Prototyping parallel functional intermediate languages

by

Andrew David Ben-Dyke

A thesis submitted to the Faculty of Science
of The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
Faculty of Science
The University of Birmingham
United Kingdom

October 1999

Abstract

Non-strict higher-order functional programming languages are elegant, concise, mathematically sound and contain few environment-specific features, making them obvious candidates for harnessing high-performance architectures. The validity of this approach has been established by a number of experimental compilers. However, while there have been a number of important theoretical developments in the field of parallel functional programming, implementations have been slow to materialise. The myriad design choices and demands of specific architectures lead to protracted development times. Furthermore, the resulting systems tend to be monolithic entities, and are difficult to extend and test, ultimatly discouraging experimentation. The traditional solution to this problem is the use of a rapid prototyping framework.

However, as each existing systems tends to prefer one specific platform and a particular way of expressing parallelism (including implicit specification) it is difficult to envisage a general purpose framework. Fortunately, most of these systems have at least one point of commonality: the use of an intermediate form. Typically, these abstract representations explicitly identify all parallel components but without the background noise of syntactic and (potentially arbitrary) implementation details. To this end, this thesis outlines a framework for rapidly prototyping such intermediate languages. Based on the traditional three-phase compiler model, the design process is driven by the development of various semantic descriptions of the language. Executable versions of the specifications help to both debug and informally validate these models. A number of case studies, covering the spectrum of modern implementations, demonstrate the utility of the framework.

Acknowledgements

Firstly, I thank my supervisor, Dr Tom Axford, for his assistance throughout this research project. I also acknowledge the members of my thesis group, Dr Lydia Kronsjö, and Dr Marta Kwiatkowska, for their helpful and timely advice. Furthermore, I acknowledge my examiners, Dr Antoni Diller of the University of Birmingham, and Dr Mike Joy of the University of Warwick, for their advice, in accordance with which this thesis has been modified. I am also grateful to my fellow research students for providing a relaxed and friendly working environment – in addition, Dr Howard Goodman must take the credit for the improvement in my grammar, punctuation, and general writing style. Finally, I would like to thank all of my family and friends for their patience and understanding, though special recognition must go to Claire for her dedicated support of my academic pursuits.

Contents

List of figures

List of tables

Glossary

This glossary is not intended to be exhaustive, and only includes entries for the most important freqently occurring items. These descriptions are based on a number of sources, include the references cited in the text as well as more general resources, such as the free on-line dictionary of computing [Howe, 1993].

abstract interpretation the execution of an abstract version of a program to deduce information about the program.

abstract machine a stylised processor design for executing an abstract machine code (which is usually the intermediate language of a compilation system) i.e. "a formal interpreter for the language which runs on a hypothetical machine" [Hennessy, 1990, page 114].

aggressive take a property of an abstract machine, whereby all of a function's arguments (after all pending updates have been performed) have to be present if evaluation is to proceed. As noted by Beemster [1994], the STG machine [Peyton Jones, 1992, rules 17 and 17a, section 5.6] is an example of this type of system.

algebraic data type a sum-of-product type using *constructors* to differentiate between each possible product [Bird and Wadler, 1988, pages 204–219]. Recursive and mutually recursive data types are permitted.

animation the process of making specifications executable for the purpose of experimentation and informal validation.

API (application-program interface) describes the formal interface through which user code can access a library's functionality. Typical details will include the argument-passing convention, and each method's input and output parameters. Additional information may include a list of possible side-effects and/or error returns.

boxed value any value which is indirectly referenced via an address pointer [Peyton Jones and Launchbury, 1991].

closure an operational structure used to represent a lambda expression, including an environment of its free-variable bindings [Peyton Jones, 1987, section 21.5, page 378].

constant applicative form (CAF) a top-level definition that may require to be updated during the lifetime of the evaluation. A typical STG′ CAF would be $nine = \mathrm{u} \rightarrow + \ 4 \ 5$ [Peyton Jones, 1987, section 13.2, page 224].

constructor a tag used to uniquely identify a product type of an algebraic data type [Peyton Jones, 1987, section 4.1, page 52].

continuation an instruction sequence, or function, that may be invoked as the final step of the current computation, and which represents "what to do next". In a physical implementation, a continuation is usually represented by a return address [Peyton Jones, 1987, sections 5.4 and 9.4].

continuation-passing style is a program notation that makes aspect of control flow and data flow explicit [Appel, 1992, page 2]. All user-defined functions take a continuation as an argument, and apply it to their result in order to effect a return to the main computation.

denotational semantics a set-based syntax-driven valuation function which maps a program directly to its meaning, or *denotation* [Stoy, 1977; Schmidt, 1986].

domain a set of values over which an ordering relation is defined, or, more specifically, the Scott domain [Burn, 1991, definition 2.2.21].

