\
in \\
\text{spawnS}_2 \leftrightarrow S_1\text{Code} \leftrightarrow \text{finishTaskCode} \leftrightarrow \\
\text{spawnS}_3 \leftrightarrow S_2\text{Code} \leftrightarrow \text{finishTaskCode} \leftrightarrow \\
\ldots \\
\text{spawnS}_n \leftrightarrow S_{n-1}\text{Code} \leftrightarrow \text{finishTaskCode} \leftrightarrow \\
\text{lastStmt} \leftrightarrow S_n\text{Code} \leftrightarrow \text{in\text{Code}} \leftrightarrow [\text{decbar}(rB)]
\end{array}
\]

6.2.4 Additional Changes for Threaded Compilation

Due to the \(\sim\) notation, it is easy to determine the order in which statements execute at compile-time, and consequently, it is easy to determine what local values have been computed by the time a particular expression executes. The compiler gathers this available identifiers information as annotations in a pass before SMT code generation. The annotation for each expression contains the set of identifiers whose values are known to be available when control-flow reaches the expression. Here is the annotated, threaded version of \texttt{fib}:
fib = \lambda x .
{ p = x >= 2 ~
t = Prj^0(p) ~
r = Case(t,
  x,
  \{ x1 = x - 1 ~
x2 = x - 2 ~
f1 = fib x1 ~
f2 = fib x2 ~
r' = f1 + f2 ~
in r' } ) ~
in r }

Available Identifiers Annotations

{ p = x >= 2 ~
{ p,t,x }
{ p,t,x }
{ p,t,x }
{ p,t,x,x1 }
{ p,t,x,x1,x2 }
{ f1,p,t,x,x1,x2,}
{ f1,f2,p,t,x,x1,x2 }
{ f1,f2,p,r',t,x,x1,x2 }
{ p,r,t,x }

Before generating the code for an expression, the compiler checks the contents of the corresponding annotation and does not emit a touch instruction if the value of the identifier is available. For example, in the expression to compute r', the identifiers f1 and f2 appear in the annotation, so the compiler will generate the following code for the primitive application:

\[
\begin{align*}
\text{f1} + \text{f2} \\
\text{load}(r1, @rF[f1]) \\
\text{load}(r2, @rF[f2]) \\
\text{plus}(rV, r1, r2)
\end{align*}
\]

The structure of the code guarantees that f1 and f2 are always computed before r'. The rules to compile identifiers, primitive functions, allocate, iFetch, mFetch, projections, Case expressions, and applications of higher-order functions change slightly to exploit these annotations.

6.2.5 SMT Code for Threaded fib

Figure 6-1 shows the dramatic improvements in code quality that result when the compiler takes advantage of threading. The left-hand column of the figure shows the code that would be generated for the threaded fib by applying the compilation strategy from Chapter 5 and by treating '~' as ';'. For reference, the code begins with the label fib.entry_point to mark the beginning of the function, as in Section 5.4. To fit on one page, we omit the code for frame allocation/initialization, the definitions for p and t in the outer letrec, and the beginning of the conditional, and we resume with the code for
<table>
<thead>
<tr>
<th>Original Code</th>
<th>Threaded Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>fib.entry point:</strong></td>
<td><strong>fib.entry point:</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>start.x1:</strong></td>
<td><strong>start.x1:</strong></td>
</tr>
<tr>
<td>load(r1,rF[task.x2])</td>
<td>sub(r.x1,r.x,1)</td>
</tr>
<tr>
<td>store(r1[pc],start.x2)</td>
<td>sub(r.x2,r.x,2)</td>
</tr>
<tr>
<td>incbar(rB)</td>
<td>load(rV,fib)</td>
</tr>
<tr>
<td>spawn(r1)</td>
<td>load(rV,rV)</td>
</tr>
<tr>
<td>touch(rF[x]) *</td>
<td>move(r1,r.x)</td>
</tr>
<tr>
<td>sub(rV, r1, 1)</td>
<td>jump(Always,fib.entry.point,rL)</td>
</tr>
<tr>
<td>istore(rF[x1],rV)</td>
<td>move(r.f1,rV)</td>
</tr>
<tr>
<td>decbar(rB)</td>
<td>load(rV,fib)</td>
</tr>
<tr>
<td>schedule</td>
<td>load(rV,rV)</td>
</tr>
<tr>
<td></td>
<td>move(r1,r.x2)</td>
</tr>
<tr>
<td></td>
<td>jump(Always,fib.entry.point,rL)</td>
</tr>
<tr>
<td></td>
<td>move(r.f2,rV)</td>
</tr>
<tr>
<td></td>
<td>plus(rV,f1,r.f2)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>start.f1:</strong></td>
<td><strong>start.f1:</strong></td>
</tr>
<tr>
<td>load(r1,rF[task.f1])</td>
<td></td>
</tr>
<tr>
<td>store(r1[pc],start.f1)</td>
<td></td>
</tr>
<tr>
<td>incbar(rB)</td>
<td></td>
</tr>
<tr>
<td>spawn(r1)</td>
<td></td>
</tr>
<tr>
<td>load(rV,fib)</td>
<td></td>
</tr>
<tr>
<td>load(rV,rV)</td>
<td></td>
</tr>
<tr>
<td>load(r1,rF[x1])</td>
<td></td>
</tr>
<tr>
<td>jump(Always,fib.entry.point,rL)</td>
<td></td>
</tr>
<tr>
<td>istore(rF[f1],rV)</td>
<td></td>
</tr>
<tr>
<td>decbar(rB)</td>
<td></td>
</tr>
<tr>
<td>schedule</td>
<td></td>
</tr>
<tr>
<td><strong>start.f2:</strong></td>
<td><strong>start.f2:</strong></td>
</tr>
<tr>
<td>load(r1,rF[task.r'])</td>
<td></td>
</tr>
<tr>
<td>store(r1[pc],start.r')</td>
<td></td>
</tr>
<tr>
<td>incbar(rB)</td>
<td></td>
</tr>
<tr>
<td>spawn(r1)</td>
<td></td>
</tr>
<tr>
<td>load(rV,fib)</td>
<td></td>
</tr>
<tr>
<td>load(rV,rV)</td>
<td></td>
</tr>
<tr>
<td>load(r1,rF[x2])</td>
<td></td>
</tr>
<tr>
<td>jump(Always,fib.entry.point,rL)</td>
<td></td>
</tr>
<tr>
<td>istore(rF[f2],rV)</td>
<td></td>
</tr>
<tr>
<td>decbar(rB)</td>
<td></td>
</tr>
<tr>
<td>schedule</td>
<td></td>
</tr>
<tr>
<td><strong>start.r':</strong></td>
<td><strong>start.r':</strong></td>
</tr>
<tr>
<td>load(r1,rF[task.in])</td>
<td></td>
</tr>
<tr>
<td>store(r1[pc],start.in)</td>
<td></td>
</tr>
<tr>
<td>incbar(rB)</td>
<td></td>
</tr>
<tr>
<td>spawn(r1)</td>
<td></td>
</tr>
<tr>
<td>touch(rF[f1]) *</td>
<td></td>
</tr>
<tr>
<td>touch(rF[f2]) *</td>
<td></td>
</tr>
<tr>
<td>load(r1,rF[f1])</td>
<td></td>
</tr>
<tr>
<td>load(r2,rF[f2])</td>
<td></td>
</tr>
<tr>
<td>plus(rV,r1,r2)</td>
<td></td>
</tr>
<tr>
<td>istore(rF[r'+1],rV)</td>
<td></td>
</tr>
<tr>
<td>decbar(rB)</td>
<td></td>
</tr>
<tr>
<td>schedule</td>
<td></td>
</tr>
<tr>
<td><strong>start.in:</strong></td>
<td><strong>start.in:</strong></td>
</tr>
<tr>
<td>touch(rF[r']) *</td>
<td></td>
</tr>
<tr>
<td>load(rV, rF[r'])</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>start.x2:</strong></td>
<td><strong>start.x2:</strong></td>
</tr>
<tr>
<td>load(r1,rF[task.x2])</td>
<td></td>
</tr>
<tr>
<td>store(r1[pc],start.x2)</td>
<td></td>
</tr>
<tr>
<td>incbar(rB)</td>
<td></td>
</tr>
<tr>
<td>spawn(r1)</td>
<td></td>
</tr>
<tr>
<td>touch(rF[x]) *</td>
<td></td>
</tr>
<tr>
<td>sub(rV, r1, 1)</td>
<td></td>
</tr>
<tr>
<td>istore(rF[x1],rV)</td>
<td></td>
</tr>
<tr>
<td>decbar(rB)</td>
<td></td>
</tr>
<tr>
<td>schedule</td>
<td></td>
</tr>
<tr>
<td><strong>start.f1:</strong></td>
<td><strong>start.f1:</strong></td>
</tr>
<tr>
<td>load(r1,rF[task.f1])</td>
<td></td>
</tr>
<tr>
<td>store(r1[pc],start.f1)</td>
<td></td>
</tr>
<tr>
<td>incbar(rB)</td>
<td></td>
</tr>
<tr>
<td>spawn(r1)</td>
<td></td>
</tr>
<tr>
<td>touch(rF[x]) *</td>
<td></td>
</tr>
<tr>
<td>load(r1,rF[x])</td>
<td></td>
</tr>
<tr>
<td>sub(rV, r1, 2)</td>
<td></td>
</tr>
<tr>
<td>istore(rF[x2],rV)</td>
<td></td>
</tr>
<tr>
<td>decbar(rB)</td>
<td></td>
</tr>
<tr>
<td>schedule</td>
<td></td>
</tr>
<tr>
<td><strong>start.f2:</strong></td>
<td><strong>start.f2:</strong></td>
</tr>
<tr>
<td>load(r1,rF[task.r'])</td>
<td></td>
</tr>
<tr>
<td>store(r1[pc],start.r')</td>
<td></td>
</tr>
<tr>
<td>incbar(rB)</td>
<td></td>
</tr>
<tr>
<td>spawn(r1)</td>
<td></td>
</tr>
<tr>
<td>load(rV,fib)</td>
<td></td>
</tr>
<tr>
<td>load(rV,rV)</td>
<td></td>
</tr>
<tr>
<td>load(r1,rF[x1])</td>
<td></td>
</tr>
<tr>
<td>jump(Always,fib.entry.point,rL)</td>
<td></td>
</tr>
<tr>
<td>istore(rF[f1],rV)</td>
<td></td>
</tr>
<tr>
<td>decbar(rB)</td>
<td></td>
</tr>
<tr>
<td>schedule</td>
<td></td>
</tr>
<tr>
<td><strong>start.r':</strong></td>
<td><strong>start.r':</strong></td>
</tr>
<tr>
<td>load(r1,rF[task.in])</td>
<td></td>
</tr>
<tr>
<td>store(r1[pc],start.in)</td>
<td></td>
</tr>
<tr>
<td>incbar(rB)</td>
<td></td>
</tr>
<tr>
<td>spawn(r1)</td>
<td></td>
</tr>
<tr>
<td>touch(rF[f1]) *</td>
<td></td>
</tr>
<tr>
<td>touch(rF[f2]) *</td>
<td></td>
</tr>
<tr>
<td>load(r1,rF[f1])</td>
<td></td>
</tr>
<tr>
<td>load(r2,rF[f2])</td>
<td></td>
</tr>
<tr>
<td>plus(rV,r1,r2)</td>
<td></td>
</tr>
<tr>
<td>istore(rF[r'+1],rV)</td>
<td></td>
</tr>
<tr>
<td>decbar(rB)</td>
<td></td>
</tr>
<tr>
<td>schedule</td>
<td></td>
</tr>
<tr>
<td><strong>start.in:</strong></td>
<td><strong>start.in:</strong></td>
</tr>
<tr>
<td>touch(rF[r']) *</td>
<td></td>
</tr>
<tr>
<td>load(rV, rF[r'])</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6-1: Original vs. Threaded Code for the Inner Letrec of fib

122
computing \( x_1 \) at label \texttt{start.x1}. From here on, every label marks the code for a different statement or for the \texttt{in} expression of the inner \texttt{letrec} block.

The right-hand column shows the code generated for the same \texttt{letrec} expression using threading information. The threaded code is much shorter, uses less memory, and executes much faster. The improvements to the code are achieved via the elimination of synchronization "glue" code in \texttt{letrecs}, the elimination of touch instructions by tracking available identifiers, and via additional threading optimizations discussed in the next section. The code is also restructured to use symbolic registers, such as \( r_x1 \) for the variable \( x_1 \).

### 6.3 Optimizations Enabled by Threading

The threading transformation reveals opportunities for optimizations in longer sequences of instructions. In particular, because execution proceeds sequentially within a thread, the compiler can arrange to share the results of computations much more effectively. For instance, values can be computed in registers and reused throughout a thread without needing to be reloaded from the activation frame.

#### 6.3.1 Codeblocks: A Control-Flow Abstraction

The optimizations discussed in this section rely on auxiliary data structures to represent information about the control-flow in a thread. In a traditional compiler for imperative languages, this data structure takes the form of a control-flow graph of basic-blocks on which dataflow analyses are performed [4, Ch. 10]. In the pHc compiler, we follow a similar approach, though our task is simpler for several reasons: a pH variable is defined only once, the language allows only structured control constructs, and loops (at present) are represented as tail-recursive function calls. This last property of the current implementation means that our control-flow graphs are acyclic.

The control-flow graphs in pHc are hierarchical and are based on the concept of a codeblock, a piece of code (possibly consisting of several traditional basic-blocks) with a single entry point and a single exit point. The entry point and the exit point are not necessarily part of a single thread of control since new threads can be spawned within a codeblock. It is guaranteed however, that control will eventually flow to the exit point so that codeblocks can be connected together. There are five kinds of codeblocks, as illustrated

123
Simple: this most basic codeblock consists of a sequence of SMT instructions, including suspensive instructions like touch. Due to the current code generation strategy, all jump instructions are always forward jumps. Simple codeblocks are the only ones that "contain" instructions. As we'll see, the four other kinds of codeblocks contain nested codeblocks.

Conditional: this kind of codeblock is used to abstract the code generated for Case expressions in λ*. It consists of a codeblock for the predicate expression, codeblocks for the alternatives, and an exit codeblock to merge the control-flow from the alternatives. Control flows into the predicate codeblock and out of the exit codeblock, but possibly not as part of a single thread. The predicate and the alternatives, recursively, can be any kind of codeblock, so in general, they can create new threads.

Parallel: the sub-blocks of a parallel codeblock represent computations that can execute in parallel. Control flows into the first sub-block where the next sub-block is spawned. Control enters the spawned codeblocks via the scheduler. Control exits the Parallel
Sequential: the sub-blocks in a sequential codeblock represent computations separated by barriers. Control flows sequentially from one sub-block to the next, though possibly not as part of a single thread. A touchbar is assumed to execute between sub-blocks.

Threaded: a threaded codeblock is similar to a sequential codeblock in that control flows sequentially from one sub-block to the next, but there are no barrier instructions executed between sub-blocks. Threaded blocks result from partitioning.

The codeblock for the threaded version of \texttt{fib}, from Figure 6-1, is shown in Figure 6-3. The outermost codeblock is a Threaded codeblock containing several simple codeblocks and a Conditional codeblock. One of the alternatives of the conditional (the base case) is a Simple codeblock while the other (containing the recursive calls to \texttt{fib}) is another Threaded codeblock.
6.3.2 Defined-Variables Analysis

Using the codeblock structure, it is easy to perform control-flow analysis on SMT code. An important *forward* analysis, one that proceeds in the same direction as control-flow, annotates each instruction with the names of the variables that have been defined in registers in the current thread. With this information, the compiler can remove instructions that load an already-defined variable from memory. We assume the SMT code has been transformed so that each identifier is assigned its own symbolic register. This transformation is common in many intermediate languages, and allows the compiler to use symbolic registers in SMT much in the same way identifiers are used in $\lambda^*$.

The algorithm to compute the defined variables in a codeblock is presented in Figure 6-4 as the procedure *definedVars*. The parameters to the procedure are a codeblock (*cb*) and a
set of defined variables \((d)\). The procedure returns a new set of defined variables, and as a side-effect, annotates all the instructions in the codeblock with the appropriate set of defined variables. The defined variables analysis is performed on each SMT procedure separately, so the initial inputs to \textit{defned Vars} are a codeblock for an entire \(pH\) procedure and the null set \((\emptyset)\) since by convention there are no defined variables on entry to a procedure.

The algorithm performs a case analysis on the kind of codeblock given as input. Though there are five kinds of codeblocks, only four cases are required because Sequential and Threaded codeblocks are considered together. For Simple codeblocks, the only ones that actually contain SMT instructions, the procedure iterates through the list of instructions in order. At each instruction, it does two things. First it sets the \textit{defvars} field of the instruction to the value of the current defined variables set \((d)\). Second, it updates this set with the variables defined by the instruction. These are gotten from the \textit{destinations} field of the instruction which contains the names of all symbolic registers changed by the instruction.

A Conditional codeblock consists of a predicate and at list of codeblocks for the alternatives. The procedure calls itself recursively to compute the defined variables of the predicate, using the original defined variables set \((d)\) as a parameter. The resulting defined variables set \((d')\) is used to process the alternatives. To understand the rest of the Conditional analysis, it is important to keep two facts in mind: first, the defined variables on entry to each alternative are the defined variables on exit from the predicate codeblock; and second, the defined variables on exit from the Conditional are given by the intersection of the defined variables on exit from each alternative. At compile time, it is unknown which alternative will be taken, so the analysis must be conservative in this way. This intersection is computed incrementally in variable \(d'\).

A Parallel codeblock contains a list of sub-blocks. According to our compilation strategy, the defined variables on entry into the first of these sub-blocks are exactly those defined on entry into the enclosing Parallel codeblock. The set of defined variables on entry into the remaining sub-blocks, however, is the empty set. This is due to the fact that the remaining codeblocks might execute in parallel with the first, and defined variables cannot cross thread boundaries at present. The defined variables on exit from the Parallel codeblock are exactly those defined on exit from the last sub-block. This last sub-block generally contains the code for the \textit{in} expression of a letrec.
The Sequential and Threaded codeblocks are handled identically. These codeblocks contain a list of sub-blocks, and control flows from sub-block to sub-block. Therefore, the set of defined variables on entry to a given sub-block is exactly the set of variables defined on exit from the previous sub-block.

The defined-variables analysis assumes that a register defined early in a thread will be available to all instructions that use it within the same thread. After the elimination of redundant load instructions, however, the live-ranges of certain registers might be lengthened considerably, so we must update the live-state annotations of all suspensive instructions. We accomplish this using a standard, backward live-variable analysis. Using the codeblock structure, we compute the set of variables (symbolic registers) live at every instruction and use these to augment the existing live-state annotations of every suspensive instruction.

The defined-variables analysis removes redundant loads from memory, but it can also be used to remove redundant stores. If after the analysis, a particular variable is never loaded from the frame because its definition and all uses occur in the same thread, then the defining store of the variable can also be removed. In turn, the frame slot, reference, and the memory location for the variable also can be eliminated. Unfortunately, there is a slight complication. If we remove references to some values, we must allow values to be passed directly to procedures and to be stored directly in data structures (frames, constructors, closures). The pHc compiler already includes support for this sort of code. It is a simple extension to the existing compilation rules, but it hasn’t been shown here in the interest of clarity. Essentially, the code checks whether or not the contents of a register is a reference or a value using a special IsReference condition in the jump instruction. This sort of test is often needed only at the beginning of a thread. When combined with improved register allocation, it costs little (only one extra instruction per value per thread), but it reduces memory traffic significantly.

6.3.3 Task Structure Sharing Analysis

Each thread in the SMT machine is assigned a task structure. The task structure provides thread-private storage space used to store live-state when a thread must suspend; the task structure is used to build up deferred lists of threads waiting for a value in memory; and the task structure is used to build up the global work queue of threads that are ready to execute. In this section, we describe how task structures are assigned to threads, and we
assignTaskId :: Codeblock → TaskId → TaskId
assignTaskId cb tid =
  case cb of
    Simple instructions →
      let cb.taskId ← tid
     in tid
    Conditional predicate alternatives →
      let tid' ← assignTaskId predicate tid
       result ← tid'
 in for cb' ← alternatives
      tid'' ← assignTaskId cb' tid'
      result ← if (tid' ≠ tid'') then tid''
        else result
    finally result
  Parallel subblocks →
    let (b:bs) ← subblocks
    tid ← assignTaskId b tid
 in for cb' ← bs
      tid' ← assignTaskId cb' (makeNewTid)
    finally tid'
  (Sequential || Threaded) subblocks →
    for cb' ← subblocks
      tid ← assignTaskId cb' tid
    finally tid

Figure 6-5: Algorithm to Assign Task Structures to Threads

show how the compiler tries to minimize the number task structures allocated.