DMMP (distributed memory, message passing) – a traditional message-passing multiprocessor [Johnson, 1988].

evaluation transformer an identity function which has the operational side effect of forcing the evaluation of an expression beyond head normal form [Burn, 1991, chapter 5].

exception "an error, unusual condition, or external signal, that may set a status bit and may or may not cause an interrupt, depending upon whether or not the corresponding interrupt is enabled" [May, Silha, Simpson and Warren, 1994, section 1.3.1, page 368] and, typically, invokes a specialised handler to deal with the error.

free variable a variable referred to in an expression, but not bound by a local definition [Peyton Jones, 1987, section 2.2, page 14].

functional (programming) language any declarative, side-effect free language whose programs are sets of recursive function definitions.

garbage collection is "the automatic reclamation of computer storage" [Wilson, 1992, page 1]. This is achieved by disposing of any heap-allocated object which can no longer be reached by the running program. (see also *root set*).

Glasgow Haskell compiler one of the three main Haskell compilers, based on the STG language and STG-machine technology [Peyton Jones, Hall, Hammond, Partain and Wadler, 1993].

graph reduction a technique for evaluating non-strict functional programming languages which uses sharing to minimise the duplication of work [Wadsworth, 1971, chapter 4].

GMSV (global memory, shared variables) – a traditional shared memory multiprocessor [Johnson, 1988].

Haskell a non-strict, purely functional language whose features include support for higher-order functions, type classes and static, polymorphic typing, user-defined data types, functional I/O, and pattern matching [Hudak, Peyton Jones, Wadler and others, 1992].

higher-order functions "functions are treated as first-class values in a language – allowing them to be stored in data structures, passed as arguments and returned as results" [Hudak, 1989, section 2.1, pages 382–383].

Hindley–Milner type-inference algorithm the classic approach to polymorphic type checking in a functional programming system [Milner, 1978].

intermediate language any language that is used as a temporary representation during the compilation of a source language to a target language.

interpreter "a piece of software that directly executes a source program" [Watson, 1989].

metalanguage "a language used to define another language" [Watson, 1989, section 1.4.2, page 14].

MIMD (multiple instruction, multiple data) – most commercial multiprocessors and collections of workstations fall into this architectural category [Flynn, 1972].

non-determinism a property of a computation which may (arbitrarily) return different results [Stoy, 1977, page 201].

non-strictness a property of an evaluation strategy such that an expression is only evaluated when its value is actually needed (normal-order reduction [Peyton Jones, 1987, section 2.3, page 25]).

powerdomain each element of a powerdomain is a set of elements of the domain from which it was formed. Powerdomains can be used in a denotational semantics to model non-determinism [Stoy, 1977, page201].

primitive function builtin routines similar to the lambda-calculus δ-rules, and the only way to perform computations on unboxed values.

prototyping "is the process of constructing software for the purpose of obtaining information about the adequacy and appropriateness of the designers' conception of a software product" [Balzer, Gabriel, Belz, Dewar, Fisher and others, 1988, page 8].

referential transparency the ability to replace any sub-expression by others possessing the same value without changing the final value of the mathematical expression [Bird and Wadler, 1988, page 2]. This is often summarised as: "equals can be replaced by equals" [Hudak, 1989, page 362].

RISC (Reduced Instruction Set Computer) machine the salient features of this class of processor include [Kane and Heinrich, 1992, chapter 1, pages 1–22]: one instruction completed per cycle; simple addressing modes and instruction formats; sufficient on-chip memory (registers and cache) to overcome the processor/memory bottleneck; and a reliance on optimising compilers to obtain the best possible performance.

root set a list of the heap addresses which are live in the local state [Wilson, 1992, section 1.2]. Using this as the main input, it must be possible for the garbage collector to identify all of the live closures of the entire system.

SIMD (single instruction, multiple data) – vector/array processors, often referred to as data-parallel machines [Flynn, 1972].

sparking the creation of a new thread to reduce an expression [Clack and Peyton Jones, 1986, section 2.1].

speculative evaluation an approach to increasing available parallelism by sparking threads to reduce non-essential expressions [Mattson Jr., 1993a, chapter 3, page 39].

STG (Shared Term Graph) language is the abstract machine code of the STG machine, and can be viewed as "a very austere purely-functional language" [Peyton Jones, 1992, section 4].

STG′ language a variant of the STG language which serves as the foundation for the prototyping system. The sequential semantics of the language is presented in chapter 4.

STG machine (Spineless Tagless G-machine) is an abstract machine designed to support non-strict higher-order functional languages [Peyton Jones and Salkild, 1989; Peyton Jones, 1992].

syntactic sugar any syntactic construct added solely for the purpose of improving programmability. The removal of these expressions is known as de-sugaring.

syntax driven a property of a language processor which takes its structure directly from the abstract syntax.

thread an independent process which computes the value of one expression and then terminates [Peyton Jones, 1989, evaluate-and-die model, page 178] (see also *sparking*).

thunk a closure which represents an expression not in head-normal form [Peyton Jones, 1992, section 3.1].

`ticky-ticky` **profiling** a feature of GHC, whereby the run-time system records the number of updates, the number of constructors entered etc. The system is so named because "that's the sound a Sun4 makes when it is running up all those counters (slowly)" [AQUA Team, 1993, section 9, page 36].

time-out "A period of time after which an error condition is raised if some event has not occured. A common example is sending a message. If the receiver does not acknowledge the message within some preset time-out period, a transmission error is assumed to have occured." [Howe, 1993]

type inference the process of deducing a program's type attributes from its syntax, as typified by the Hindley–Milner system [Milner, 1978].

unboxed value any value which can be represented using a machine literal, including, for example, 32-bit integers and 64-bit floating point numbers [Peyton Jones and Launchbury, 1991].

Chapter 1

Introduction

1.1 Motivation

Non-strict higher-order functional programming languages are elegant, concise, mathematically sound and contain few environment-specific features. Furthermore, current implementations of functional programming languages generate sequential code of a comparable efficiency to that of their imperative rivals. This combination suggests the possibility of architecture-independent parallel programming, and the validity of this approach has been established by a number of experimental compilers [Hill, 1994; Chakravarty, 1994; Hammond, Mattson Jr. and Peyton Jones, 1994; Hudak, 1991]. However, a would-be designer of a parallel functional system is faced with three major obstacles:

1. due to the large number of dimensions involved and to the lack of a common benchmarking system, it is extremely difficult determine which components are central to the performance of the system.
2. having selected and integrated the components, the cost of developing an efficient implementation for just one platform is considerable.
3. once the base implementation is complete, experimentation with any but the most trivial of subsystems may require significant effort.

What is needed is a system to rapidly develop and test ideas before committing to a full-scale implementation. However, as each existing implementation tends to prefer one specific platform and a particular way of expressing parallelism (including implicit specification) it is difficult to envisage a general purpose framework.

Fortunately, most of these systems have at least one point of commonality: the use of an intermediate form [Peyton Jones, 1987]. Typically, these abstract representations explicitly identify all parallel components but without the background noise of syntactic and (potentially arbitrary) implementation details. To this end, this thesis outlines a framework for rapidly prototyping such intermediate languages, split into three stages:

language specification the language is specified in terms of its syntax, type rules [Milner, 1978] and denotational semantics [Stoy, 1977]. This provides the reference model against which to test the output of the subsequent stages.

parser construction a number of parallel languages have been outlined [Hill, 1994; Hudak, 1991; Kelly, 1989; Burton, 1984] and so it is important to verify that the proposed abstract form can act as a suitable target (for as large a subset of these as possible).

compilation rules to provide a degree of architecture independence to the source languages, the code generator must produce efficient output for a variety of diverse architectures. This stage is driven by the development of an operational semantics for the intermediate language.

The design process is driven by the development of semantic models of the stages, and these are used primarily to validate and motivate the parse and compilation rules. To improve confidence in the models themselves, executable version of the specifications, written in the functional programming language Haskell, are constructed. While some parts of the framework could be automated, it is worth stating that we have made no attempt to develop an automatic system of the sort typified by CERES [Tofte, 1990].