A naive approach would assign a separate task structure to each Simple codeblock in a
procedure. This approach is correct, but it creates many more task structures than strictly
necessary. Since task structures must be allocated and initialized on procedure invocation,
having too many task structures leads to unnecessary overhead. A task structure can be
shared between two threads if it is certain that the two threads will not execute concurrently.
Not surprisingly, we can use control-flow analysis to find regions in the SMT code where
this conditions holds.

The analysis is shown in Figure 6-5 as the procedure assignTaskId. We assume that
each Simple codeblock, the basic container of instructions in our control-flow abstraction,
has a field taskId with the location of its task structure in the activation frame. The goal of
the analysis is to assign an identifier to each taskId field. The identifier represents a unique
offset in the activation frame representing a pointer to a task structure. This identifier is transformed into an index in the later stages of code generation. New identifiers are created using the `makeNewId` routine, but the compiler tries to re-use an existing identifier as much as possible to reduce the number of task structures that will be allocated. The inputs to the `assignTaskId` procedure are a codeblock \((cb)\) and an initial task structure identifier \((tid)\). The output is a task structure identifier that can be used by codeblocks that do not execute concurrently with \(cb\).

As in `definedVars`, the `assignTaskId` procedure performs a case analysis on the input codeblock. Simple codeblocks are the easiest: the codeblock's `taskId` field is set to the current task id which is then returned.

Conditional codeblocks, on the other hand, are the trickiest. The input `tid` is used to process the predicate recursively. In turn, the resulting task identifier from the predicate is used to process each of the alternatives. If all the alternatives return the same task identifier as the predicate, then that task identifier is the final one for the Conditional. Otherwise, the last task identifier returned by an alternative that is different from that returned by the predicate is returned. The algorithm becomes tricky in the case of Conditional codeblocks because it attempts to reuse task structure identifiers returned by the alternatives whenever possible.

In a Parallel codeblock, the first sub-block is processed recursively using the input `tid`. The remaining sub-blocks, however, are assigned fresh task structure identifiers since they might execute concurrently. The task structure identifier returned by the entire Parallel codeblock is that of the last sub-block since control flows out of this sub-block to the rest of the program.

Sequential and Threaded codeblocks are processed together because they share similar control flow patterns. The task structure resulting from processing one sub-block is passed as the input task structure to the next sub-block. The task structure identifier resulting from the last sub-block is returned for the entire codeblock.

### 6.4 Partitioning

So far, we have assumed the code generator was given a threaded program, and we have shown how to generate better SMT code from such a program. Now we discuss how a non-
threaded $\lambda^*$ program is transformed into a threaded one. The program analysis that drives this process is partitioning. In $\lambda^*$ terms, partitioning involves combining several statements in a parallel letrec into a single thread.

The key to a correct partitioning is that data dependences among tasks are obeyed, regardless of context. Remember that by “context” we mean the dependences that can be induced on a procedure by its caller or by its callees. Intuitively, a correct partitioning requires that we do not place code that produces a value after code that consumes that value in the same thread. In many cases, the analysis is simple, as in the following function:

```
easyPart = \lambda x.
  \{ a = b * x ;
    b = x + 5
  in a \}
```

There is a certain dependence between $a$ and $b$: $a$ cannot be computed, under any context, before $b$ is computed. The analysis is easy in this case because the expressions contain only strict operators like $*$ and $+$. The conclusion is that it is safe for $a$ and $b$ to be combined into a single thread, as long as the bindings are reorganized according to data dependences.

We use ‘$\sim$’ to specify this important relationship between statements. This threaded statement conjunction can be used to rewrite the example above without changing the value it produces or its termination properties:

```
easyPart = \lambda x.
  \{ b = x + 5  \sim
    a = b * x \sim
  in a \}
```

The real difficulty in partitioning is due to non-strictness. We noted that non-strictness introduces the notion of a context dependence, and these dependences are not evident by analyzing the code of a function in isolation. The mkPair function in Section 3.3 is a function that contains a context dependence. We repeat the code here, in $\lambda^*$ syntax.

```
mkPair = \lambda p q.
  \{ a = p * 2 ;
    b = q + 3 ;
    t = Pair(a,b)
  in t \}
```

Here, devoid of context information, $a$ and $b$ appear independent, so it would seem that they can be put in the same thread in any order, as in:
\[ \text{mkPair} = \lambda p \ q. \]
\[
\begin{array}{l}
\{ \ a = p \times 2 \quad \\
\quad b = q + 3 ; \\
\quad t = \text{Pair}(a,b) \}
\end{array}
\]

To compute \( a \), the code must get the value of \( p \). If \( p \) is available, then the multiplication occurs and control-flow continues directly to the code for \( b \). But if \( p \) is not available, then the thread will suspend. Since the computation for \( b \) is in the same thread following the suspension point, then it is also effectively dependent on \( p \). This is where the problem arises. In a context like:

\[ \{ x = \text{mkPair} \ y \ 9 ; \\
\quad y = \text{Prj}^2 x \\
\quad \ldots \}
\]

the computation will deadlock because it is the value of \( b \) in the body of \( \text{mkPair} \) that gets fed back into the function as argument \( p \). In other words, the context introduces a dependence from \( a \) to \( b \), so the code for \( a \) should execute after the code for \( b \).

We could try switching the order of the bindings in the thread, putting \( b \) before \( a \). But a context like

\[ \{ x = \text{mkPair} \ 8 \ y ; \\
\quad y = \text{Prj}^1 x \\
\quad \ldots \}
\]

will simply reverse the situation. This context introduces a dependence from \( b \) to \( a \) in the body of the function and will also lead to a deadlock if \( b \) and \( a \) are placed in the same thread. The answer, of course, is that \( a \) and \( b \) have to be placed in separate threads since the actual dependence is determined only at run-time.

The code for \( t \), the constructor expression that allocates and initializes the storage for the pair, must also be placed in a separate thread. The non-strict semantics of the language say that \( \text{mkPair} \) can return a value even if the arguments have not been computed. This means that the function must return a value if that value does not depend strictly on the arguments. Constructors are non-strict, so even if \( p \) and \( q \) are never computed, the function must still return a tuple, albeit an empty one. The conclusion, then, is that the constructor expression cannot be placed in the same thread as any computation that may suspend.
The complexity in a partitioning algorithm arises from the need to preserve correct non-strict semantics while generating long, efficient threads. Non-strictness forces algorithms to be conservative whenever the possibility of a context dependence arises, so the compiler must assume the context dependence exists even if it never arises at run-time. Effectively, this means that statements that might otherwise be placed in the same thread have to be separated.

To improve threading, some algorithms implement inter-procedural analyses that can often get around the problems of context dependences if all call-sites are known statically. However, inter-procedural analysis is always at odds with a modular language because the compiler must process an entire program at once. In addition, inter-procedural analysis is much more complex in a higher-order language like pH since functions can be passed as parameters or stored in data structures before they are applied.

6.5 Partitioning in pHc

We do not contribute any new partitioning results in this dissertation. Rather, we have concentrated on generating suspensive threads from partitioned code. The partitioning algorithm used in the current version of pHc is a simple intra-procedural demand-tolerance set algorithm. It was implemented by Jan-Willem Maessen and works directly on λ* programs.

To produce a threaded program, the pHc compiler proceeds in two phases. First, the partitioning phase analyzes the original λ* program and generates two data structures, the SameThread relation and the AlwaysAfter map. Second, the threading phase uses the SameThread and AlwaysAfter results to transform the original λ* program into one that uses '∴' statement conjunctions. The threaded program is then passed to the SMT code generator, which suitably modified as we have shown, produces better code. In this section, we discuss the meaning of the partitioning data structures in pHc.

The SameThread relation divides the set of identifiers in a program into equivalence classes. Two identifiers are in the same equivalence class if the code to compute their values can be executed in a single thread. More precisely, the computations represented by the identifiers in a SameThread equivalence class can be sorted and executed in dependence order, in any context, without creating a deadlock. Identifiers can represent function arguments, function results, or letrec statements.
Here is the SameThread relation produced by the partitioning algorithm for the function easyPart. It consists of a single equivalence class:

\{ a, b, easyPart, x \}

The identifiers a and b correspond to the left-hand sides of the bindings in the function. The identifier x is the function argument and the identifier easyPart is used to name the result of the function, effectively the in expression of the letrec block. The SameThread relation confirms our intuition that all the computations in easyPart can be placed into a single thread.

The second data structure produced by partitioning analysis is the AlwaysAfter map. The information it captures about a program is more difficult to understand than the SameThread relation, but it is still quite useful. The AlwaysAfter map maps an identifier to the set of identifiers whose computations can always precede it. Intuitively, this seems to express the same information as the SameThread relation, but in fact it does not. For the purposes of grouping statements into threads, the information in the AlwaysAfter map is weaker than that in the SameThread relation. If the compiler tried to form threads directly from just the AlwaysAfter map, the threads that would result would generally be smaller than those given by the SameThread relation because computations that shared no dependences could not be placed in the same thread.

The AlwaysAfter map essentially captures data dependences between identifiers, even across threads. In particular, if the AlwaysAfter entry for identifier x consists of the set \{ x, y, z \} (the identifier always forms part of its own set), then x cannot be computed until y and z have been computed. However, this does not imply that x, y, and z can be computed in the same thread. The values for y and z might have to be computed in separate threads based on their own sets of dependences. What the information for x does say is that it is reasonable to wait until y and z have been computed before spawning the thread that computes x. If spawned earlier, the code for x would suspend until y and z were available.

Still, the AlwaysAfter map and the SameThread relation are consistent in the following way. If y and z happened to be in the same equivalence class according to the SameThread relation, then it is certain that x would also be in the same equivalence class. Here is the AlwaysAfter map for easyPart.
In this case, we do not get any additional information beyond that contained in SameThread since all statements in easyPart can be placed in the same thread.

6.6 Partitioning fib

The partitioning analysis in the pHc compiler yields the following SameThread relation for the original, non-threaded version of fib:

\[
\{ p, t, x, x1, x2 \} \\
\{ f1, f2, r', r, fib \}
\]

Thus, fib can be split into two threads:

1. \{ p, t, x, x1, x2 \}: The variables x, p, and t together determine the discriminant of the Case expression. The Case is strict in the value of the discriminant, so it is also strict in the value of these variables. Consequently, these three can be placed in the same thread. Furthermore, computations in the branches of the Case can also be placed in the same thread. This means that control-flow can proceed from the discriminant to the branches without breaking the current thread. The statements for x1 and x2 can be placed in the same thread as the discriminant because they share a strict dependence on x with the discriminant.

2. \{ f1, f2, r', r, fib \}: The current partitioner in pHc lacks the sophisticated analysis of non-strict, recursive function calls needed to determine what otherwise seems obvious: that the whole of fib can be placed in a single thread. Instead, the partitioner takes the conservative approach of creating a new thread for f1, and f2, and for those values that are strictly dependent on the results of these two function calls. In fact, r', r, and fib are simply different names for the result of the function. This result is strict in f1 and f2 due to the primitive '+' function used to compute r'. A more powerful algorithm would be able to place all of fib into a single thread.
The \emph{AlwaysAfter} map for \texttt{fib} is:

\begin{align*}
\text{p} & \rightarrow \{p, x\} \\
\text{t} & \rightarrow \{p, t, x\} \\
\text{x1} & \rightarrow \{p, t, x, x1\} \\
\text{f1} & \rightarrow \{f1, p, t, x\} \\
\text{x2} & \rightarrow \{p, t, x, x2\} \\
\text{f2} & \rightarrow \{f2, p, t, x\} \\
\text{r'} & \rightarrow \{f1, f2, p, r', t, x\} \\
\text{r} & \rightarrow \{f1, f2, p, r, r', t, x\} \\
\text{fib} & \rightarrow \{f1, f2, fib, p, r, r', t, x\}
\end{align*}

The map shows the strict data dependences among different computations, even across \emph{SameThread} threads. For instance, notice that \texttt{r} is always after \texttt{t} since \texttt{t} is the discriminant of the \texttt{Case} expression for \texttt{r}. But \texttt{r} and \texttt{t} are not in the same thread according to the \emph{SameThread} relation. Below, we show how to use this information to combine these two bindings into a single thread. Also, notice that due to non-strictness \texttt{x1} does not appear in the entry for \texttt{f1}, nor \texttt{x2} in the entry for \texttt{f2}. Because the current partitioning algorithm does not work across function calls, it cannot make any assumptions about the strictness of arguments on any call. The conservative decision, then, is to place the function and its arguments in separate threads.

\section{Threading fib}

The partitioning analysis reveals what computations can be serialized. The threading algorithm, in turn, makes use of partitioning results to transform the original $\lambda^*$ program into one that uses `$\sim$' instead of `;'. For instance, with only the \emph{SameThread} relation, \texttt{fib} can be threaded as follows:

\begin{verbatim}
\texttt{fib} = \lambda \texttt{x} . \\
\texttt{\{ p = x >= 2 \sim} \\
\texttt{\texttt{t} = \texttt{Prj}^0(\texttt{p}) ;} \\
\texttt{\texttt{r} = \texttt{Case}(\texttt{t}, \texttt{x}, \{ \texttt{x1} = \texttt{x} - 1 \sim} \\
\texttt{\texttt{x2} = \texttt{x} - 2 ;} \\
\texttt{\texttt{f1} = \texttt{fib} \texttt{x1} \sim} \\
\texttt{\texttt{f2} = \texttt{fib} \texttt{x2} \sim} \\
\texttt{\texttt{r'} = \texttt{f1} + \texttt{f2} \sim} \\
\texttt{\texttt{in} \texttt{r'} \}} \}) \sim \\
\texttt{\texttt{in} \texttt{r}} \}
\end{verbatim}
The algorithm to produce this new version of \texttt{fib} is the following:

**Threading Algorithm for the \textit{SameThread Relation}**

1. In each parallel \texttt{letrec} block, group the statements that belong in the same thread.
2. Within a group of statements, order them according to data dependences (topological sort).
3. Within a group, replace parallel statement conjunctions (\texttt{;}) with threaded statement conjunctions (\texttt{~}). Across groups, the \texttt{;} remain unchanged.
4. Arrange the groups of statements so that the one containing the \texttt{in} expression is last in the block. In the example above, the \texttt{in} expression in the outermost block is called \texttt{fib}, and in the innermost block, it is called \texttt{r}.

In the outermost \texttt{letrec} of \texttt{fib}, the statement groups are \{\texttt{p,t}\} and \{\texttt{r,fib}\}. We cannot group all four statements together because the function calls in the expression for \texttt{r} overwhelm the capabilities of the simple partitioner. Likewise for the inner \texttt{letrec}. There the groups are \{\texttt{x1,x2}\} and \{\texttt{f1,f2,r',r}\}.

Clearly, the algorithm has created longer threads by combining bindings, but the results are not quite satisfactory. Despite the shortcomings of our current partitioner, our aim is to squeeze \texttt{fib} into a single thread. To improve the partitioning, we make a second pass over the code using the information in the \textit{AlwaysAfter} map.

This second pass is more complicated than the first, so we illustrate its behavior. Figure 6-6(a) shows the results of using \textit{AlwaysAfter} on \texttt{fib}. The key observation is that the use of this additional information allows us to combine the code for \texttt{p, t, r, and fib} into a single thread. This is certainly an improvement over using just the \textit{SameThread} relation. Because \texttt{fib} is a fairly simple example, we also show the behavior of the algorithm on a more complex, though artificial, example in Figure 6-6(b).

The algorithm for the second pass operates on an \textit{AlwaysAfter} graph built from the statement groups found in each \texttt{letrec} block using the \textit{SameThread} relation. The \textit{AlwaysAfter} graph is a directed graph where the nodes correspond to the statement groups in a \texttt{letrec} block and the edges correspond to the entries in the \textit{AlwaysAfter} map. For example, in the graph for \texttt{fib}, we place an edge from the node representing \{\texttt{p,t}\} to the node for \{\texttt{r,fib}\} since at least one of the statements of the second node is always after at least one of the
Figure 6-6: Threading Using the AlwaysAfter Map. Part (a) illustrates the behavior of the algorithm on fib. Part (b) illustrates its behavior on a more complex, though artificially contrived, input graph.
statements in the first node. In this particular case, the relationship is much stronger, though it need not be, since both \( r \) and \( \text{fib} \) are always after both \( p \) and \( t \). In the graph for the inner \texttt{letrec}, we insert no edges since no similar relation exists between any pair of statements in different nodes.

After building an initial \texttt{AlwaysAfter} graph, the next task is to coalesce nodes to produce new statement groups. Compared to the first pass using only the \texttt{SameThread} map, the coalescing algorithm should produce fewer groups but more statements per group (i.e., longer threads). It proceeds as follows:

**Threading Algorithm Using the \texttt{AlwaysAfter} Map**

1. Traverse the \texttt{AlwaysAfter} graph in breadth-first order, labeling each node with a timestamp indicating when it was first encountered.
2. Define the last parent of a node \( n \) to be the unique node \( p \), such that there exists an edge \((p,n)\) in the \texttt{AlwaysAfter} graph, and \( p \) was visited latest in the breadth-first traversal.
   
   If a node is a root of the \texttt{AlwaysAfter} graph, then it serves as its own last parent.
3. Traverse the nodes again, but in topological order. For each node \( n \), add it to the group containing its last parent \( p \) only if \( p \) was the last node added to the group. If \( p \) was not the last node added to the group, then form a new group initially containing only \( n \).

To generate the new program, we order the statements in each new group according to data dependences, join them with \`-\`, and separate the groups with \`;\`.

The effect of the \texttt{AlwaysAfter} algorithm is to link chains of statement groups that are safe to execute in order. The last parent serves two purposes. First, it marks the longest path to a node in the graph since it was computed based on information gathered from a breadth-first traversal. We want to add nodes to the longest possible chains to maximize the size of threads. Second, it prevents multiple unrelated children of one parent from being added to the same group. If nodes \( b \) and \( c \) are children of node \( a \) but have no edges between them, it is not always safe to place \( a, b, \) and \( c \) in the same thread.

The statement groups generated in the second pass with the \texttt{AlwaysAfter} map produce better code for \texttt{fib}:
fib = λ x .
  { p = x ≥ 2 ~
  t = Prj^0(p) ~
  r = Case(t, x, { x1 = x - 1 ~
                 x2 = x - 2 ;
                 f1 = fib x1 ~
                 f2 = fib x2 ~
                 r' = f1 + f2 ~
               in r' } ) ~
  in r ~
}

Now all the bindings in the outermost letrec block are in the same thread. This makes a lot of sense because the Case expression for r must always wait until t to produce a value.

Still, there is one remaining ';', in the inner letrec block. A third pass over the code uses available identifier information to find threads that are guaranteed not to suspend at run-time. The thread consisting of x1 and x2 is an example of such a thread. The compiler can determine that by the time control-flow reaches the code to compute x1 and x2, all values on which these statements depend will have been computed. In this case, the only data dependence is on x, and since x must be computed for the discriminant of the Case, its value will be available in the branches of the Case. Thus, the third pass can safely push the bindings for x1 and x2 into the next thread. The final version of fib consists of just one thread as we expected:

fib = λ x .
  { p = x ≥ 2 ~
  t = Prj^0(p) ~
  r = Case(t, x, { x1 = x - 1 ~
                 x2 = x - 2 ~
                 f1 = fib x1 ~
                 f2 = fib x2 ~
                 r' = f1 + f2 ~
               in r' } ) ~
  in r ~
}

Similarly, if the last thread in a block is non-suspensive, then it is combined with the preceding thread in the block.

This third pass of the threading algorithm might seem a little suspicious. Why is it that the information needed for this transformation is not expressed directly in the SameThread map? In this pass, we are actually using information beyond that considered by the parti-
tioner to remove extraneous threads. Specifically, we are using information about control flow in conditionals. In our compilation strategy, the code for Case expressions is structured so that control flows directly from the predicate (discriminant) to the alternatives without any intervening scheduling operation. This means that we can be certain that the values required to compute the predicate are available in each alternative. Case expressions occur often enough in $\lambda^*$ that this third pass becomes an important part of the threading process.

In summary, the partitioning analysis reveals what statements can be grouped together, but it is the threading algorithm that transforms the original $\lambda^*$ code into a threaded program. The threading algorithm consists of three passes. The first pass uses only the SameThread map to perform the basic grouping of statements into threads. It replaces ';' with '~' for threaded statements. The second pass considers cross-thread dependences in the AlwaysAfter map. It traverses an AlwaysAfter graph to discover chains of threads that can be combined into even longer threads. A final cleanup pass makes sure that any thread without a suspensive operation is combined with the thread preceding or following it. Non-suspensive code has trivial non-strict semantics, so such combinations are always safe.

6.8 When Programs Become Too Serial

Using partitioning and threading, we were able to generate much better code for fib than with the naive compilation strategy of Chapter 5. Unfortunately, the code has lost all parallelism. This situation arises because the only metric used to judge the quality of a thread is length. To the compiler, longer threads are always better than shorter threads, and in some cases, the partitioning and threading process will yield a single thread for a procedure. In previous pH (Id) compilation efforts using non-suspensive threads, this situation rarely arose because it is almost impossible to generate a single non-suspensive thread for an entire function.

To fix the problem of lack of parallelism, we insert a pass after partitioning and threading that splits a single thread containing multiple function calls (application expressions) into two threads, each with roughly half the function calls of the original. Thus, each thread will be spawned separately at run-time, and each set of applications will have the chance to run in parallel. In a doubly recursive program like fib, this change is enough to expose all
useful parallelism. This transformation is also safe according to the semantics of $\lambda^*$ since we can always add a parallelism to a letrec block that has been partitioned.

The code for the re-parallelized `fib` follows. We also include the new annotations for available identifiers. These must change since the order in which statements execute cannot be predicted exactly at compile-time once the parallelism is put back in.

$$
fib = \lambda x .
\begin{cases}
\{ p = x \geq 2 \} & \{ \} \\
t = \text{Prj}^0(p) & \{ p, x \} \\
r = \text{Case}(t, \\
x, & \{ p, t, x \} \\
\{ x1 = x - 1 \} & \{ p, t, x \} \\
x2 = x - 2 & \{ p, t, x, x1 \} \\
f1 = fib x1 ; & \{ p, t, x, x1, x2 \} \\
f2 = fib x2 & \{ p, t, x \} \\
r' = f1 + f2 & \{ f2, p, t, x \} \\
in r' \} & \{ f2, p, r', t, x \} \\
in r \}
\end{cases}
$$

We split the code between the two recursive calls to `fib`. This means that computation of `f1` and `f2` can run in parallel, and this is exactly what we want to happen to parallelize `fib`.

When we produce SMT code using the new available identifiers information, we must also insert additional touch instructions that were not needed in the single-thread code. For instance, the compilation of the expression for $r'$, now in the same thread with only `f2`, must check that `f1` is available. This touch instruction effectively synchronizes the two parallel activities for the recursive calls to `fib`.

$$
f1 + f2
$$

$$
touch(@rF[f1]) * 
load(r1, @rF[f1]) 
load(r2, @rF[f2]) 
plus(rV, r1, r2)
$$

The parallelization algorithm can be refined further to include a cost metric for procedure calls. Some calls are "cheap" because they perform little work and should execute within a thread. But "expensive" calls, like the recursive calls in `fib`, tend to contain lots of work and should be allowed to execute in parallel. The relative cost of calls can help determine a better place (other than the default "middle") to split threads. We have not implemented
this refinement in pHc.

6.9 Incremental Spawning

Up to now we have taken for granted the incremental spawning of threads in letrec expressions, but in fact, this order was chosen deliberately. In our earlier work [6], all statements in a letrec were spawned at the beginning of the letrec and then control flowed immediately into the in expression. The statements were assumed to be scheduled in some unspecified order, but dynamic synchronization made certain that data dependences were obeyed. Though a perfectly correct strategy in theory, it is a poor choice in practice.

The incremental strategy is a much better complement to the work-stealing scheduler implemented in the pHc run-time system for two reasons. The first advantage is that, assuming stealing and suspensions are infrequent, it makes it possible to run a pH program in a mostly serial order. Statements can execute from top-to-bottom, left-to-right, just like in C or Fortran. This predictable order makes optimization based on control-flow across threads possible. For example, the scheduler recognizes and optimizes the common case where the next thread scheduled is the last thread spawned. The second advantage is that when a steal does occur, the amount of work stolen is substantial. The work consists not just of a single statement, but conceptually, of all statements following it. This tends to decrease the total number of steals in an execution of a program. We discuss the scheduler in more detail in Section 7.2.

The scheduling strategy is not explicit in the SMT machine model because we wanted to allow implementations the freedom to experiment with different strategies. It might seem that we are undermining the SMT abstraction by specializing the SMT code for the work-stealing scheduler in the pHc RTS. However, this approach turns out to have a minor effect on the structure of the SMT code and a major impact on the efficiency of the executable code.

6.10 Summary

This chapter described how the compiler accomplishes one its most important tasks: to convert a pH program into long suspenseive threads. Long threads are better than short threads because they use processor resources more effectively and because the fixed-cost of
scheduling can be amortized over more instructions. In addition, suspensive threads can encompass rich-control flow and embody an optimistic synchronization strategy, features that are useful for mostly-strict programs on an SMP.

To make threads as long as possible, the compiler implements a partitioning analysis and a threading transformation which take short threads and merge them into long threads. In the context of \( \lambda^* \), this amounts to restructuring statements in letrec blocks and replacing certain parallel statement conjunctions (;) with threaded conjunctions (\( \sim \)). Undoubtedly, there is opportunity to improve both the partitioning and the threading phases of the compiler in the future, but even with a simple example like the \texttt{fib} function, it is clear that the existing algorithms provide a major improvement in the efficiency of the code. Sometimes programs can get transformed into a single thread, thus removing all parallelism, but these can be re-parallelized easily due to the original parallel semantics of the language.
Chapter 7

Executable Code Generation

This chapter describes how the pHc compiler generates executable code from SMT programs. It marks the transition in the compiler from our own SMT abstraction to abstractions offered by real systems. Much of the discussion focuses on low-level issues that must be addressed to implement pH, and we assume the reader is familiar with the C language and concepts like processes, address spaces, and memory models.

The problem of generating executable code consists of two parts. The first concerns the implementation of the three SMT abstractions—processors, synchronizing memory, and the global work-queue—on a real machine. The implementation of these constitutes the core of the pH Run-Time System and is shared by all programs compiled with pHc. In implementing the RTS, we have to consider issues such as how to exploit parallelism in a real machine, what constitutes an efficient scheduling algorithm, how to represent data in pH programs, and how to manage memory automatically.

The second part of the problem is how to emit code that implements the SMT instructions of a program. This part constitutes the rest of the back-end of the pHc compiler. We review the different options available for executable code-generation and then provide translations to executable code for each SMT instruction.

The last section in this chapter addresses the issues involved in re-targeting pHc to a new architecture. The first version of pHc was designed for Sun UltraSPARC SMPs. We chose Sun machines because they are readily available at MIT and because they are designed to scale up to 64 processors, thus allowing testing of the compiled code under a variety of configurations. We spent significant effort, however, making sure that the compiler was
portable.

Large portions of the pH Run-Time System described here, including the algorithms for garbage-collection algorithm and scheduling, are the work of Jan-Willem Maessen, the other member of the pHc team.

7.1 Implementing SMT Processors

In the SMT machine, a processor consists of a register file, a scratch memory, and a mechanism to schedule and execute threads. In the pHc implementation, an SMT processor is not mapped directly to a machine processor. Rather, it is realized by a C procedure and by a data structure containing the local state of the processor. Together, we refer to the procedure and its attendant state as an SMT virtual processor (SVP). The following sections describe the design of SVPs and show their implementation in C.

A note about terminology: whenever we refer to a real processor, like the UltraSPARC, we use the term CPU (central processing unit). This makes it easy to keep virtual processors, SVPs, separate from real processors, CPUs, in the discussion.

7.1.1 Exploiting Parallelism in an SMP

In order to exploit parallelism in a real machine, several SVPs must be scheduled concurrently on different CPUs by the operating system. Modern multiprocessor operating systems provide the kernel thread as the single object that is scheduled by the system. The classic notion of a process is implemented today by a pair of objects: a protected addressing context, representing the virtual address translations for the code, and a default kernel thread. Additional kernel threads must be created explicitly. Therefore, a program must contain several kernel threads in order to run in parallel on an SMP. There are two ways to organize a program with multiple kernel threads. A program can consist of multiple processes, each with a single thread, or it can consist of a single process with multiple threads (we ignore the multiple processes/multiple threads case). Each of these structures has its own advantages and disadvantages.

A program organized as multiple processes—where the default kernel thread in each process implements a single SVP—is perhaps more reliable than a single process with multiple kernel threads. Single-thread processes are well understood and thoroughly tested
abstractions in all operating systems. Such a program also contains multiple addressing contexts useful for protection, so that the internal state of an SVP cannot be corrupted by other SVPs. Unfortunately, multiple addressing contexts are also a disadvantage when implementing \( pH \). Our compilation model requires shared memory, and we rely on the fact that pointers can be passed from one SVP to another. Sharing a run-time heap is not too difficult using standard shared memory APIs, but sharing statically-specified (but mutable) program data is cumbersome.

Instead, we implement a parallel \( pH \) program as a single process with multiple kernel threads. This structure guarantees that all memory is shared by default as there is only a single addressing context for the program. We chose the standard POSIX \texttt{pthread}s package to improve the portability of our compiler across different operating systems, even though \texttt{pthread}s has well-known deficiencies. One particularly glaring POSIX problem arises in the code for the garbage collector. When one thread cannot allocate enough space, a garbage collection must be triggered. Ideally, this involves suspending all other threads immediately, performing the collection, and then restarting all threads. A POSIX thread, however, cannot be forced to suspend in mid-execution by some other thread. A POSIX thread can only suspend by waiting on a synchronization variable. This aspect of the POSIX specification tends to make synchronization in the garbage collector more complex.

### 7.1.2 SVP Internal State

The state of an SVP is defined by the \texttt{perThreadInfo} structure shown in Figure 7-1(a). The state accessible from the SMT instruction set consists of the first two fields of the structure. The first field describes the registers of the processor in the SMT model. These are implemented as an array of \texttt{pHob} elements. The type \texttt{pHob} is used to represent \( pH \) scalars and is described in Section 7.3. In practice, most of these SVP registers will be mapped to CPU registers by the code generator, but they must also exist as separate entities since several SVPs might time-share a single CPU during execution of a program.

The remaining declarations concern the state of the processor that is not directly accessible to the instructions. These declarations are explained throughout the rest of this chapter.
typedef struct {
    /* State accessible to instruction set */
    pHob virtual_registers[NUMBER_OF_VIRTUAL_REGISTERS];
    pHob scratch_area[SCRATCH_SIZE];

    /* State to implement the work-queue abstraction */
    Int32 localQueueLock;
    pHptr externalWork;
    UInt32 randomNext;

    /* State to implement the memory manager */
    pHptr chunkBot, chunkHP;
    chunkInfo *toSpace, *allocChunk, *fromSpace;
    size_t ToUsed;
    Int32 done_by_me, Int32 threadNum;
    thread_data this_thread;
} perThreadInfo;

(a) SVP Internal State

static Void
*rts_invoke_thread(Void *state)
{
    localState = (perThreadInfo *) state;
    rts_run_thread();
}

(b) SVP Code

{ Int32 j = 0;
    for (; j < numSVPs; j++)
        threadGCInit( states[j] );

    for (j = 1; j < numSVPs; j++)
        thread_start( ..., &rts_invoke_thread, states[j], ... );
}

(c) SVP Instantiation in the Run-Time System

Figure 7-1: SMT Virtual Processor Implementation
7.1.3 SVP Procedure

The code for an SVP is shown in Figure 7-1(b). This routine is designed to execute in its own POSIX (kernel) thread. It is passed a pre-initialized `perThreadInfo` structure, and it assigns it to a variable `localState`. The routine then invokes `rts_run_thread` (not shown here) to grab an initial SMT thread from the global work queue and begin executing. The effect of `rts_run_thread` is similar to that of issuing an SMT schedule instruction. Once an initial thread starts executing on a processor, it will become responsible for scheduling the next thread on that processor via a schedule instruction. Threads maintain this self-scheduling behavior until the program terminates.

The semantics of the variable `localState` are unusual and merit additional discussion. This variable has what we refer to as “thread-local” scope. This means that though it appears to be a regular global variable, its value is actually different within (local to) each kernel thread of the program. The C programming language used to implement the pH RTS does not offer thread-local variables. It is limited to global variables, shared by all threads in a program, or procedure-local variables, accessible only within a procedure activation. Thread-local variables are quite useful in parallel programming, so libraries like POSIX offer facilities to implement them. Unfortunately, the POSIX implementation is clumsy and slow.

Our solution is to exploit the thread-local state implicit in kernel threads. A kernel thread virtualizes the underlying CPU, so each kernel thread has its own register state. We implement `localState` by mapping it always to a CPU register, and we prevent any other code from using that register. Using the GNU C compiler gcc, this is accomplished with the following definition for `localState`:

```c
register perThreadInfo *localState __asm__(register-name);
```

Gcc makes sure to use the given register in any code where this definition is valid. For compatibility with other library code, it is wise to choose a register specified as callee-saved by the system calling conventions.

As noted by the implementors of Mercury [24], the use of register windows in the SPARC calling conventions makes it all but impossible to place a thread-local variable in a register. SPARC registers are divided into four groups: the global registers `g0-g7`, the output registers `o0-o7` used to pass procedure arguments, the input registers `i0-i7` used to receive
procedure arguments, and the local registers 10-17 used for local variables. On a function call, the register window hardware in the SPARC remaps the output registers of the caller to the input registers of the callee and allocates a fresh set of local and output registers. The global registers remain unchanged between caller and callee.

A thread-local variable cannot be placed in an output register since these are effectively caller-saved. Local and input registers are not useful either since we want thread-local variables to be visible across function calls into the pH RTS. Even global registers are problematic. None of these registers is specified as callee-saved: register g0 is always zero, g1-g4 are caller-saved, and g5-g7 are reserved by the operating system and apparently are used for multiplication, division, and modulus routines on older SPARCs. We also observed that register g7 is initialized and used by the POSIX library on SPARCs.

After lots of experimentation, we resorted to using another of the useful features of gcc: the ability to compile SPARC code that does not use register windows but that remains compatible with the standard calling conventions. This code-generation option is specified using the -mflat flag to gcc. Using the flat register model, we were able to specify any callee-saved register for localState. We chose 10, as shown in the following declaration:

```
register perThreadInfo *localState __asm__(10);
```

With this declaration, every kernel thread effectively gets its own version of the variable localState.

### 7.1.4 Instantiating the SVPs

The fragment of code in Figure 7-1(c) is responsible for instantiating the SVPs at the beginning of a program. The variable numSVPs is specified by a command-line argument or is determined automatically based on the number of processors in the system. Currently, we create at most one SVP per CPU. The thread_start routine is a wrapper around the POSIX pthread_create function. It creates a new kernel thread and executes rts_invoke_thread, with parameter states[j], in the new kernel thread. The parameter is a perThreadInfo data structure that has been previously initialized.

It is important to make clear the differences between kernel threads and SMT threads. Kernel threads are used to implement SVPs and SVPs in turn are used to execute SMT threads. Only a small, fixed number of kernel threads are created during the execution
of a $pH$ program. In contrast, thousands or millions of SMT threads might be spawned as a program runs. Though “lightweight” by operating system standards, kernel threads, each of whose state includes at least all CPU registers and a stack, are massive compared to the gossamer-like SMT threads. The context switch time between SMT threads is, consequently, much faster than between kernel threads. Finally, kernel threads are scheduled by the operating system, while SMT threads are self-scheduled without any operating system intervention.

7.2 Implementing the Global Work-Queue

The second major abstraction in the SMT machine is the global work-queue. Threads that are ready to run are placed in the global work-queue by one of three instructions: a spawn instruction; an istore instruction updating a memory cell with a deferred list; or a decbar instruction discharging a barrier on which a touchbar had previously executed. A thread is removed from the work-queue by a schedule instruction. The processor that eventually schedules a thread is not necessarily the same processor that spawned it. In fact, a thread that suspends many times may be resumed on a different processor each time. In this way, parallelism in the SMT machine is completely symmetric.

The design of this portion of the $pH$ RTS was influenced heavily by Cilk[12]. We discuss this influence in more detail in Section 9.3.

7.2.1 Randomized Work-Stealing

In practice, exploiting locality is often the key to achieving good performance, and our implementation distributes the work queue across all SVPs for this purpose. Conceptually, the idea of a single, global queue is preserved, but during execution, most threads are executed by the SVP that spawned them. Each SVP has its own local queue, and threads enabled by instructions executing on a particular SVP are always placed into its local queue. When a schedule instruction executes, a thread is dequeued from the head of the local queue according to the last-in-first-out (LIFO) policy. If the queue is empty, the processor must retrieve work elsewhere. This is the basis for the work-stealing scheduler in the $pH$ run-time system. If the local queue is empty, the current SVP becomes a thief and steals a thread from the queue of some other SVP, the victim. The victim is chosen randomly from all
other SVPs, and the thread is taken from the tail of the victim’s local work queue. In other words, the thread stolen is the oldest thread in the victim’s local queue.

Figure 7-2 shows most of the code used in the pH RTS to steal work. We have elided the code to traverse the queue itself, but this is fairly straightforward. The code in the figure consists of two nested while loops. The outer loop is an infinite loop, so that the steal process will not terminate until it finds work or until the program exits. The inner loop forces the SVP to pause every so often. If the SVP does not find work after trying on average twice for every processor, then it executes an operating system yield call. On a yield, the SVP gives up the CPU and allows the operating system to schedule another kernel thread. The SVP will be rescheduled later and will resume its attempts to steal work. This is the simplest possible back-off scheme. Other systems, like pFluid and Cilk, use a back-off scheme where the waiting time increases proportionally to the number times a processor has failed to acquire new work.

The core of the steal code is the critical region within the inner while loop. This part of the code accounts for the fields localQueueLock, externalWork, and randomNext in the perThreadInfo data structure shown in Figure 7-1(a). The code computes a random victimId, a number between 1 and the number of SVPs, and uses it to index into a global array of all SVP states. The randomNext field is used to hold the state of a random number generator for each SVP. This is necessary because it is difficult to find a portable random number generator in Unix that is also safe in a multithreaded environment.

With the state of a possible victim in hand, the code then attempts to acquire the localQueueLock protecting the local queue of the victim. The lock is essential because the victim and one or more thieves may be trying to access the victim’s local queue at the same time. For performance, the lock is implemented directly in the RTS using atomic memory operations, not POSIX lock objects. This approach is about four times faster than the using POSIX. After acquiring the lock, the thief SVP is free to traverse and modify the victim’s local work queue.

The work queue is accessed by a thief via the externalWork variable. It is simply a linked list of task structures. In a typical implementation, the code elided in Figure 7-2 would traverse the linked list and would extract the last element, if any, as the stolen work. However, our scheduler is slightly more complex. It includes a provision whereby task data structures can be marked as stealable or local. Stealable tasks are those judged
/* Try to steal from a victim. */
while (1) {
    /* We try to steal this many times. If we don’t
     * succeed, then we yield. */
    UInt32 yieldCount = numThreads * 2;

    while (yieldCount--) {
        UInt32 victimId = randomVictimId( myId );
        perThreadInfo *victimState = allThreadState[ victimId ];

        /** BEGIN CRITICAL REGION */
        if (tryQueueLock( &(victimState->localQueueLock) )) {
            pHptr prevElt = victimState->externalWork;
            pHptr currElt = asPtr(slot(prevElt,snext));
            pHptr stealCandidate = NULL;
            pHptr stealCandidatePrev = NULL;

            ... /* Code elided: traverse queue and find a steal candidate */
            ...

            if ( stealCandidate != NULL ) {
                /* Stole some work */
                slot(stealCandidatePrev,snext) = slot(stealCandidate,snext);
                work = stealCandidate;
                releaseQueueLock(&(victimState->localQueueLock));
                goto schedreturn;
            }
            else { /* Didn’t find anything: try another victim. */
                releaseQueueLock( &(victimState->localQueueLock) );
            }
        }
        /** END CRITICAL REGION **/
    }

    /* If we get to this point, then we have tried to steal
     * too many times. We yield the processor. */
    yield();
}

Figure 7-2: Work-Stealing Code in the Run-Time System
by the compiler to contain enough work to be worth executing on another processor. Local threads contain little work, and the overhead of executing them elsewhere would outweigh any advantage. Thus, the traversal of the victim’s queue actually searches for the oldest enabled thread that is marked as stealable.

Once it finds a valid thread to steal, the code updates the queue, releases the lock, and jumps to the label sched_return. The code at sched_return (not shown) extracts the instruction pointer from the task structure, as well as the frame pointer and barrier register, and transfers control to the thread. If the thief finds no valid thread to steal, it releases the lock and begins the steal loop again, looking for another victim.

7.2.2 Exploiting Locality

Using local queues with work-stealing, we exploit locality in the program in several ways. First, most threads spawned by one SVP will be executed by that SVP. This means that storage used by those threads is likely to reside in the cache of that processor and does not have to be reloaded from main memory every time a thread is scheduled. Second, the use of a local queue allows us to implement a fast path through the scheduler to handle the most common case where the next thread scheduled was the last thread spawned. In this fast path, less state has to be reloaded from the task structure into the SVP registers since the next scheduled thread and the thread that just terminated have much in common.

Finally, the use of local queues allows many queue operations to be executed without the need for locking. As shown in Figure 7-3, only a portion of the tail of the local queue is accessible via the externalWork variable. The front of the queue, where the SVP places and extracts its threads, is not accessible to thieves, and thus, can be modified without paying the cost of acquiring the localQueueLock. Periodically, the SVP updates its externalWork variable to make more work available to thieves. Using this technique, it is possible to allow most accesses to the queue to proceed without incurring the cost of atomic memory instructions. Only steals and updates to the externalWork variable, both of which we expect to be infrequent, require locking the queue.
7.3 \( pH \) Data Representation

The last abstraction in the SMT model is the synchronizing shared-memory. This part of the model maps directly to the shared-memory of an SMP. The real problem in the implementation is finding a good representation for \( pH \) data, the contents of memory. In simple languages like C, basic types like pointers, characters, integers, and floating point numbers are mapped directly to the underlying types supported by the target processor. Languages like Haskell complicate matters by including polymorphism, so values of unknown types can be stored in data structures and passed to functions. \( pH \) presents the additional complication of synchronizing data structures. And to top things off, a complete \( pH \) implementation requires automatic memory management since the language does not include provisions for explicit deallocation of data structures.

These factors have shaped the unique representation of \( pH \) scalars. To simplify the implementation of polymorphism, scalars have a fixed size. We use 64-bits because we want to represent IEEE double-precision floating-point numbers directly. The other scalar types include characters, fixed-precision integers, single-precision floating-point numbers.
and pointers. Fixed-size scalars do carry the cost of increased memory usage, both in terms of footprint and bandwidth. We are not worried about footprint, but bandwidth may become a concern. To simplify the implementation of synchronizing memory and garbage collection, pH scalars include a multi-bit tag field. Tags are used to represent both presence information (full or empty) for I/M-Structures and type information (pointer or non-pointer) for copying garbage collection.

7.3.1 An Efficient Encoding of Tagged Data

There are two basic ways to store tagged data in untagged memory. Tags can be stored in a separate region of memory, as in Shaw's implementation of Id [59], or tags can be combined with data and stored in the same word of memory. Early on, we discarded the idea of storing tags and data in separate memory regions due to the problems posed by the weakly-coherent memory systems in current microprocessors. Our code requires atomic access to tags and data in order to guarantee that these values are consistent when viewed by all processors. Weakly-coherent memory systems make it prohibitively expensive to implement atomicity across multiple words of memory. Thus, we settled on the second approach: the tag information for a memory slot is encoded within the contents of the slot.

Typical encodings of tagged data make use of some number of high bits or low bits of a
data word for tag storage. A 64-bit implementation might use the low-order 4 bits and leave 60 bits available for data, as shown in Figure 7-4. We were not satisfied with this solution. The first problem is that the range of scalars is significantly curtailed. In the figure, we show the range of fixed-precision integers if tags bits are simply stolen from the low-order (or high-order) bits of a word. The range is only 6.25% of its original size. The second problem is that this tagging scheme requires untagging/retagging steps on every scalar operation. To guide the design of an alternative scheme, we came up with the following list of requirements:

1. Floating point numbers must be represented directly, not boxed.
2. The structure and precision of floating point numbers must not change.
3. The range of integers and pointers should be curtailed as little as possible.
4. Pointer/non-pointer checks should be fast.
5. Full/empty checks should be fast.
6. Untagging/retagging overhead for arithmetic operations should be kept to a minimum.

We placed great importance on the integrity of floating-point values because Id and pH have been used to implement scientific applications. This led us to look closely at the structure of IEEE-754 floating point numbers (double-precision), and we observed that not all patterns of 64 bits are valid floating point values. The IEEE standard specifies that positive and negative double precision numbers with an exponent field of all ones and a non-zero mantissa are Not-a-Numbers (NaNs). NaNs are invalid, but well-specified, values and are the result of certain computations such as arithmetic on infinities or NaNs. NaNs have several very interesting properties. First of all, there are lots of them, roughly $2^{53}$ in the double precision representation. Second, when interpreted as twos-complement integers, the “positive” NaN bit patterns (+NaNs, for short), those with a sign-bit of zero, correspond exactly to the very largest $2^{52}$ positive integer bit patterns. Third, most CPUs use only a handful of NaNs to signal errors.

Based on these observations, we created a data representation that uses NaNs to represent tagged pointers. As shown in Figure 7-5, the address field of a pointer in pH is 44 bits wide. The remaining bits are divided into two fields: a 7-bit tag and a fixed 13-bit NaN pattern. The large tag space exceeds our current needs and might we reduced in the future in order to increase the size of the address field. The handful of tags we have defined are
Encoding tagged pointers as NaNs turns out to be a surprisingly effective technique. On a point-by-point basis, it fulfills the requirements of our data representation quite well:

1. Floating point numbers are represented directly.

2. Because NaNs are outside the range of valid floating point numbers, the structure and precision of floats is untouched.

3. Integers are curtailed slightly to the range $[-2^{63}, 2^{63} - 1 - 2^{51}]$. This restriction prevents the bit patterns of NaNs and integers from overlapping since the NaNs used to represent pointers use the same bit patterns as large integers. But the restriction is tiny. Over 99.98% of the integers in the original 64-bit range are still available. On the other hand, pointers are limited to 44-bits. This is a severe limitation when compared the full 64-bit addressing space, but we believe it is acceptable for at least the next decade, especially in a language where memory is managed by the compiler/run-time system.

4. Pointer/non-pointer checks are fast. If a bit pattern, interpreted as a signed two's-complement integer, is greater than $2^{63} - 1 - 2^{51}$, then it is a pointer. Otherwise, it is a valid integer or floating-point number.

5. Full/empty checks are similarly fast, requiring a simple range check in the worst case.
<table>
<thead>
<tr>
<th>Bit Pattern (64-Bits, Hexadecimal)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000000000000000</td>
<td>Integer or floating point zero. We never have to tell these apart, so ambiguity is not a problem.</td>
</tr>
<tr>
<td>0x7FF0000000000000</td>
<td>The integer 9,218,868,437,227,405,312 or positive infinity in the IEEE floating point representation. Again, the ambiguity between these two values is not important.</td>
</tr>
<tr>
<td>0x7FF7FFFFFFFFFFFF</td>
<td>The integer 9,221,120,237,041,090,559, the largest positive integer that can be represented directly in our data encoding, or a signaling NaN.</td>
</tr>
<tr>
<td>0x7FF8000000001CF0</td>
<td>A pointer to a data structure at memory address 0x1CF0 tagged with the normal tag.</td>
</tr>
<tr>
<td>0x7FFE000000000000</td>
<td>The unique empty value emptyval. No value is available is the cell.</td>
</tr>
<tr>
<td>0x7FFE000000001CF0</td>
<td>A pointer to the head of a deferred list (a task structure) at memory address 0x1CF0. Task structures are allocated on the heap just like any other data structure, but when they form part of a deferred list, pointers to task structures are tagged with the empty tag.</td>
</tr>
</tbody>
</table>

**Figure 7-6: Examples of Tagged Data**

They are described in the next section.

6. There is no untagging/retagging overhead for arithmetic operations because the representation of valid integers and floating-point values is unchanged. For safety, the compiler can insert overflow checks to make sure that a large positive integer does not overflow into pointer space or that a pointer is not incremented beyond its 44-bit range, but these checks are usually omitted.

Conversations with Prof. William Kahan of Berkeley and Dr. Cleve Moler of MathWorks, both members of the original IEEE-754 committee, confirmed that the committee crafted NaNs to allow encoding of useful information. Few language implementations seem to exploit this useful feature of the standard.
7.3.2 Tags for I/M-Structures

We have defined two tags for pointer values that might be stored in I-Structures or M-Structures. The first is the empty tag (value 0x7E). If the address field of a pointer with an empty tag is all zeros, then the bit-pattern corresponds to the unique empty value "emptyval". This value is used to initialize freshly allocated memory. If the address field in non-zero, then the bit-pattern represents a pointer to a list of deferred computations. This is the list of task structures representing the threads that have attempted to access the value of the memory location before it has been computed. The second tag is the normal tag (value 0x00). This tag is used for the results of normal allocation from the heap. It is used to mark pointers to a standard, program-allocated data structures.

Figure 7-6 shows several examples of data tagged with empty and normal. It is interesting that while we have an explicit empty tag, we do not have a complementary full tag. Memory cells containing valid integers, floats, or pointers tagged with normal are implicitly full. The presence of valid data makes this status manifest. Otherwise, a cell would contain emptyval or a pointer tagged with empty to a deferred list.

Given this rationale, one might wonder why the normal tag is required at all. As long as pointers to regular data structures do not exceed 44-bits, they will not conflict with the emptyval or with pointers to deferred lists. Normal pointers could appear as implicitly full values in memory cells. The reason for the existence of the normal tag is garbage collection. In a copying collector, pointers are traced while non-pointers are simply copied during a traversal of the heap. The garbage collector must be able to distinguish pointers from non-pointers unambiguously, so as a consequence, all pointers are tagged.

7.4 Choices for Code Generation

So far in this chapter, we have tackled the first problem of generating executable code from SMT programs, namely the implementation of the SMT machine on real hardware. This section begins the discussion of the second part of the problem: the implementation of the SMT instruction set. We considered three solutions for translating SMT programs into programs that run on real machines: an interpreter for SMT instructions, a native code compiler, and a C-based compiler.

An SMT interpreter has the advantages of flexibility, portability, and simplicity. In this
solution, the compiler emits a compact binary representation of the SMT instruction set which is then interpreted by the run-time system. The approach is flexible because SMT instructions can be implemented directly in the interpreter; it is portable because across architectures, the only part of the system that changes is the interpreter, not the compiler; and it is simple because most of the code is written within the clean abstractions of the SMT machine. The acceptance of Java and Visual Basic in the past three years has lent widespread credibility to interpreted languages. The problem with interpreters, of course, is their poor performance when compared to true compiled code, and it was for this reason that this option was not pursued in the development of pHc.

Compilers that generate native code (or assembly language) are at the other end of the performance spectrum from interpreters. By emitting native code, the compiler has fine control over the execution of the program and can generate the fastest code possible. In a multithreaded language like pH, this level of control is often useful since control-transfers and thread context-switches must be handled with great care to avoid overhead. But because they must manage so many details in code generation, native code compilers are difficult and time-consuming to build. It also takes much longer to re-target a good native code-generator to a new architecture. Given the limited resources at our disposal in the pHc project, we concluded that we could not generate native code.

The option we chose in our code-generator was to emit C code for SMT programs. C is simple and has almost "transparent" semantics, making it easy to predict the performance of programs written in the language. As a compiler target, it serves the role of a high-level assembly language. Good C compilers exist for just about any platform, so it offers good possibilities for re-targeting pHc to different architectures. Generating C code is also a fairly straightforward process when compared to generating assembly language. A C-based code-generator for pHc can pass a lot of tedious work—like register-allocation and instruction-scheduling—onto the C compiler. The major problem with C is that it was not designed as a multithreaded language. Therefore, it includes no built-in notion of threading, lightweight contexts, or scheduling. All this support has to built using libraries like POSIX or from scratch. In the next section, for example, we tackle a critical problem in executing C code in a multithreaded environment: avoiding the allocation of stack frames on procedure calls.

C has been used as a target in compilers for a variety of complex languages including
Haskell [49], ML [67], Scheme [11], and Mercury [24]. In all these efforts, the implementors
found that a C code-generator was easier to develop and port compared to a native code-
generator. They also found that code performance was significantly worse than native code
if they limited themselves to generating pure ANSI C, but that it improved dramatically if
they used extensions available in some compilers like the GNU C Compiler (gcc). Indeed,
this approach to code generation has become so popular that Peyton Jones et al. [51] have
proposed C--, a portable assembly language loosely based on C syntax that corrects many
of the deficiencies of C in this role.

The code generated by pHc relies on many of the features provided by gcc. We make
particularly heavy use of the ability to take the address of a label and store it in a data
structure and of the ability to perform indirect jumps via computed-goto statements. Both
of these features are used to implement thread scheduling and suspension. We also rely
on the seamless support for 64-bit values provided by the type long long in gcc, and on
the ability to place specific global values in registers. We have few regrets in choosing
to specialize our code to gcc. The compiler is widely available, free, and it consistently
produces high-quality scalar code for just about any architecture.

7.5 Avoiding Stack Allocation

Activation frames for pH procedures cannot be allocated on a stack because the lifetime of
a callee’s frame can exceed that of its caller. In our implementation, frames are allocated
on the heap and garbage-collected just like any other data structure. The implication for
pH procedures is that, though they are emitted as C procedures, they cannot be called
using the standard C procedure calling mechanism because the standard entry point to a
C procedure includes the code to allocate a frame on the stack.

Our solution to this problem is to bypass the standard procedure entry point by adding
a new one, special to pH. We illustrate it in Figure 7-7 with fib, our usual Fibonacci
example. First, using the gcc inline assembler facility, we insert a label in the C code
for fib as the first statement in the procedure. Given the way code is generated by gcc,
this label will appear immediately after the stack allocation code normally emitted at the
beginning of every procedure. This label is the pH entry point of fib. The name of this
label is constructed using a standard convention so that references in other modules to this
Void Main_fib_PROCSHELL(Void) {
    _asm_(".global Main_fib.entry_point");
    _asm_("Main_fib.entry_point:");
    ...
}

(a) $pH$ Entry Point in Definition of $fib$

extern Void Main_fib.entry_point(Void) __asm__("Main_fib.entry_point");

(b) $fib$ Entry Point Declaration in Calling Module

goto *((Void*) Main_fib.entry_point);

(c) Invocation of $fib$ in Calling Module

Figure 7-7: Alternative Entry Points to $pH$ Procedures

entry point will be generated correctly.

Second, in any module that calls $fib$, we generate a declaration for the label we created. In this way, we introduce a valid identifier for the $pH$ entry point of $fib$ into the C compiler's namespace, and we use the __asm__ extension to assign a specific name to this identifier. This declaration is essential since our system supports separate compilation. Without it, the C compiler could not emit a proper reference to $fib$ in a calling module.

Finally, a call to $fib$ is coded using a gcc computed-goto. The statement in Figure 7-7(c) tells gcc to generate code to jump directly to the address represented by the symbol $Main_fib.entry_point$, which was declared earlier in the module. At link time, this symbol will be resolved to the address of the entry point we defined in the C code for $fib$.

This three-part technique allows us to transfer control to a $pH$ procedure without allocating any space on the stack, just as we wanted. One last detail completes the scheme. Even though we circumvent the stack frame allocation as we transfer control from caller to callee, the code generated by the C compiler naturally expects a stack frame to be available for local variables, register spills, and scratch space. We address this problem by allocating a single large stack frame for each SVP. This is the purpose of the routine $rts_run_thread$ referenced in Figure 7-1(b). Thus, all code executing on a particular SVP shares the same C stack frame. This might seem dangerous, but we are careful to save data that lives across
suspension points to the heap (in frames or task structures). In addition, only one thread from one \textit{pH} procedure can access the stack of a particular SVP at a time.

We only encountered one unexpected problem in the implementation of our no-stack calling convention: support for shared libraries. Shared libraries are used extensively in the compiler to implement the \textit{pH} Prelude and the run-time system. Shared-libraries make for much faster linking and loading of programs at the price of slightly slower procedure calls and global symbol access. In Solaris, shared libraries require the use of Position Independent Code (PIC), so gcc inserts extra instructions to support PIC along with the stack allocation code at the beginning of every procedure. Since our scheme for \textit{pH} procedure calls bypasses this code, we must insert code explicitly into our program to support PIC. We were able to add this support using a few macros.

We have had excellent results using our calling convention. However, there is some question about the efficiency of heap allocation and garbage collection of activation frames. On one side, Appel et al. [3, 5] argue that it's a great idea because among other things as memories gets larger, the overhead of garbage collection decreases. Others like Miller and Rozas [39] disagree. We chose heap-allocated frames in our compiler not for reasons of performance but because this strategy makes our generated code and the run-time system very clean. We also get the additional benefit that we can avoid the problem of tracking procedure termination.

An activation frame can be deallocated only when all computation using the frame has terminated, but detecting whether a procedure has finished executing is not easy in a multithreaded, non-strict language like \textit{pH}. In compilers for \textit{Id}, termination detection was implemented via the signal mechanism explained in Section 3.5.1. In our implementation of \textit{pH}, we found it natural to let the garbage collector take care of this problem. If no references to an activation frame are found during garbage collection, then the corresponding procedure instance must have terminated and the frame can be deallocated. Turbak [73] uses a similar approach to compute the rendezvous (synchronization) condition of \textit{synchrons}, first-class barrier objects.

In contrast to pHc, activation frames in pHluid are allocated in a special area called the \textit{store}. This region is managed explicitly by the run-time system and is separate from the garbage-collected heap. Frame deallocation is triggered by a termination signal in the code. Goldstein et al. [21] also discuss explicit frame management in the Berkeley Id compiler.
using stacklets to implement a cactus stack. A cactus stack consists of a group of small stacks linked into a tree structure related to the call tree of a program. The advantage of a cactus stack is that, when parallel function calls can be “inlined” on single processor, frame allocation can be turned into stack allocation. When execution diverges from the LIFO stack model, a new stacklet is created and grafted onto the tree, and stack allocation can continue. In the common case, when a procedure returns, its frame on the stacklet is simply popped-off. Of course, this technique also presumes a mechanism for detecting procedure termination.

7.6 Implementing SMT Instructions

The previous sections established the framework for the executable code produced by pHc. The abstractions of the SMT machine were given concrete implementations; pH scalars were designed to include useful tag information; and C with gcc extensions was chosen as the form for emitted code. This section shows how SMT instructions are translated into C code. This code is then passed through gcc and linked with the pH run-time system to produce executable programs for Sun UltraSPARC SMPs.

The descriptions in this section follow a format similar to that of Chapter 4. Effectively, we implement the RTL description of the SMT instruction in C code. A typical instruction is given followed by its compilation to C code.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>C Code for Instruction</th>
</tr>
</thead>
</table>

7.6.1 The pHob Type

The C code generated by the compiler relies on the implementation of pH objects given by the pHob type in Figure 7-8. The pHob type is a C union consisting of four slots: a signed 64-bit integer slot, a double-precision floating point number slot, a “pure bits” 64-bit slot divided into two 32-bit words, and a pointer slot. In a union, these four components occupy the same 64 bits of storage, but this storage can be treated as if it had four different types.

The pointer slot merits some additional discussion. It is divided into two 32-bit fields, hiPtr and loPtr. This seems odd given that the definition of NaN-tagged pointers in Section 7.3 specified 44-bits of address. However, our compiler target is the UltraSPARC
typedef union phobjunionname {
    Int64 intSlot;
    Double floatSlot;
} pHob;

typedef pHob *pHptr;

#define asInt(x) ((x).intSlot)
#define asFloat(x) ...
#define asPtr(x) ...
#define isEmpty(x) ...
...

Figure 7-8: pHob: The Implementation of pH Scalars

processor, and our version of Solaris (2.6) only uses 32-bit addresses. Therefore, we can
dispense with the extra 12 bits available for pointers, and we can treat the low-order 32-bits
as the full address. The upper 32-bits hold the tag information for the pointer and zeros
for the 12 unused address bits.

Along with the type definition of pHob, there are a number of utility macros used by
the code. For example, the asInt macro “converts” a pHob into a 64-bit integer in C by
extracting the intSlot field of the union. Most of these macros do not generate any code.
Rather, they exist to uphold the type discipline of C. Other macros, like isEmpty, are
used to determine if a memory location is empty or holds a deferred list. These macros do
generate a small amount of code to check the tag field of the value read. All of the macros
are explained as they appear in the following sections.

7.6.2 Arithmetic and Move: plus, etc., and move

An arithmetic or move instruction is emitted as a single line of C code. Here is an example
of a plus instruction (integer addition) in SMT and in C:
plus(rV, r1, r2)

RV = (pHob) ( asInt(R1) + asInt(R2) );

The SMT registers RV, R1, and R2 are shortcuts to access the virtual_registers state of the current SVP (Figure 7-1). These are defined as:

```
#define RV (localState->virtual_registers)[0]
#define R1 (localState->virtual_registers)[1]
#define R2 (localState->virtual_registers)[2]
```

The type of RV, R1, R2 is pHob, so the macro asInt enables these to be treated as integers for the addition. The cast back into pHob to restore the correct type of the pHob union.

The move instruction also generates very simple code, except this time, no casting is necessary. The value of one pHob is transferred to another:

```
move(r1, r2)
R1 = R2 ;
```

### 7.6.3 Control Flow: jump and switch

The jump and switch instructions are used to manage control flow in SMT. The jump instruction takes an optional condition, a target address, and an optional return link register. In the example, we show an instance of jump with all optional components:

```
jump((r1 < r2), r3, r4)
if ( asInt(R1) < asInt(R2) ) {
  obPtrNoTag(R4, &IL.15);
  goto *((Void *) asPtr(R3));
}
IL.15:
```

The conditional jump is translated into a C if statement that compares R1 and R2 as integers. If the condition is false, control passes to the next statement after the if. When the comparison is true the code proceeds to generate the return link address. To do this, the compiler creates a unique internal label IL.15 that marks the code immediately after the jump. Using the gcc extension &&, it computes the address of this label and stores it in R4 with the obPtrNoTag macro. The effect of this macro is to create a ptrSlot object of the pHob union in R4 by assigning the proper values to the hiPtr and loPtr. The loPtr
gets the address of IL_15 and the hiPtr gets zero. Pointers to code are treated as integers (untagged) because they are not to be traced by the garbage collector. The obPtrNoTag macro generates the following C code when it is expanded fully:

```
R4 = (pHob)((struct phobjptrslot) {0, &IL_15});
```

Next, the code jumps to the target address contained in register R3. The effect of the asPtr macro is to remove the NaN pattern and the tag from a pHob so that only the address information is left.¹ Unlike the asInt macro, this conversion does involve a tiny amount of computation. Then, using the computed-goto extension supported by gcc, the code transfers control to that address. In standard C programs, the target of a goto statement can be only a static label identifier.

The code for the switch instruction is even simpler than jump since it maps well to the C switch statement:

```
switch(r1,SMT.L_32,SMT.L_35,SMT.L_51)
{
    case 0: goto SMT.L_32; break;
    case 1: goto SMT.L_35; break;
    case 2: goto SMT.L_51; break;
    default: SWITCHERROR(__FILE__, __LINE__);
    exit(1);
}
```

The compiler always adds a default case to handle the error where the discriminant of the switch does not correspond to any alternative. This type of error is due to a problem with the compiled code or with a corruption of the heap. It does not reflect an error in the original source program. In our prototype implementation, these sorts of checks cost little and help catch errors that might otherwise be detected much later in the code.

### 7.6.4 Scheduling: spawn and schedule

The spawn instruction is coded as a procedure call into the run-time system. This approach centralizes and simplifies the debugging of this subtle piece of code. The spawn_RTS procedure takes care of adding the given task structure to the front of the local scheduling queue.

The implementation of the spawn instruction in C is:

---

¹Actually, the asPtr is not needed here since R3 should contain an untagged code pointer. For the present, we keep it in the interest of safety.
The `spawn` instruction terminates the current thread and extracts a new one from the work queue. We code it, using the gcc computed-goto extension, as a branch into the scheduler code in the run-time system. We use a branch instead of a procedure call because the control never "returns" to the current thread. Rather, the run-time system will give control to a new thread.

```
spawn(r1)
spawn_RTS( R1 );
```

The `schedule` instruction terminates the current thread and extracts a new one from the work queue. We code it, using the gcc computed-goto extension, as a branch into the scheduler code in the run-time system. We use a branch instead of a procedure call because the control never "returns" to the current thread. Rather, the run-time system will give control to a new thread.

```
schedule

goto *((Void *) schedRTS);
```

The `sched_RTS` entry point for the scheduler is declared as

```
Void sched_RTS(Void) __asm__("sched_RTS");
```

so that the C identifier `sched_RTS` corresponds to the assembly language label `sched_RTS`. That label is defined in the body of the scheduler using the gcc inline assembly extensions:

```
__asm__(".global sched_RTS");
__asm__("sched_RTS: ");
```

Thus, the SMT instruction `schedule` gets compiled into a single `goto` in C which causes a direct transfer of control from the program to the run-time system.

### 7.6.5 Memory Allocation

The `allocate` instruction requests memory from the memory manager in the run-time system. The code generated is slightly complicated because it must handle the case when no memory is available. In this case, the code must save any live register state to the scratch memory of the SVP and then call the run-time system again, possibly triggering a garbage collection. Instead of using the entire SVP register file as the root set, only that part stored in scratch memory becomes part of the root set. This strategy helps lower the amount of useless storage retained across garbage collections since it excludes any dead pointers that have not been cleared out of the register file. It agrees well with the findings of Appel and Shao [5] on compilation techniques that are "safe for space complexity."

The instruction below allocates a block of memory whose size is in `r2` and places a pointer to the block in `r1`. 

```
allocate
allocate_RTS( R2 );
```
allocate($r_1, r_2$)

{ pHptr mem;
  const Int64 allocsize = asInt( R2 );
  reserve(mem, allocsize);
  if (mem == NULL)
    mem = reserveFail( localState, allocsize );
  obPtr( R1, mem );
}

The reserve function is the main interface to the memory manager in the run-time system. Normally, a call to reserve returns a pointer a memory block of the requested size. However, when no memory is available, the call returns NULL. In this case, the code calls reserveFail to yield control to the memory manager. The code for reserveFail synchronizes all SVPs, performs a garbage collection, and returns the requested memory blocks to its callers. The invariant for reserveFail is that it will never return NULL. If memory is not available after a garbage collection, then the program has exceeded its maximum heap limits, and it will stop with an error. We provide more details about the implementation of the memory manager in Section 7.7.

7.6.6 Simple Loads and Stores

The load and store instructions are straightforward to implement because they do not pay attention to the presence field of a memory word, i.e., they are not used for synchronizing different threads. The following is a typical load instruction:

\[
\text{load}(r_1, rF[5]) \\
R_1 = \text{asPtr}(RF)[5];
\]

A store instruction is compiled as follows:

\[
\text{store}(rF[5], r1) \\
\text{asPtr}(RF)[5] = R_1;
\]

The store instruction is supposed to set the presence field of the memory cell to full. Given the encoding of tags in our data representation, this operation is done by the code above if the value in $R_1$ is a valid integer, float, or pointer tagged with normal. These values implicitly make the state of the memory cell full. The code generator guarantees that values stored with the store instruction fall in one of these categories.
7.6.7 Synchronizing Memory: touch, istore, and take

The touch instruction is the basic synchronizing instruction in the SMT instruction set. Because touch is suspensive, it might include a live-state annotation as in the following example:

\[
\text{touch}(r1) \{r2,r3\}
\]

\[
\{\text{const pHptr p = asPtr( R1 );} \\
\text{if ( !isFull( *p ) )} \\
\{ \text{obPtrNoTag(asPtr(RT)[2], &IL_27);} \\
\text{asPtr(RT)[3] = R2;} \\
\text{asPtr(RT)[4] = R3;} \\
R1 = p; \\
\text{goto *((Void*) suspend.RTS);}
\}
\]

The code above computes the address of the memory cell it going to touch and then tests its contents with the isFull predicate. The isFull predicate returns true if the given value does not have a tag of empty. In our encoding scheme, this means that the bit-pattern of the value when interpreted as a signed integer, must be less than 0x7FF8000000000000. If the test is true, then the predicate of the conditional is false, and execution continues.

If the test is false and the predicate is true, however, then the thread must suspend because the value of the memory cell is not available. Before suspending, the thread saves its resume address (marked by the label IL_27) and all live state into its task structure in register RT. The offsets within the task structure are one greater than those shown in Section 4.4 due to a header word reserved by the garbage collector. Notice that instead of using a register mask to identify the saved registers, we explicitly encode them into the generated code. After saving state, the thread jumps directly to suspend.RTS in the run-time system. The code for suspend.RTS adds the task structure to the deferred list of the memory cell and schedules a new thread from the local work queue. The code at suspend.RTS also takes care of preserving the RF and RB registers of the current thread.

When a value becomes available, the suspended code will resume at IL_27. By convention, when the code resumes, the address of the task structure is in RT and the values
of RF and RB have been restored. The code makes certain to restore any additional live state before proceeding. Notice that after the resumption, the code does not use any of local variables. This approach avoids problems that might occur if the code resumes on a different processor, with a different stack configuration, than the one on which it suspended.

The code for touch is substantial, but it is important to remember that the common case is that a value will be available when the instruction executes. In this case, the cost of the touch is essentially the fast isFull test and a conditional jump that is easy to predict for modern branch prediction hardware. Section 8.4 gives empirical evidence that indeed most touch instructions do not suspend.

The istore instruction is used to synchronize with computations waiting for values. Like store, the istore sets the state of a memory cell implicitly by storing a valid integer, float, or normal pointer. Unlike store, it also takes care of resuming any computations waiting on the value. The tricky part of an istore is that the code must access the memory cell atomically since the istore instruction might be running concurrently with any number of touch instructions.

```plaintext
isetore(r1,r2)
{
    const Int64* storeAddr = (Int64*) asPtr( R1 );
    const Int64 swap = asInt( R2 );
    Int64 old = *storeAddr;
    Int64 compare;
    do { compare = old;
        old = compareAndSwap64(storeAddr, compare, swap);
    } while ( compare != old );
    if ( !isEmptyVal((pHob) old) )
        IMStore.Complex_RTS(storeAddr, old, swap, 0);
}
```

The core of the istore code is the compareAndSwap64 routine. This routine is provided by the machine-dependent portion of the run-time system, and it is designed to be inlined into its callers. The routine behaves as follows: if the contents of the memory cell at address storeAddr equal the value of compare, then the value swap is stored in the memory cell. The operation is atomic because it guarantees that no other processor can access the memory cell between the comparison with compare and the store of swap. If the contents of storeAddr are not equal to compare, then the swap does not occur. In both cases, the contents of memory at storeAddr are returned as the result of the function call. This allows the caller
to determine whether or not the swap occurred by comparing the return value with compare.

After the compareAndSwap64 succeeds, the code for istore checks the returned value with the isEmptyVal predicate. This predicate returns true only if its argument is emptyval (0x7FFFEO00000000000). If the predicate is true, it is certain that no deferred list existed in the cell, so execution can proceed to the next SMT instruction. If the predicate is false, then a deferred list has to be processed. This case is not the common case, so we leave it up to the run-time system routine IMStore_Complex.RTS to handle it. This routine takes care of splicing the deferred list onto the local work queue.

The most complex of the synchronizing memory instructions is take. A take of an M-Structure cell with a value causes the value to be fetched and the location to be emptied out, so that any future take defers. A take of an empty cell (or of a cell with a deferred list) suspends and waits on the deferred list. An istore to an M-Structure cell behaves as it does for an I-Structure cell, so it is the special behavior of take that assures the mutual-exclusion semantics of M-Structures. Here is an example of a take:

```c
.take(r1,r2) {r2}
```

```c
{  
    IL.31:  
    obPtrNoTag(TR1, R2); /* Test presence state */
    if ( !isFull( *asPtr(TR1) ) )  
        { obPtrNoTag( asPtr(RT)[2], &&IL.32 );  
            asPtr(RT)[3] = R2; /* Save live state. */
            /* suspend.RTS expects: R1 = suspend addr, RT = task structure */
            R1 = TR1;
            goto *((Void*) suspend.RTS);  
        } 
    IL.32:  
        R2 = asPtr(RT)[3]; /* Restore live state using RT. */
        /* Reset the Mcell pointer. */
        obPtrNoTag(TR1, R2);  
    /* Is cell EMPTY due to another thread? */
    TR2 = *asPtr(TR1);
    if ( !isFull(TR2) ) goto IL.31; /* Try again */
    TR1 = (pHob) compareAndSwap64((Int64*) asPtr(TR1), asInt(TR2), EMPTYVAL);
    if ( asInt(TR1) != asInt(TR2) ) goto IL.31;
    /* If we reach this point, the swap has succeeded */
    R1 = TR1;  
}
```
The code first stores away the pointer to the M-Structure cell in R2 into a temporary register TR1. Temporary registers are not visible to the SMT instruction set, but they are useful in generating code for SMT instructions. Next, the code tests the presence state of the M-Cell using the isFull predicate. If the cell is not full, then the take must suspend until a value is available. As in the case of touch, the resume address and the live state are saved in the task structure (pointer in RT) and then the code jumps to suspend.RTS.

Control returns to the code when an istore is issued against the M-Cell. The code restores the live state and checks the status of the cell again. Why? The istore may have caused multiple deferred take instructions to resume, and one of them may have already grabbed the contents of the cell. If so, then this take must suspend itself again until the next istore. However, if the cell is not empty, the code attempts to grab the value atomically using the compareAndSwap64 routine. The swap value is the emptyval (the C constant EMPTYVAL). If the swap fails, then some other instruction has beaten this take to the contents of the cell. The code tries to grab the contents again from the beginning (label IL_31). But if the swap succeeds, this take has been successful. The results of the swap (the old value of the cell) are placed in the destination register R1 and the program continues.

The code for take is complex because it comprises both a touch and a load. Moreover, the presence state of an M-Structure cell might be changing all the time from empty to full and back. In contrast, a plain touch is much simpler to implement because the presence state of an I-Structure cell can only transition from empty (or deferred) to full. The code for take has not been the focus of much optimizations since M-Structure operations are fairly rare in pH code.

7.6.8 Barrier Instructions: initbar, incbar, decbar, and touchbar

The initbar instruction initializes the count and prev fields of a barrier structure. In properly structured code, this instruction is guaranteed to execute before any incbar, decbar, or touchbar on the barrier being initialized. Thus, there are no synchronization issues to consider. The instruction is compiled as follows:

```
initbar(r1, r2)

asPtr( R1)[1] = asInt( R2 ) + 1 ;
asPtr( R1)[2] = RB ;
RB = R1 ;
```
Again, due to the header word for the garbage collector, offsets into the barrier object are one greater than otherwise expected. Another minor implementation detail is that the counter is initialized to one greater than the initialization count in the instruction. Correspondingly, the touchbar instruction (explained later) also decrements the counter when it executes. This change allows us to synchronize the last decbar with the touchbar using only the counter. The abstract behavior of the instruction in the SMT machine remains unchanged.

The incbar instruction increments the count field atomically. It uses the atomicInc64 primitive provided by the run-time system. Here is an example of the code generated for incbar:

\[
\text{incbar(rB)}
\]
\[
\text{atomicInc64( (Int64*) (asPtr( RB ) + 1), 1 );}
\]

The code for atomicInc64 is built on top of compareAndSwap64 as follows:

\[
\text{extern inline Int64 atomicInc64(volatile Int64* incAddr, Int64 incVal)
}\]
\[
\{ 
\text{Int64 oldVal;}
\text{Int64 newVal = *incAddr;}
\text{do }
\{ 
\text{oldVal = newVal;}
\text{newVal = oldVal + incVal;}
\text{newVal = compareAndSwap64(incAddr, oldVal, newVal);} 
\text{while (oldVal != newVal);} 
\text{return oldVal;}
\text{}
\}
\]

The decbar instruction decrements the count field. Additionally, a decbar will enable the post-region if it notices it is the last instruction to execute against the barrier. The implementation of decbar relies on the run-time system primitive atomicDec64 to decrement values atomically.

\[
\text{decbar(rB)}
\]
\[
\{ \text{const Int64 oldCount = atomicDec64( (Int64*) (asPtr( RB ) + 1), 1);} 
\text{if ( oldCount == 1 )}
\text{ */}
\text{spawn.RTS( *(asPtr( RB ) + 3 ));}
\}
\]

Remember that the subtle issue in the implementation of decbar is determining whether or not the last decbar executes before or after the single touchbar for a given barrier. In
our implementation, we have solved this problem by initializing the count to a value one greater than that required and by making *touchbar* to decrement the count. Thus, if the count is ever 1 when a *decbar* instruction executes, then it is guaranteed that the *touchbar* has executed and that the task structure for the post-region is available in the third field of the barrier structure. In this case, the code for *decbar* makes sure to spawn the post-region before continuing.

One of the effects of the partitioning analysis and the threading transformation is to decrease the number of *incbar* and *decbar* instructions emitted versus the naive compilation algorithm. This reduction is important in improving the performance of the code since both these instructions require atomic access to memory.

The last instruction for handling barriers is *touchbar*. In properly structured code, a *touchbar* is executed only once per barrier. If the counter has reached 1 when the *touchbar* accesses the barrier, then it is certain the last *decbar* has executed and the post-region is free to proceed. Otherwise, the post-region must suspend until the last *decbar* executes.

```c
touchbar(r1)

if ( asInt( (asPtr(R1))[1] ) == 1LL )
    RB = (asPtr( R1 ))[2];
else
{
    Int64 oldCounter;
    /* Setup resume point */
    obPtrNoTag( asPtr(RT)[3], &&IL.54 );
    /* Set post-region in barrier */
    (asPtr(R1))[3] = RT;
    oldCounter = atomicDec64( (Int64*) (asPtr( R1 ) + 1), 1LL);
    assert( oldCounter >= 1 );
    /* Check counter again, in case last 'decbar' has executed */
    if (oldCounter == 1)
        /* Restore previous barrier */
        RB = (asPtr( R1 ))[2];
    else goto *((Void*) schedRTS);
    IL.54:
}
```

The tricky part of *touchbar* is that the code must be very careful that the last *decbar* does not "sneak by" while the *touchbar* is getting ready to suspend. This explains two features of the implementation above. First, the counter is decremented after the task structure for the post-region is written into the barrier. If done in the reverse order, the last *decbar* might
see a counter value of 1, but garbage in the post-region slot. Second, the code checks the value of the counter again after decrementing just in case a decbar has executed between the time the of the first check and the time of the decrement.

If the counter has reached 1, whether before or after the touchbar decrements it, the previous barrier is restored to the barrier register RB and execution continues in the post-region. If the counter has not reached 1, then there are outstanding threads, and the code suspends by writing itself into the barrier and by jumping directly to sched_RTS in the run-time system.

The implementation of the touchbar instruction is obviously complex since it is synchronizing multiple threads. Fortunately, touchbar instructions occur only in code that uses the seq construct and nowhere else. Sequential code is fairly rare in pH, and in situations that call for the use of barriers, the programmer should be prepared to pay extra cost for such extensive synchronization.

7.6.9 Summary

In this section we have shown how SMT instructions are converted into C code with GNU extensions. While some instructions compile into simple statements, others like touch and istore require significantly more code. These instructions are fairly common, and they tend to make the C programs generated by our compiler rather large. However, we believe the impact on run-time is less marked. We have structured the code for instructions like touch so that the common case runs quickly, and indeed, the bulk of the code is devoted to handling the uncommon case, i.e., suspension.

The dynamic performance of programs is affected by several choices in our code generation strategy. First, the use of the GNU extensions for getting the address of a label (&&) and for computed-gotos—while extremely useful for establishing thread contexts—tends to trip up the optimizer because it increases the number of branch targets in a procedure. Second, we have chosen to use run-time system functions for common code, such as frame initialization. For debugging and instrumentation, it is preferable to have a centralized routine for common code, but for performance, it might be better to inline this code everywhere. Of course, this inlining comes at the cost of much larger programs.
7.7 Garbage Collection

Because garbage collection tends to affect many different parts of a compiler and run-time system, we approached the problem with great apprehension, as have other pH implementors before us. Though the design of the language effectively dictates automatic memory management, only one earlier system, pHluid, actually supported it. Previous implementations at MIT and Berkeley made do with ugly, explicit deallocation annotations (@free). But once the structure of the data representation was settled, developing the garbage collector was actually a very pleasant experience. It is important to note that most of the garbage collector is the work of Jan-Willem Maessen.

The collector is based on the standard two-space, copying design but includes a few modifications. The to and from spaces are not mapped as linear spaces in memory, but rather are divided into chunks of several virtual memory pages. Initially, an SVP is assigned its own chunk when it is initialized. The SVP allocates from this chunk and requests a new one only when it exhausts the current one. A round of garbage collection occurs when no free chunks remain. In this way the available heap is given to the SVPs that are allocating most quickly. Memory can be allocated in parallel, as each SVP manages its own allocation chunk. However, all SVPs must synchronize (via reserveFail) to perform the garbage collection which is currently implemented as a sequential algorithm. The processor that ran out of memory first is assigned the task of tracing the heap and restarting the others when it is done. We considered building a fully parallel collector, but concluded that the complexity of the effort would gravely affect the project schedule.

The memory allocation and garbage collection algorithms account for the remainder of the fields in the perThreadInfo data structure of Figure 7-1. The chunkBot, chunkHP, toSpace, fromSpace, and toUsed fields are used to keep track of each the chunk assigned to each SVP. The fields done by me, threadNum, and thread data are used when all SVPs synchronize before a garbage collection. They are also used to keep track of which SVP is responsible for doing the garbage collection and restarting the others.

The garbage collector defines four additional pointer tags beyond empty and normal. First, the reference tag is used to mark pointers that serve as references to values. When the GC traverses the heap, it uses this tag to determine that references point to single-cell memory blocks. Second, the malloc tag is used for pointers to objects that are not
movable. The memory allocator maintains an area of memory that it manages using the operating system malloc and free routines. Malloc'ed objects include arrays and other large objects. Third, the global tag is used to mark top-level objects such as closures for top-level functions, i.e., objects that are specified statically at compile time. These are handled specially because their space is allocated by the operating system (more properly, the program loader) not by the pH memory manager. The global tag can be combined with any other tag (e.g., to form global-reference). Finally, the forwarding tag is used only during garbage collection, not during normal program execution. It is used to mark objects that have already been copied to fresh memory.

The code generated by the compiler as well as the code in the RTS must be designed to operate within a garbage-collected environment. For example, the code must save all live register state to scratch memory whenever a call to reserve fails. The garbage collection algorithm uses this state as part of its root set for tracing memory. Without a proper root set, the garbage collector might not deallocate garbage, or even worse, might discard live data.

Another example occurs in the work-stealing loop in Figure 7-2. The code as shown in the figure is susceptible to a subtle livelock condition. Suppose one SVP tries to steal work but no work is available because all other SVPs are waiting for a garbage collection. The work-stealing SVP requires no heap allocation, so it will never call reserveFail to join with the others in the garbage collection. Instead, it will try to steal work from the others forever. To solve the problem, we check the global GCFlag before every call to yield, as follows:

```c
if (GCFlag) {
    pHptr garbage;
    /* Clear out state sitting in our registers to avoid leaks */
    obInt(RF,OLL); obInt(RV,OLL);
    obInt(RB,OLL); obInt(RSS,OLL);
    garbage = reserveFail(regs, 0); /* Check only! */
}  
else yield();
```
/* Machine Instructions: Sparc V9 64-bit loads and stores */
#define STX(r,a)
 { _asm volatile ("stx %0,[%1]": "r" (r), "r" (a): "memory"); }
#define LDX(a,r) { __asm__("ldx [%1],%0": "r" (r): "r" (a)); }

/* Sparc V9 64-bit compare and swap. */
#define CASX(a,c,s)
 { __asm__("casx [%2],%3,%0": "r" (s): "0" (s), "r" (a), "r" (c): "memory"); }

extern inline Int64 compareAndSwap64(volatile Int64* swapAddr,
Int64 compareVal,
Int64 swapVal)
{
    /* Declare temporary, caller-save 64-bit global registers. Only the
     * global (gxx) and out (oxx) registers are saved as 64-bit quantities
     * in Solaris 2.6.
     */
    register Int32 compareTemp _asm__ ("g1");
    register Int32 swapTemp _asm__ ("g2");

    LDX(&compareVal, compareTemp);
    LDX(&swapVal, swapTemp);
    CASX(swapAddr, compareTemp, swapTemp);

    /* We store the swapTemp value back into a UInt64 variable to
     * make sure we can pass it back properly. UInt64 values are not
     * stored in a single register on 32-bit Solaris.
     */
    STX(swapTemp, &swapVal);

    return swapVal;
}

Figure 7-9: UltraSPARC Definition of compareAndSwap64

7.8 Machine Dependences and Porting

Though we have focused on Sun UltraSPARC SMPs during the development of the pH
code generator, we spent a substantial portion of our time making sure that the code was
portable to other systems. Indeed, one of our major design decisions, that of using GNU
C as our output format, was due in large part to portability concerns. In this section, we
describe the machine-dependent definitions in the run-time system and provide some of the
UltraSPARC versions as examples.
All machine dependences in the run-time system are segregated into a single file in the run-time system. This file provides two sets of definitions to the rest of the code. The first is a set of basic type definitions including signed and unsigned versions of characters (Char), 8-bit integers (Int8), 16-bit integers (Int16), 32-bit integers (Int32), 64-bit integers (Int64), IEEE single precision floating-point numbers (Float), IEEE doubles (Double), void (Void), and booleans (Bool). The C standard does not provide for stable type definitions, so portable code must define its own types. The second set of definitions concerns basic atomic operations. These include 32 and 64-bit versions of compare-and-swap, `compareAndSwap[32,64]`, as well as 32- and 64-bit versions of atomic increment and decrement, `atomicInc[32,64]` and `atomicDec[32,64]`.

In the UltraSPARC implementation, these last four routines are built on top of the 32- and 64-bit compare-and-swap primitive as we discussed in Section 7.6.8. The definition of `compareAndSwap64` is given in Figure 7-9. The implementation of this primitive on the UltraSPARC is slightly complicated because it makes use of the 64-bit processor registers. Unfortunately, only very recently has a fully 64-bit version of Solaris become available, and GNU compiler support for it is not ready. The result is that in the 32-bit Solaris 2.6 used for developing the compiler only a few of the 64-bit processor registers are preserved across context switches. These include the “global” registers (g1-g7) and the “out” registers (00-07). In `compareAndSwap64`, we use two of the global registers as operands for the native UltraSPARC 64-bit compare-and-swap instruction `casx`.

The `compareAndSwap64` is the most important machine-dependent routine in our compiler. It requires that the underlying processor support some sort of 64-bit atomic memory operation. Today, processors from Sun, Intel, MIPS/SGI, Digital, and IBM provide this support. Beyond the definitions of the machine-dependent types and routines, there are no other changes required to port the system. Once the RTS has been ported, code emitted by the compiler can be recompiled by gcc for the new architecture.

7.9 Summary

This chapter has presented the strategy used in the pH compiler to generate executable code. We divided the problem into two parts. The first part concerned the implementation of the SMT machine abstractions on a real machine. These abstraction include the proces-
sors, the scheduling queue, and the synchronizing memory of the machine. Their concrete implementation comprises the bulk of the run-time system of the \( pH \) compiler.

Each SMT processor is implemented using a kernel-scheduled POSIX thread and a small C routine that schedules an initial thread. After that, the SMT code is self-scheduling. All the SMT virtual processors (SVPs) execute within the same operating system process, so they share a single addressing space. The global work-queue of the SMT machine is implemented using a set of distributed work queues, one per SVP, to take advantage of locality. A processor spawns new threads onto its local queue and schedules threads out of its local queue. When the local queue is empty, the processor attempts to steal work from another randomly chosen processor. This type of scheduling has exhibited excellent properties in systems like Cilk, and though these performance results are not generally applicable to a non-strict language like \( pH \), they will tend to hold in the common case of \( pH \) program execution. To implement the SMT synchronizing memory, we designed a novel representation for \( pH \) data. This encoding is based on IEEE floating-point Not-a-Numbers and provides a seven-bit tag field for encoding presence information or GC types directly in each \( pH \) scalar. The range of integer scalars is reduced about 0.02% while that of floating point numbers is unchanged.

The second part of the problem concerned code generation for SMT instructions. We chose to emit C code, rather than byte-code or assembly language, for two main reasons. By emitting C, we can achieve good performance without having to write many optimizations (like register allocation and instruction scheduling) since these already part of the C compiler. Furthermore, a C back-end is much easier to port from one architecture to another. By taking advantage of extensions in the GNU C compiler, we have been able to generate lightweight threads in a way that is fairly generic across gcc versions. We showed how to generate C statements for each SMT instruction. Some instructions compiled to a single statement, while others, especially those involving synchronization, require substantially more code. In all cases, however, there is a single "fast-path" in the common case through the code. This path is the one we expect will be executed most of the time.
Chapter 8

Preliminary Evaluation

This chapter presents a preliminary evaluation of the code-generator we have designed and implemented. Throughout this dissertation, we focused on correctness and completeness first and on performance second. The experimental evidence bears this out. The principal goals of this evaluation effort were to develop the tools to measure the behavior of the code and to help us understand where performance might be improved. A detailed performance study is left for future work.

8.1 Experimental Setup

8.1.1 Programs

We selected five programs for the evaluation. The programs were chosen because they demonstrate a variety of programming strategies and because they are known to be correct. To lessen the effect of startup transients in paging and caching, the input parameter to each program was tuned to yield a single-processor execution time of about 30 seconds. It is also important to note that bounds-checking for arrays was disabled. Finally, for each program, the same executable was run on all machine configurations. We did not compile special “uniprocessor” versions. The programs we chose are:

**Fib:** a doubly-recursive version of the Fibonacci function. This program consists of only 15 lines of pH code, but it is a good test of scalability and procedure calling conventions. Fib was executed with the argument 32.
Queens: determines the number of ways in which $n$ queens can be positioned safely on a chess board of size $n \times n$. Like Fib, Queens is a simple program (only 27 lines of source code), but it is a good test of procedure calls and lists. Queens was executed with an argument of 9, so it computed the results for values of $n$ from 1 to 9.

Paraffins: a well-known benchmark in functional languages, Paraffins enumerates the distinct isomers of each paraffin of size up to $n$.\footnote{A paraffin is a molecule with the chemical formula $C_nH_{2n+1}$, where $n > 0$. Isomers are compounds with identical atomic compositions but differing geometric structures.} The number of paraffins grows exponentially with $n$, and in our tests, the parameter $n$ was set to 20. Paraffins consists of 185 lines of $pH$ code.

MMT: a blocked version of $n \times n$ matrix multiply. The elements of the matrix are 64-bit integers. The innermost loop of the matrix multiply algorithm reads 4 elements from each input matrix and computes 16 partial results from these 8 elements. This is a well-known strategy to reduce the number of memory accesses in matrix multiply. For our tests, the value of $n$ was set to 150, and the input matrices were generated at run-time. MMT consists of 131 lines of $pH$ code.

Gamteb: a Monte Carlo simulation of the trajectories of photons passing through a carbon rod divided into two cells. Gamteb contains an enormous amount of parallelism since each particle can be simulated independently. For our purposes, we simulated 40,000 particles. Gamteb consists of 691 lines of $pH$ code.

8.1.2 Machine

The benchmark programs were executed on a Sun Enterprise 5000 multiprocessor with eight UltraSPARC 168MHz processors and 512MB of RAM. The operating system was Solaris 2.6. Though the machine was not running in “single-user” mode, the load apart from the $pH$ benchmarks was negligible.

8.1.3 Instrumentation

Both the code produced by the compiler and the RTS were instrumented to gather information about the benchmark programs. The compiler emits code that keeps counts of
the SMT instructions executed. In addition, the code tracks whether or not suspensive instructions like touch actually suspend. For good performance, the statistics are gathered on a per virtual-processor basis and are aggregated at the end of execution. If global statistics counters were used instead, the code would need to update them with slow atomic operations.

The RTS is instrumented to keep track of the frequency and duration of the calls to major routines such as garbage collection and frame allocation/setup. The timing routines take advantage of the high-resolution timer interface, gethrtime, under Solaris. Program execution times were gathered using the timex system command, and we recorded the “real” time in our measurements.

It is always important to keep in mind the “probe effect” when gathering performance data of parallel programs: instrumentation code affects the behavior programs, sometimes quite substantially. To limit this effect, we tried to insert as little instrumentation code as possible, especially timing code. In the following sections, we discuss how these concerns affected our measurement techniques.

### 8.2 Components of Execution Time

In our first investigation, we wanted to get an idea of where programs were spending time during execution. We divided the run time of a program into Compute time, where the program is doing useful work to produce a result, and RTS time. Within the RTS time, we further sub-divided the time into one of three categories: garbage collection, scheduling, or frame initialization.

Figure 8-1 consists of a graph with five bars, one for each benchmark, showing the percentage of execution time dedicated to each of these four activities. The data correspond to single-processor executions of each of the program. The total run time is gathered from uninstrumented executions of the program, while the RTS component times are gathered from instrumented runs. We combine the timings of instrumented and uninstrumented runs because we did not want to add the overhead of timing the RTS sections to the total run time. The Compute time is the difference of the total run time and the sum of the time spent in the RTS.

We knew that our frame initialization strategy was not particularly efficient, but the
results shown in the graph were still surprising: frame initialization is a significant overhead in all benchmarks. The behavior of Fib is not of particular concern since Fib performs a tremendous number of function calls. But the results for the other benchmarks point out that we need to re-examine our strategy for loops. In the current code generator, loops are transformed by the front-end of the compiler into tail-recursive function calls. Tail-recursion in our compiler simply means that a callee does not necessarily have to return to its parent. Instead, the caller information passed to the callee can be that of some ancestor of its parent. This scheme allows the frames of tail-recursive procedures to be garbage-collected efficiently.

It also means that we do not reuse frame memory between garbage-collections, so each loop iteration must allocate and initialize its own activation frame. Without independent activation frames, loop iterations cannot execute in parallel. Tail-recursion, however, may be more general than necessary for loops. In particular, loop “functions” are never curried, so they can always expect to be called as a result of a full application. Also, loop parameters are often constants, so loops can expect a portion of their arguments to be evaluated. These two facts may allow us to design a faster, loop-only frame initialization strategy. Eventually,
we may also want to reuse already allocated loop frames directly, bypassing the garbage collector, but this must be done in a way that does not affect non-strictness or the semantics of barriers.

There is some good news in Figure 8-1. Excluding Paraffins, it appears that garbage collection accounts for less than 5% of the total run-time in four of the applications. This seems a very reasonable overhead for fully automatic heap management. Of course, as we decrease the overhead of frame initialization, the relative cost of garbage collection might increase. The behavior of Paraffins, on the other hand, is disturbing. The problem with Paraffins seems to be that its peak working set is huge, often 50MB and sometimes close to 90MB. Traversing and collecting a working set of this size is expensive, and it has a first-order effect on performance.

The Paraffins working set consists largely of activation frames, not user data structures. The problem is that several “loop” nests generated from list comprehensions (not from loop constructs) are structured so that iterations are not called in a tail-recursive manner. This means that at the end of each of these loop nests the number of live activation frames is equal to the sum of the iteration counts at each loop level. With an input of 20 to Paraffins, this could result in more than 100,000 live activation frames at some points in the execution. It is interesting to note that the version of the code compiled by the Id compiler for Monsoon does not suffer from this condition. Id list comprehensions are implemented using “open-lists” and explicit side-effects. This allows their construction to be expressed as efficient tail-recursive code.

Though disappointing on the whole, this experiment did yield important results. It shows that before considering other optimizations, like parallelizing the garbage collector or improving the work-stealing algorithm, we must reduce the cost of frame initialization. In addition, the problems with storage management in Paraffins pointed out a deficiency in the compilation of list comprehensions. The solutions to these problems will certainly involve changes to both the compiler and the run-time system.

8.3 **Instruction Mix**

After determining the major components of execution time, we wanted to better understand the Compute portion of a program by examining the dynamic SMT instruction mix. In
Figure 8-2: SMT Instruction Mix

Figure 8-2, we show the percentage of instructions executed in eight different categories: ALU (arithmetic and register-to-register moves), Touch, Spawn/Schedule, Control Flow (jump and switch), Memory Access (synchronized and unsynchronized), Memory Allocate, Barrier (initbar, incbar, etc.), and Frame Initialization (RTS primitive). The data shown pertains to single-processor executions. The instruction mix for any particular application is essentially identical across any number of processors.

In most of the benchmarks, the portion of the instruction mix due to memory accesses seems high. A principal cause appears to be poor threading, and it affects the code in two ways. First, if a value is used in several different threads, it must be reloaded from the frame in each thread. This situation is commonplace in the code to compile pattern matching of data structures. The data structure is broken down into components in one thread, and the components are used in other threads. In the interim, the components must be stored in the frame since registers cannot be used to transmit values from thread to thread. Second, when a value is used in several threads, it is is usually assigned a reference and a location. To access the value, the code must use traverse the extra indirection of the reference.

In contrast, the instruction mix data shows that touch instructions account for less than
7% of the instructions executed in the benchmarks. This number does not represent an important overhead, and it also decreases as partitioning improves. In fact, as we discuss in the next section, the cost of touch instructions in most programs is very small since the majority of these do not cause suspensions.

It is interesting to compare the memory management data for Paraffins from the previous section and this section. In the previous section, we saw that garbage collection accounted for roughly a third of the execution time, yet in the instruction mix, Paraffins does not seem to execute an outrageous number of allocation instructions. A possible explanation for this behavior is that Paraffins is allocating large, long-lived data structures. This led us to examine the behavior of Paraffins more closely and to discover that the culprits are in fact activation frames.

Of course, not all SMT instructions are created equal. The machine-code implementations of some instructions are significantly more complex than others. Consequently, it would also be useful to determine the machine-code instruction mix for a program. We decided to leave this task for the future since it would seriously complicate our otherwise portable implementation. In any case, efficient SMT code is a prerequisite if machine-code optimizations are to be of any use.

8.4 Touch Behavior

An important assumption in our compilation strategy is that touch instructions are cheap because most programs suspend rarely. We wanted to determine whether this assumption was actually correct, so we inserted instrumentation code to record the outcome of each touch instruction. Figure 8-3 shows the percentage of touch instructions that actually encountered an empty data slot and caused the thread to suspend.

In three of the five benchmarks, the suspension rates are essentially negligible. This result agrees well with our notion that most pH programs can execute in a fairly strict fashion. In a program without data structures like Fib, we can even execute without any suspensions. Some other interesting patterns emerge from this data. As the number of processors increases, the number of touch instructions that suspend also tends to increase. There is a simple explanation for this phenomenon. In most programs, threads that consume values run after the threads that produce values, but as more processors are made available,
the probability that consumers execute concurrently with, or even ahead of, producers increases, and concomitantly, so does the chance of suspension.

It is useful to compare these results with those of Schauser and Goldstein [57]. In their study, they were able to run Fib, Paraffins, and MMT without any suspensions. How? They inserted barriers into the programs to guarantee a static schedule. In contrast, the code generated by pHc still exhibits some suspensions because the scheduling guarantees of partitioning are much weaker than those of barriers. The two programs that exhibit significant rates of suspension, Queens and Gamteb, are two of the few that Schauser and Goldstein found to require dynamic scheduling, i.e., they are fundamentally non-strict. Thus, it is not surprising that our simple "most serial" scheduling strategy leads to lots of suspensions. The proper schedule for these programs is data dependent and complex.

The analysis of the behavior of touch instructions has affirmed a basic assumption of our compiling strategy: that programs rarely need to suspend. We saw earlier that touch instructions are not a significant portion of the instruction mix, and now we know that most of these can execute very cheaply. We conclude, then, that optimizing touch should not be a priority in the current compiler.

8.5 Speedup

In this section we present the results of running our benchmark programs on different numbers of processors. Remember that for any particular program, the executable and the input parameters are identical across all processors. To use different numbers of processors, we simply change a command-line switch that tells the pH RTS how many virtual processors (SVPs) to create. Threads are scheduled onto the SVPs by the work-stealing scheduler, but
we leave it up to the operating system to schedule SVPs onto real CPUs.

Figures 8-4, 8-5, and 8-6 show the speedup curves generated from our measurements. The speedup factor for running on \( n \) processors is calculated using the formula:

\[
\text{Speedup}_n = \frac{\text{Time on 1 Processor}}{\text{Time on } n \text{ Processors}}
\]

The elapsed times used to compute the speedup factors correspond to the execution of *uninstrumented* programs. We did not want to distort the figures by including the overhead of statistics collection.

The speedup results are mixed. On the positive side, Fib, MMT, and Gamteb show between 2.7 and 3 speedup on four processors. Queens comes close with a 2.4 speedup on four processors. In addition, most codes continue to improve at least up to six processors. But beyond four processors, the rate of improvement begins to degrade significantly, and some programs actually slow down. MMT, for instance, shows only 2.7 speedup on eight processors.

The best speedup achieved was 5 on Fib running on eight processors. While promising, there should be enough work in Fib to achieve perfect linear speedup. We expect that better memory usage in function calls may improve the performance of Fib since it currently performs about 600 garbage-collections during its execution.

It is most disappointing, however, that speedups are fairly meager. The algorithms in these benchmarks are known to contain lots of parallelism, but our current compiler seems to be unable to exploit it beyond four processors. Paraffins is particularly galling since it does not speed up at all. We believe this problem is directly related to the issues outlined earlier with list comprehensions and a huge working set.

### 8.6 Single-Processor Performance vs. Haskell

The lenient evaluation strategy of \( pH \) should yield more efficient code than the full-blown lazy evaluation of Haskell. As discussed in Section 2.5, the correct implementation of lazy evaluation places a great bookkeeping burden on the implementation. On the other hand, Haskell compilers do not have to worry about parallelism while the \( pH \) compiler always generates programs that execute in parallel by default, without any uniprocessor optimizations. A uniprocessor-specific implementation of \( pH \), for instance, could do without
Figure 8-4: Speedup Curves for Fib and Queens
Figure 8-5: Speedup Curves for Paraffins and MMT
the expensive atomic operations required for synchronization.

To determine the effect of these two opposing forces on the performance of pH code, we decided to compile our standard pH benchmarks using two common Haskell compilers, the Glasgow Haskell Compiler (GHC 4.00) and the Chalmers Haskell Compiler (HBC 0.9999.4). The MMT benchmark could not be compiled in Haskell because it uses I-Structure side-effects directly, but the other four benchmarks required only minor changes (due to different type class structures for I/O) to compile under GHC and HBC. In Figure 8-7, we show the execution time in seconds for five versions of each program. The input parameters are as given in Section 8.1. The first column shows the pH single-processor execution time; the second and third columns show the optimized and unoptimized versions under GHC; and the last two columns show the optimized and unoptimized times for HBC.

Both Haskell compilers, even generating unoptimized code, trounce the single-processor performance of pHc. This is particularly apparent in Fib, where the two Haskell compilers properly recognize that the code is strict and optimize procedure calls dramatically. Nevertheless, it is important to understand that beyond generating parallel-ready code, pHc is also operating under another handicap: it is generating 64-bit code instead of 32-bit code.
Figure 8-7: Single-Processor Execution Times. The table shows the time in seconds for five versions of each benchmark: the first is the original pHc program while the other four are compiled with Haskell compilers.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>pHc Optimized</th>
<th>pHc Unoptimized</th>
<th>GHC 4.00 Optimized</th>
<th>GHC 4.00 Unoptimized</th>
<th>HBC 0.9999.4 Optimized</th>
<th>HBC 0.9999.4 Unoptimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fib</td>
<td>64.03</td>
<td>3.05</td>
<td>3.76</td>
<td>1.80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Queens</td>
<td>29.89</td>
<td>2.27</td>
<td>12.84</td>
<td>0.95</td>
<td>7.50</td>
<td>2.86</td>
</tr>
<tr>
<td>Paraffins</td>
<td>30.29</td>
<td>2.56</td>
<td>3.76</td>
<td>2.10</td>
<td>4.58</td>
<td>2.86</td>
</tr>
<tr>
<td>Gamteb</td>
<td>291.91</td>
<td>41.83</td>
<td>190.71</td>
<td>83.44</td>
<td>43.16</td>
<td></td>
</tr>
</tbody>
</table>

like the Haskell compilers. This can have a tremendous impact on performance, especially since our native code generator (gcc) tends to emit much less efficient code for 64-bit quantities. In one extreme, we observed that a simple integer matrix multiply program, written in C and compiled with gcc, ran six times slower when using 64-bit (long long) matrix elements rather than 32-bit (long) elements.

8.7 Summary

This preliminary evaluation of the pH code-generator has shown that the code is slow and often resource intensive. In looking at the major components of execution time, we discovered that programs spend an inordinate amount of time in frame initialization. Some of overhead comes from inefficiencies in our calling conventions, but the bulk seems to be due to the transformation of loops into regular (though tail-recursive) procedure calls. Loops are less general than procedure calls, and we should develop a special, simpler, and faster frame convention for them. We also discovered that, except for Paraffins, garbage collection was not a major component of execution time for most benchmarks. The huge working set of Paraffins seems to be the result of poor compilation of certain list comprehensions. This area of the compiler will certainly receive our attention in the future.

We also looked closely at the dynamic SMT instruction mix. We found that the most common instructions executed were Memory Access instructions. This is likely due to deficiencies in our partitioning and threading algorithms which often force the use of frame slots instead of registers for storing values. We also found that touch instructions account for a modest percentage of all instructions executed. Looking more closely at the behavior of
touch instructions, we found that in three of the five benchmarks (Fib, Paraffins, MMT) at least 98.8% of these executed quickly without suspending. The suspension rates in Queens and Gamteb were much higher (up to 16.2%), but these programs are known to be highly non-strict.

We measured the performance of each benchmark with the same input parameter in machine configurations ranging from one to eight processors. Overall, the speedup numbers we obtained were disappointing. The best speedup was a factor of 5 for Fib on eight processors. The worst case was Paraffins, showing a speedup of 1.3 on eight processors. We believe the cost of garbage collecting a huge working set is mostly to blame. Most benchmarks showed marginal speedups up to six processors, but beyond, we actually observed slowdowns in all except Fib. In fact, beyond four processors, speedups rates declined noticeably.

Finally, we compared the single-processor performance of pH code with optimized and unoptimized code from two Haskell compilers, GHC and HBC. We found that our code is much slower than that produced by the other two compilers. In theory, the lenient evaluation of pH should help us produce faster code, but in reality, pHc is hampered by the fact that it produces 64-bit, parallel programs rather than 32-bit, sequential ones.

The instrumentation framework we have implemented in the compiler and run-time system has given us good insight into the behavior of our code. Still, there are other measurements we may want to consider in the future. For instance, we did not study how the pH code consumes memory bandwidth. The Sun Enterprise server architecture provides excellent performance in this area, but it is possible that our current codes, which tend to access memory very frequently, might be saturating the system. This problem is not easy to detect with our current instrumentation. We expect to look into it eventually, but only after the efficiency of the compiled code has been improved somewhat.
Chapter 9

Related Work

Excellent research on parallel functional programming languages goes back at least 20 years. Hammond [23] provides a thorough overview of the field, so in this section, we concentrate on the work most closely related to the pHc compiler.

9.1 Id/pH Compilers

The pHc compiler owes much to three previous research efforts that have wrestled with the problem of generating code from Id/pH for non-dataflow parallel machines: Berkeley Id, pHluid, and Id-S.

9.1.1 Berkeley Id

Shortly following Traub's initial work on partitioning, Culler and his group at Berkeley began work on an Id compiler that utilized the front- and middle-ends from the MIT dataflow compiler but included a new code generator for parallel machines based on standard processors. The Berkeley team made a number of important contributions, the first of which was the development of the Threaded Abstract Machine (TAM) [16]. The TAM framework consisted of self-scheduling, non-suspensive threads; a hierarchical storage model consisting of registers, local frames, and heap-allocated data structures; explicit message-based communication between tasks; and a two-level scheduling model. The TAM framework was designed to compile Id on massively-parallel machines like the CM-5, and it owes much to its dataflow heritage. However, TAM can stand on its own as a general multithreaded compiling target for any language. For Id, it allowed development on the language to be
decoupled from the underlying execution vehicle.

The second important contribution made in the Berkeley Id compiler was the development of a series of powerful partitioning algorithms [56, 70, 55]. The best of these, in Schauser's dissertation, handles all the complex features of the language and generates code that is almost as fast as if programs were compiled strictly. In fact, Schauser notes that progress in other areas, such as I-Structure analysis, should be made to support more advances in partitioning.

The third contribution of the Berkeley work was a family of Id compilers for both workstations and parallel processors like the CM-5. Impressive speedups were reported over a wide range of applications, though absolute performance tended to lag behind more conventional languages like C or Fortran. These compilers eventually set the benchmark for Id execution since their performance on ever-faster standard processors surpassed even that of the custom-made, though aged, Monsoon.

There are four basic differences between the approach taken in Berkeley Id and pHc. First, pHc generates suspensive threads while TAM generates non-suspensive threads. One might think there is a similarity between a suspensive thread and a TAM quantum (an epoch in which all threads scheduled belong to the same procedure), but actually, suspensive threads are very different. They are statically defined, and they can include control-flow that spans procedures. Second, TAM generates code from structured dataflow graphs while pHc uses $\lambda^*$. Third, the scheduling mechanism in the SMT machine does not necessarily follow the TAM two-level scheme. We purposely left the scheduling mechanism under-specified to allow more flexibility in the implementation. Finally, pHc was designed strictly for SMPs, and as such, it assumes a much lower-latency memory interface than TAM. This assumption tends to influence the system design substantially, especially in the implementation of synchronization.

9.1.2 pHluid

The pHluid system [18] is an Id compiler and run-time system developed at DEC CRL and is an outgrowth of Nikhil's work on P-RISC [43]. The compiler emits parallel code suitable for loosely-coupled distributed memory machines, based on the assumption that the most widespread and economical parallel computers in the near future will be clusters of workstations. Internally, pHluid translates Id source programs into P-RISC graphs. The
nodes of these graphs are not single operations, as in dataflow, but rather short sequences of instructions for a RISC-like multithreaded abstract machine. Optimizations and thread partitioning are performed on the P-RISC intermediate code, and the compiler generates C code that, like pHc, leverages on the specialized features of the GNU compiler.

The pHluid system focuses on two crucial issues essential to good performance in workstation clusters: good load balancing/management and effective use of the memory hierarchy. The run-time system implements a work-stealing scheduler with clever extensions for prioritized threads. The heap allocator incorporates many tricks to keep the cost of I-structure accesses as low as possible, includes a cheap but effective heap caching scheme that takes advantage of the monotonic behavior of I-structure presence states, and implements a full distributed garbage collection algorithm. Early performance measurements indicate that pHluid generates decent uniprocessor code as well as good speedups on small parallel benchmarks like nqueens.

Berkeley Id and pHluid exhibit many similarities, such as the use of non-suspensive threads and a graphical intermediate representation, but pHluid is targeted at parallel machines where communication latencies are enormous compared to processor cycle times. Thus, it places great emphasis on the heap caching scheme which is absent from TAM. Also, scheduling in pHluid, as in pHc, does not follow the two-level structure of TAM.

9.1.3 Id-S

Andy Shaw [59] analyzed the performance of Id and concluded that the overheads incurred precluded an efficient implementation of the full multithreaded language on small SMPs. His solution to the performance problem was radical: he defined Id-S, a new dialect of the language. Id-S is a sequential language, so it is supposed to execute in “top-to-bottom, left-to-right” order. In addition, it is strict, first-order, and assumes, but does not enforce, I-Structure semantics for heap data structures.

It is easy to generate good sequential code from Id-S. In fact, the language is even easier to compile than ML. It is also fairly straightforward to generate parallel code for some applications. For this, Shaw relies heavily on the single-assignment semantics of data structures and variables since these allow him to parallelize loops while ignoring output- and anti-dependences. In addition, the lack of pointers in the language simplifies alias analysis. The compiler also assumes it has access to the whole program so that it can see
all dependences across functions.

The performance results for Id-S on certain programs are impressive. Matrix multiply, for instance, achieves a speedup of 3.6 on 4 processors with respect to an efficient sequential version. For less structured programs, like Gamteb, speedup on four processors over the sequential version is negligible. Still, Shaw proves his point. Programming in Id-S is much easier than programming in an explicitly parallel language, and for the right kind of applications, Id-S can generate code that utilizes even a small SMP effectively.

The goal of the pHc compiler is to compile the entire pH language with no compromises. In this respect, its task is more complex than that of the Id-S compiler. Still, Id-S is useful because it shows how efficient pHc code might be if partitioning and scheduling were omniscient.

9.2 Concurrent and Parallel Versions of Haskell

In addition to pH, there exist two major parallel Haskell efforts: Concurrent Haskell and Glasgow Parallel Haskell (GPH).

9.2.1 Concurrent Haskell

Concurrent Haskell [31] arose as a solution to the difficulties of supporting modern graphical user interface programming in a lazy functional language. The authors noted that with a few carefully chosen concurrency extensions, such applications can be written easily as collections of communicating processes.

Concurrent Haskell adds two new elements to the original language: a forkIO primitive for explicit creation of new processes within a Haskell application and an Mvar primitive type of mutable locations derived from Id/pH M-Structures. Mvar objects are used to implement the synchronization abstractions needed by concurrent programs. These extensions to the language require major updates to the run-time system, but only minor changes to existing compilers.

Concurrent Haskell and pH differ in several important areas. First, parallelism in pH is implicit while in Concurrent Haskell, it is explicit in forkIO. Second, pH is designed to generate parallel multithreaded code. Concurrent Haskell is not designed to exploit multiprocessors, though it could be adapted to do so.
9.2.2 Glasgow Parallel Haskell (GPH)

While Concurrent Haskell uses concurrency to simplify programming, GPH has been designed to exploit the performance of multiprocessors. GPH is an extension of non-strict, purely functional Haskell that includes sequential and parallel combinators. The `seq` combinator is used to serialize two computations while the `par` combinator is used to indicate that an expression can be evaluated in parallel even its value has not been demanded. The principal execution vehicles for GPH are GUM [72] and GRANSIM [37].

GUM is a parallel run-time system built on top of the portable PVM communications infrastructure. Thus, it assumes a distributed memory machine architecture. GUM implements garbage-collected local and global heaps and a work-stealing thread scheduler. In GUM, GPH expressions identified with the `par` combinator are sparked at run-time. A pointer to a sparked-expression's thunk is placed in the local processor's spark pool. A spark is simply an indication that the expression can be evaluated concurrently, not that it must be evaluated. If the spark pool overflows, the processor can discard sparks. The values will be computed eventually if they are demanded by some part of the program.

Synchronization between GUM threads is handled in a manner similar to I-structures. When a thread starts to evaluate a thunk, it replaces the contents of the thunk with a black hole. If a subsequent thread attempts to evaluate the black hole, its context is saved, and the thread is placed on a suspension queue. Eventually, the value of the thunk is computed and written back, and any suspended threads are rescheduled.

GRANSIM was designed as a highly-parameterized parallel machine simulator for GPH code. The structure of GRANSIM is similar to that of GUM, with spark and thread pools, but it includes much more flexibility. For instance, the user can control thread migration, synchronous versus asynchronous communication, the granularity of work distribution, as well as the number of processors (possibly infinite). GRANSIM also supports extensive visualization facilities. It has been used to parallelize large functional programs including Lolita, a 47,000 line natural-language system.

There are several differences between pH and GPH. First, pH is lenient and includes side-effects while GPH is lazy and pure. More importantly, GPH is explicitly parallel while pH is implicitly parallel. Finally, pH is designed to exploit fine-grain, unstructured parallelism. GPH is better suited to coarse-grain programs since it is up to the programmer to expose
9.3 Cilk Scheduling

The work in Cilk [12] inspired the design of the scheduler in the pH Run-Time System. Cilk has shown that randomized work-stealing is an efficient way to schedule multithreaded programs. Indeed this type of scheduling is within a constant factor of optimal for the kinds of computations that can be written in Cilk. These are programs whose execution graphs are restricted so that all children created by a thread synchronize later in that same thread. The optimality result does not apply in general to pH because pH programs exhibit much more complex computation graphs.

The key property of a Cilk-like scheduler is that it performs a mostly depth-first traversal of the call-tree as the program executes. This means that most procedures that are spawned are actually executed on the same processor that spawned them, using the standard, efficient, sequential calling conventions. Only when a processor runs out of work and grabs work from another does the program pay the overhead of parallelism. As Schauser and Goldstein [57] demonstrated, non-strict pH programs in practice tend to exhibit the same kind of execution graphs as the strict programs generated by Cilk.
Chapter 10

Conclusion

This dissertation has described the design and implementation of a new compiler for pH. The principal goal of pH is to make parallel programming easy by making the compiler do most of the work of finding and exploiting concurrency. pH includes high-level language facilities like polymorphism and higher-order functions and combines these with non-strictness, implicit parallelism, synchronizing memory, and fine-grain barriers. This unique set of features makes it very challenging to generate efficient code for pH programs.

10.1 Contributions

The work described in this dissertation has produced the following contributions:

- A compiler for pH with complete support for all language features.
- A new code-generation algorithm that works directly on $\lambda^*$, an intermediate representation based on the $\lambda$-calculus. The multithreaded code generated by the compiler makes use of suspensive threads, an approach that has been largely ignored by previous efforts.
- In joint work with Jan-Willem Maessen, a complete Run-Time System for pH including a work-stealing scheduler and garbage collector.
- A novel tagged data representation that supports the implementation of synchronizing memory and garbage-collection with minimal impact on the range of scalars.

Our approach throughout this project has been to emphasize functionality over performance. In hindsight, we believe this strategy has served us well for two reasons. First, in considering all language features at once, we were forced to build all the important internal abstractions, and we are certain there are no holes in our underlying framework. We finally
know what it takes to build a complete pH implementation. Second, we have found that optimizing an existing feature, given that all the proper abstractions exist, is easier than adding a new feature to an existing implementation. We have a complete compiler and a solid reference against which we can design optimizations.

10.2 Future Work

In the preliminary evaluation of the code-generator for pH, we discovered that our compiler generates slow code that tends to consume a lot of memory. The data we gathered has pointed out several areas that need to be improved. We discuss the most important of these here.

10.2.1 Loops

The most critical change that needs to be made to the compiler is to improve the compilation of loops. The current strategy of implementing loops via full tail-recursive procedure calls leads to excessive overhead. Recall that tail-recursion in our compiler simply makes garbage-collection of loop frames more efficient. It does not decrease allocation or initialization costs. Loops bodies are less general than full procedures, so there seems to be a good opportunity to streamline the loop invocation mechanism. The challenge, however, is to do this while preserving correctness. This work involves both high-level changes to loops and low-level implementation improvements.

At the low-level, we need to improve how loop constants are stored and how loop activation frames are invoked. At the high-level, we need to devise new strategies for executing loops. The Id compiler for Monsoon implemented three kinds of loops—unbounded, bounded, and sequential—each with increasing efficiency. However, only unbounded loops truly implemented the proper non-strict semantics of the language in all cases. Unbounded loops are essentially identical to the tail-recursive procedure calls in pH. In Id, it was up to the programmer to specialize a loop using one of the more efficient schemes and to guarantee its correctness. It seems compelling, however, to perform these transformations automatically. It is possible that with relatively direct analysis the compiler could specialize a loop by considering the loop indices, the computations in the body, and the cost of the loop. Following our mostly-serial execution strategy, it is conceivable that many small
loops could be turned into sequential code, thus avoiding all the overhead associated with executing parallel loop bodies.

In a related area, the compilation of list comprehensions needs some work to make sure that it generates loops as often as possible, probably combined with the use of open lists. There is substantial code in the compiler already to remove intermediate lists via deforestation, but we have discovered that list comprehensions are often turned into non-tail-recursive function calls. In our Paraffins benchmarks, this has the effect of increasing the size of the working set to the point where garbage collection becomes a major overhead. With proper tuning and with improvements in the compilation of loops, it will be possible to dramatically improve the performance of list comprehensions.

10.2.2 Better Partitioning and Threading

The existing partitioning analysis in the pH compiler only considers intra-procedural dependences, so it needs to be extended to handle inter-procedural dependences. Such analysis will determine the ways in which input arguments are passed to procedures and will open up opportunities for increasing thread lengths and for getting rid of locations for arguments. Despite Schauser's [55] observation that the benefits of inter-procedural partitioning were modest, we believe this is a necessary addition to the compiler. It is conceivable that, since our baseline code is somewhat less strict than that produced by the Berkeley compiler, the results of inter-procedural partitioning will be more significant in pHc.

A related area that is also ripe for further work is the analysis of program data structures to improve threading. Our intermediate $\lambda^*$ code includes lots of destructuring of aggregate data structures like tuples and algebraic types. Often, the code to extract a value from a data structure ends up in a separate thread from the code that uses the value. Currently, this situation forces the compiler to insert synchronization code that, in fact, may not be needed. By tracking the creation, use, and lifetimes of data structures, the compiler might be able to determine that synchronization is not necessary because a particular data structure is produced completely before it is consumed. Thus producer and consumer code can be combined into a single large thread.
10.2.3 Procedure Cloning

One of the difficulties of partitioning is that a partition must be correct with respect to all contexts in which a procedure might be invoked. In many cases, the compiler must be conservative and has to produce a less efficient partition when there exists the possibility of a non-strict use of a procedure. With inter-procedural information, this situation will be less common, though across program modules or in higher-order functions, it still exists.

An interesting solution to this problem is to create not one but several versions of a procedure with different strictness properties. Each version of the procedure is compiled for a particular kind of context, and thus, it can make strong assumptions about the behavior of incoming arguments. To prevent code explosion, we could simply compile two version of every procedure: a general one applicable to all contexts and a more efficient, strict one that assumes all arguments have been computed when the procedure is invoked. In turn, the compiler can choose the proper version to apply at each call site, independent of all other call sites!

This strategy may prove particularly useful in improving loops and recursive functions. Often, the first iteration of a loop (or the outermost recursive function call) must be compiled with weak partitioning assumptions since it might occur in a variety of contexts. Subsequent iterations (recursive calls), however, are often completely strict. With cloning, these iterations can be made much more efficient at the small cost of doubling the static code size.

10.2.4 Inter-thread State

When neither better partitioning nor cloning can yield longer threads, the compiler must arrange to pass state from one thread to the next. The current strategy for accomplishing this task involves the use of the activation frame and lots of memory operations: every thread that needs a value created elsewhere must read it out of the frame. It seems, therefore, that there is ample need for an inter-thread register allocation algorithm, perhaps one that can be combined with procedure cloning.

This problem is markedly different from the standard inter-basic-block (global) register allocation problem. After all, the standard algorithms rely on the fact that the order in which basic blocks execute, in well structured code, can be approximated quite accurately
by static analysis. The order in which threads in a multithreaded program execute is, by
definition, dynamic and not computable at compile-time. Still, all is not lost. The compiler
can make a good guess at what might be the “most likely” schedule for thread execution,
given that it knows that programs are rarely non-strict and that the run-time system will
try to follow a mostly-sequential scheduling strategy. When viewed from this perspective,
the problem resembles trace-scheduling.

The dynamic component of such an algorithm would keep track of the scheduling deci-
sions made by the run-time system. On entry to a new thread, a simple test can determine
whether or not the previous thread was the one that should have preceded it in the “most
likely” compile-time schedule. If so, the new thread can execute quickly because it can
reuse many values already in the register file. If not, the new thread must reload these
values from the frame, possibly by reverting to code in a “slow” clone of the procedure.
At run-time, if the code rarely diverges from the “most likely” compile-time schedule, then
most inter-thread state will be passed in registers. Though not as efficient as pure static
register allocation, the scheme promises to be far superior to saving and restoring values
from memory.

10.3 Conclusion

We have come a long way with the completion of the first pH compiler. With it, we have
a solid foundation for applications development in pH and for future research into new
compilation strategies and optimizations. Nevertheless, we still have significant work ahead
until pH reaches the level efficiency that will make it compelling to a wide audience.
Bibliography


Appendix A

Semantics of $\lambda_S$

This appendix\(^1\) summarizes the semantics of the $\lambda_S$-calculus, the basis for $\lambda^*$, using a Term Rewriting System. For additional details, refer to [7]. It might seem odd to give a semantics for $\lambda_S$ rather than $\lambda^*$, but there are two good reasons for this. First, semantics research in $pH$ has been conducted using $\lambda_S$, so it is a tried and true formalism. Second, the differences between $\lambda^*$ and $\lambda_S$ are very few. Specifically, they are:

- Lambda abstractions and function applications in $\lambda^*$ allow multiple arguments. Nevertheless, these can be transformed into nested, single-argument definitions or chains of single-argument applications in $\lambda_S$ with no change in semantics.
- The '~' statement conjunction is used to mark threads in $\lambda^*$ but does not exist in $\lambda_S$. For the purposes of semantics, '~' can be treated like ';'.
- $\lambda^*$ uses a more general `Case` expression for conditionals than the `Cond` in $\lambda_S$. Still, the former can be rewritten as a nested version of the latter with no changes in semantics.
- $\lambda^*$ uses separate `iStore` and `mStore` statements, while $\lambda_S$ uses a single `Store`. The additional detail in $\lambda^*$ will enable future M-Structure optimizations, though currently both statements are implemented identically.
- $\lambda^*$ uses `iFetch` and `mFetch` expressions instead of `Fetch` and `Take` primitives. For the purposes of semantics, this choice make no difference.
- $\lambda_S$ includes syntax for dynamic terms, like `heap` and `empty()`, created during evaluation. These are not present in $\lambda^*$ because it is designed to be a static program representation.

\(^1\)Thanks to Jan-Willem Maessen for his help in preparing this material.
A.1 Syntax for $\lambda_S$

We begin by presenting the syntax of $\lambda_S$, in several parts. It will become evident that this syntax is more general than that of $\lambda^*$. However, after applying the kernelization rules (shown later), the two become very similar. First, we give the Expressions, Primitive Functions, and Constants:

$$E ::= x \quad \text{Identifier}$$

$$| E \ E \quad \text{Function application}$$

$$| \{ \ S \ in \ E \ \} \quad \text{Letrec block}$$

$$| \lambda x . E \quad \text{\(\lambda\)-abstraction}$$

$$| \text{Cond}(E,E,E) \quad \text{Conditional}$$

$$| \text{PF}_k(E_1,\cdots,E_k) \quad k\text{-ary primitive}$$

$$| \text{CN}_0 \quad \text{Constant}$$

$$| \text{CN}_k(E_1,\cdots,E_k) \quad \text{Constructor application}$$

$$| \text{CN}_k(x_1,\cdots,x_k) \quad \text{Constructor data structure}$$

$$| \text{allocate()} \quad \text{Allocation of heap cell}$$

$$| 0_j \quad \text{Heap cell}$$

$$\text{PF}_1 ::= \text{negate} \mid \text{not} \mid \text{Proj}_1 \mid \text{Proj}_2 \mid \cdots \quad \text{Unary primitives}$$

$$| \text{Fetch}$$

$$| \text{Take}$$

$$\text{PF}_2 ::= \cdots \quad \text{Binary primitives}$$

$$\text{CN}_0 ::= \text{Number} \mid \text{Boolean} \mid \cdots$$

The dynamic expression $\text{CN}_k()$ represents the data structure that results from evaluating $\text{CN}_k()$. For simplicity, the fields of such data structures, which follow I-Structure semantics, are not represented as individual memory cells. The dynamic expression $0_j$, on the other hand, does represent an individual I- or M-Structure memory cell.

Next, we present the Statements:

$$S ::= \epsilon \quad \text{Empty statement}$$

$$| x = E \quad \text{Identifier binding}$$

$$| S ; S \quad \text{Parallel statements}$$

$$| S \rightarrow S \quad \text{Sequential statements}$$

$$| \text{Store}(E,E) \quad \text{I- and M-Structure store}$$

$$| \text{heap} \quad \text{The global heap}$$

$$| \text{full}(0_i,V) \quad \text{Full heap cell}$$

$$| \text{empty}(0_i) \quad \text{Empty heap cell}$$

$$| \text{error}(0_j) \quad \text{Dynamic error}$$

The full(), empty(), and error() statement syntax is used to represent the presence state of I- and M-Structure cells.
To simplify the reduction rules, it is useful to classify some of the syntactic constructs into special subsets. The first of these subsets is \( V \), the syntax for Values. Values are expressions that constitute the valid results of programs. Put another way, a Value is not going to change \textit{observably} if more evaluation rules are applied to it, so it can be considered to have terminated.

\[
V ::= \lambda x. E \\
| \text{CN}_0 \\
| \text{CN}_k(x_1, \ldots, x_k) \\
| 0_j
\]

Next is the subset \( SE \) of Simple Expressions. These consist of an identifier or a Value.

\[
SE ::= x \mid V
\]

The syntactic subset \( H \) represents statements that have terminated. This is an important category when dealing with barriers because terminated statements are those that can be removed from a barrier pre-region. As we will see, once all terminated statements are removed, the barrier can discharge. Notice that all expressions used here are Values.

\[
H ::= x = V \\
| H ; H \\
| \text{heap} \\
| \text{full}(0_i, V) \\
| \text{empty}(0_i) \\
| \text{error}(0_j)
\]

The final subset is \( ET \) for Terminated Expressions. These are easy to understand: they consist of a Value or a letrec wherein all the statements and the \textit{in} expression have terminated.

\[
ET ::= V \mid \{ H \text{ in } SE \}
\]

\section*{A.2 Contexts in \( \lambda_5 \)}

A context is a device used to identify a portion of program syntax. That portion is the one that is to be modified by the reduction rules. For instance, the context \( C[x] \) for the expression:

\[
\text{Cond}(p, x + w, y * z)
\]
identifies the occurrence of $x$ in the first branch of the conditional. There are two kinds of contexts: $C$ is an expression context and $SC$ is a statement context.

$$C[] ::= []
\mid C[] E
\mid E C[]
\mid \lambda x.C[]
\mid \{ SC[] \text{ in } E \}
\mid \{ S \text{ in } C[] \}
\mid \text{Cond}(C[],E,E)
\mid \text{Cond}(E,C[],E) \mid \text{Cond}(E,E,C[])
\mid \text{PF}_k(C[],\ldots,E_k) \mid \text{PF}_k(E_1,\ldots,C[])
\mid \text{CN}_k(C[],\ldots,E_k) \ldots \text{CN}_k(E_1,\ldots,C[])
$$

$$SC[] ::= []
\mid x = C[]
\mid SC[] ; S
\mid SC[] \quad S
\mid S \quad SC[]
\mid \text{Store}(C[],E) \mid \text{Store}(E,C[])
$$

The definition of these contexts is not particularly clever. They can be used to pick out any arbitrary expression within any other arbitrary expression. But this is exactly what is needed by the reduction rules, since, in general, they are applicable anywhere a valid redex exists.

### A.3 Equivalence Rules for $\lambda_S$

In this section, we present three rules that can be used to manipulate expressions without changing their meaning. The first of these is the standard $\alpha$-renaming which is well-known from the $\lambda$-calculus. It allows bound identifiers to be renamed freely since the name of an identifier, intrinsically, contributes nothing to the meaning of a term. In the rules, the syntax "$e[t/x]$" is read as "substitute $t$ for all free instances of $x$ in expression $e$.”

$$\lambda x.e \quad \equiv \quad \lambda t.(e[t/x])$$

$$\{ SC[x = e]_\alpha \text{ in } e_0 \} \quad \equiv \quad \{ SC[t = e]_\alpha \text{ in } e_0 \}[t/x]$$

where statement contexts for renamed bindings are defined as follows:

$$SC[]_\alpha ::= []
\mid SC[]_\alpha ; S
\mid SC[]_\alpha \quad S
\mid S \quad SC[]_\alpha$$

216
The second rule shows how the parallel statement conjunction can commute and distribute across parentheses.

\[
\begin{align*}
\epsilon ; S & \equiv S \\
S_1 ; S_2 & \equiv S_2 ; S_1 \\
S_1 ; (S_2 ; S_3) & \equiv (S_1 ; S_2) ; S_3
\end{align*}
\]

The final rule deals with barriers and is more subtle than the previous two. It shows how barriers can be distributed in the face of terminated statements (from syntax subset \(H\)). This amounts to pulling terminated statements out from the pre-region of a barrier.

\[
(H ; S_1) \rightarrow S_2 \equiv H ; (S_1 \rightarrow S_2)
\]

### A.4 Reduction Rules for \(\lambda_S\)

At last, we reach the reduction rules for \(\lambda_S\), shown in Figure A-1. These rules tell us how to proceed from an initial term in \(\lambda_S\) to a final result, a term in the syntactic subset \(ET\).

To simplify the rules, we assume that all bound variables in the initial term are unique. The rules work as follows: if the term matches the left-hand side of a rule, then the term can be replaced with the right-hand side of that rule, and evaluation continues. If a term matches several rules, then an arbitrarily chosen matching rule can be applied.

Some points about notation: in all the rules, \(a\) represents a Simple Expression; the notation \([x]\) represents a free occurrence of identifier \(x\) in context \(C[x]\) or \(SC[x]\); \(t\) is a completely fresh identifier distinct from all others in the program; and, \(e'\) and \(S'\) are versions of \(e\) and \(S\) wherein identifiers have been renamed to avoid name conflicts. In addition, notice that the rules are divided into two types: kernel rules and dynamic rules. For the present, this division is not important, but we return to it in the next section.

**\(\beta_{let}\)** This first rule models function application. A new letrec scope is created. Within that new scope, the function parameter is bound to a fresh variable \(t\), and then \(t\) is substituted for all free occurrences of \(x\) in the body of the function, now the in expression of the letrec.

**Inst1–Inst3** The next three rules are different versions of instantiation. To instantiate a bound variable means to replace it with its value in the context of an expression. Inst1 instantiates within the in expression of a letrec, and Inst2 within statements of a letrec. Inst3 is an optional rule because it is not a fundamental part of the semantics but rather is
Rule

<table>
<thead>
<tr>
<th>Rule</th>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$(\lambda x . e_1) e_2$$</td>
<td>$\beta_{let}$</td>
<td>$D$</td>
</tr>
<tr>
<td>${ x = a ; S \text{ in } C[x] } \Rightarrow { x = a ; S \text{ in } C[a] }$</td>
<td>Inst1</td>
<td>$D$</td>
</tr>
<tr>
<td>$x = a ; SC[x]$</td>
<td>Inst2</td>
<td>$D^*$</td>
</tr>
<tr>
<td>$x = C[x]$</td>
<td>Inst3</td>
<td>(optional)</td>
</tr>
<tr>
<td>$x = { S \text{ in } e }$</td>
<td>Flat1</td>
<td>$D$</td>
</tr>
<tr>
<td>${ e \text{ in } e }$</td>
<td>Flat2</td>
<td>$D$</td>
</tr>
<tr>
<td>$e$</td>
<td>Name</td>
<td>$K$</td>
</tr>
<tr>
<td>$\Rightarrow { t = e \text{ in } t }$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>where $e \notin SE$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>${ S_1 \text{ in } { S_2 \text{ in } e } }$</td>
<td>LiftB</td>
<td>(optional)</td>
</tr>
<tr>
<td>${ S \text{ in } e } e_2$</td>
<td>LiftAp1</td>
<td>$K$</td>
</tr>
<tr>
<td>$e_1 { S \text{ in } e }$</td>
<td>LiftAp2</td>
<td>$K$</td>
</tr>
<tr>
<td>Cond(${ S \text{ in } e }, e_1, e_2$)</td>
<td>LiftCond</td>
<td>$K$</td>
</tr>
<tr>
<td>$PF_k(e_1, \ldots { S \text{ in } e }, \ldots e_k)$</td>
<td>LiftPF</td>
<td>$K$</td>
</tr>
<tr>
<td>Store(${ S \text{ in } e }, e_2$)</td>
<td>LiftM1</td>
<td>$K$</td>
</tr>
<tr>
<td>$\Rightarrow S ; \text{Store}(e, e_2)$</td>
<td>LiftM2</td>
<td>$K$</td>
</tr>
<tr>
<td>$\Rightarrow S ; \text{Store}(e_1, e)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Rightarrow { t_1 = e_1 ; \ldots t_k = e_k \text{ in } CN_k(t_1, \ldots t_k) }$</td>
<td>Cons</td>
<td>$K$</td>
</tr>
<tr>
<td>projj($CN_k(x_1, \ldots x_k)$)</td>
<td>Proj</td>
<td>$D$</td>
</tr>
<tr>
<td>$\Rightarrow x_j$, $1 \leq j \leq k$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cond(True, $e_1, e_2$)</td>
<td>CondT</td>
<td>$D$</td>
</tr>
<tr>
<td>$\Rightarrow e_1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cond(False, $e_1, e_2$)</td>
<td>CondF</td>
<td>$D$</td>
</tr>
<tr>
<td>$\Rightarrow e_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$PF_k(v_1, \ldots v_k)$</td>
<td>$\delta$</td>
<td>$D$</td>
</tr>
<tr>
<td>$\Rightarrow pf_k(v_1, \ldots v_k)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\epsilon \mapsto S$</td>
<td>$Bar1$</td>
<td>$D$</td>
</tr>
<tr>
<td>$S \mapsto \epsilon$</td>
<td>$Bar2$</td>
<td>(optional)</td>
</tr>
<tr>
<td>heap; $x = \text{allocate}()$</td>
<td>Alloc</td>
<td>$D$</td>
</tr>
<tr>
<td>$\Rightarrow \text{heap; empty}(0_j); x = 0_j$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>where $0_j$ is a new object identifier</td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty$(0_j); \text{Store}(0_j, v)$</td>
<td>Store</td>
<td>$D$</td>
</tr>
<tr>
<td>$\Rightarrow \text{full}(0_j, v)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>full$(0_j, v); \text{Store}(0_j, w)$</td>
<td>Error</td>
<td>$D$</td>
</tr>
<tr>
<td>$\Rightarrow \text{error}(0_j)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>full$(0_j, v); x = \text{Fetch}(0_j)$</td>
<td>IFetch</td>
<td>$D$</td>
</tr>
<tr>
<td>$\Rightarrow \text{full}(0_j, v); x = v$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>full$(0_j, v); x = \text{Take}(0_j)$</td>
<td>MFetch</td>
<td>$D$</td>
</tr>
<tr>
<td>$\Rightarrow \text{empty}(0_j); x = v$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$e$</td>
<td>HeapInit</td>
<td>$K$</td>
</tr>
<tr>
<td>$\Rightarrow { \text{heap in } e }$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* $a$ is restricted to values when rule is used dynamically

Figure A-1: Reduction Rules for $\lambda S$
used to express certain optimizations like loop unrolling: it allows a value to be instantiated within its own definition.

**Flat1, Flat2** The next two rules *flatten* a program by removing extra letrec scopes that might be created during evaluation, e.g., by the \( \beta_{\text{let}} \) rule. Notice that when scopes are merged in Flat1, renaming must be done to prevent name clashes.

**Name** This rule introduces a new scope and binds an identifier to an otherwise unnamed expression.

**Lift Rules** The effect of the lift rules is to pull nested statements out of sub-expression and into a common scope with the enclosing expression. The LiftB rule is optional because it is only needed as a convenience for top-level definitions. Definitions at deeper levels are handled by other rules.

**Cons** This rule "creates" an algebraic data structure from its specification. It generates a new scope where the expressions for the fields are named, and thus allows evaluation of these to proceed in parallel with the use of the data structure itself.

**Proj** Given a data structure, this projection rule yields the variable that identifies the computation for a selected field of the data structure.

**CondT, CondF** These rules evaluate conditionals. If the predicate is True, the term is rewritten to the first alternative. If the predicate is False, the term is rewritten to the second alternative.

\( \delta \) This rule actually represents many rules for primitive functions. It simply states that when all the arguments to a primitive function are evaluated (are values), then the term can be replaced with computation of the primitive function on those values.

**Bar1, Bar2** These rules show how to discharge barriers. In the first rule, the barrier discharges, as expected, when the pre-region is empty. In the second rule, the barrier discharges because the post-region is trivial (empty). This is an optional optimization rule.
Alloc  This rule models allocation of individual memory cells. Given the heap and a binding with an allocate expression, it converts the term into the heap, an empty new cell, and a binding for the new object identifier.

Store, Error  These two rules show the effects of stores on I- and M-Structure cells. A correct Store is one to an empty cell. The result is a full cell. A Store to a full cell, on the other hand, leads to an error.

IFetch, MFetch  The IFetch rules models an I-Structure access. Given a full cell, the value of the cell is extracted and bound to an identifier, and the cell remains full. In contrast, the M-Structure access changes the state of the cell to empty.

HeapInit  This rule is used to add a single heap to the initial term of the program.

A.5  An Efficient Reduction Strategy

The syntax and contexts given already are too permissive because they allow rules to be applied anywhere. This can lead the process of evaluation to unproductive reductions. This section shows how to apply the rules given in the previous section efficiently by first kernelizing the input term and by using a restricted set of contexts.

As we noted, the reduction rules are of two types: kernel rules and dynamic rules. The process of kernelizing a program is accomplished by applying only the kernel rules, repeatedly, to the initial term until no more can be applied. After this pre-processing step, the dynamic rules can be applied, but now, to a term that follows the much simpler syntax shown below:

\[
E ::= x \mid \lambda x.E \mid \text{SE SE} \mid \{ S \text{ in } E \} \mid \text{Cond(SE,E,E)} \mid \text{PF}_k(S E_1, \cdots S E_k) \mid \text{CN}_0 \mid \text{CN}_k(x_1, \cdots x_k) \mid \text{allocate()} \mid 0_j
\]

\[
S ::= x = E \mid \epsilon \mid S ; S \mid S --- S \mid \text{Store(S E,S E)} \mid \text{full}(0,V) \mid \text{empty}(0_i) \mid \text{heap} \mid \text{error}(0_j)
\]
In conjunction with the kernelization process, the efficient strategy also requires more restricted contexts to avoid reductions inside the bodies of λ-abstractions that have not been applied. The contexts are divided into two groups with, somewhat confusingly, the same names. The first group, for Evaluation, dictates the contexts where a rule can be applied. In other words, the Evaluation Contexts identify what portions of the program can be targeted for rewriting by one of the reduction rules.

**Restricted Evaluation Contexts**

\[
C[] ::= [] \\
| \{ SC[] \text{ in } E \} \\
| \{ S \text{ in } C[] \}
\]

\[
SC[] ::= [] \\
| x = [] \\
| SC[] ; S \\
| SC[] \text{ --- } S
\]

The second group of contexts are the Instantiation Contexts. These are the contexts used within the definition of the three instantiation rules, Inst1–Inst3. These work in concert with the Evaluation Contexts as follows. First, an Evaluation Context is used to identify a portion of the whole term which matches the left-hand side of one of the instantiation rules. Then, within the body of the instantiation rule, the Instantiation Contexts are used to pick a sub-portion to rewrite. For example, in the rule Inst1, the “\(C[x]\)” refers to an Instantiation Context, not an Evaluation Context.

**Restricted Instantiation Contexts**

\[
C[] ::= [] \\
| [] E \\
| Cond([],E,E) \\
| PF_k([],\ldots,E_k) \mid PF_k(E_1,\ldots,[])
\]

\[
SC[] ::= x = C[] \\
| SC[] ; S \\
| SC[] \text{ --- } S \\
| Store(C[],E) \mid Store(E,C[])
\]