1.2 Overview

1.2.1 Background

By critically examining the relevant literature, chapter two motivates the central work of this thesis: the design of a prototyping framework for parallel intermediate languages.

1.2.2 Prototyping parallel functional intermediate languages

Chapter three describes an approach to the design of an explicitly parallel intermediate language for use during the compilation of non-strict higher-order functional programming languages. The framework is based upon the development of both a denotational and operational model for the intermediate language, which are then used to produce specifications for the parser and code generator. Haskell animations of these components aid with both debugging and informal validation. (For a more detailed and example-driven description of the prototyping system see [Ben-Dyke and Axford, 1995], which is reproduced in appendix A.)

1.2.3 The sequential STG$'$ language

Chapter four describes the STG$'$ language, a variant of the *Shared Term Graph (STG) language*, both in terms of its abstract and concrete syntax, and denotational semantics. A Hindley–Milner style type-inference algorithm is also presented, which serves to restrict the language and produces information useful to a compilation system.

1.2.4 Expressing parallelism – static models

In chapter five a number of guidelines are presented for adding support for parallelism into the sequential STG$'$ language, as described in chapter 4. Typically, this involves extending the abstract syntax, adding language restrictions, and developing a denotational model of the parallel components. The examples used to motivate each of the steps are, where possible, based on the constructs presented in section 2.4. While the issues of language design are not directly addressed, MacLennan's principles [1987, page 547] serve as a useful guide, and are thus reproduced in table 5.1.

1.2.5 Managing parallelism – operational models

Chapter six discusses the development of an operational description to augment the denotational semantics of the parallel STG′ language (see chapter 5). The STG machine provides the basic recipe, into which the parallel ingredients, including threads, messages, and shared memory, are added. To facilitate testing and debugging, the animation of the model, which is essentially a state-transition system, is also considered. The final description is then used by chapter 8 to provide the foundation upon which the compilation system is built.

1.2.6 Simulating the target architecture

Chapter seven describes the simulator used to test and debug the output of the STG′ compiler (see chapter 8). A RISC-like instruction set, based on the DEC Alpha processor family, serves as the interface between the two systems. The simulator is interpretive and is specified using the state-transition notation presented in chapter 6. While overall performance is relatively poor, the extensible nature of the state-transition model is more important for this particular application.

1.2.7 Compilation rules

Chapter eight describes how the state-transition model can be used to model a compilation system. Particular emphasis is placed on encoding important optimisations, including register allocation, closure layout, and dead-code elimination. The validity of this approach is demonstrated by developing a compilation system for a subset of the sequential STG′ language.

1.2.8 Prototyping parallel functional intermediate languages

In chapter nine the use of the prototyping framework is illustrated by four case studies. Each of the studies are based upon existing well-known systems, and, between them, include examples of the main programming abstractions used in modern parallel functional programming and cover both message-passing and shared-memory architectures. The first study is based upon shared-memory Haskell, and considers the introduction of parallel threads into the STG′ language. This provides a simple overview of the methodology, and serves as a foundation upon which the other case studies build. The second moves on to consider GUM Haskell [Trinder et al., 1996]. While the static semantics are very similar to those of the first case study, the operational model is far more complex, and demonstrates how message passing can be modelled by a state-transition system. The third investigates the data placement primitives of para-functional Haskell –this proves interesting both in terms of the denotational and operational models. Skeletal parallelism is the subject of the final case study, dealing with farms, pipes and divide-and-conquer skeletons.

1.2.9 Summary, evaluation, and further work

Chapter ten concludes the thesis by re-stating the main contributions of the work, and attempting to evaluate the prototyping framework. Finally, there is a discussion of the limitations of the proposed approach, and possible areas for future research are examined.

Chapter 2

Background

2.1 Introduction

By critically examining the relevant literature, this chapter motivates the central work of this thesis: the design of a prototyping framework for parallel intermediate languages.

Section 2.2 introduces the field of parallel processing, while section 2.3 deals with the specifics of parallel functional programming. This leads on to a review, in section 2.4, of the idioms used by explicitly-parallel functional programming languages. Section 2.5 reviews the existing approaches to prototyping, and the chapter is then summarised in section 2.6.

2.2 Parallel processing

2.2.1 Architectural taxonomies

The term *parallel* can be used to describe a wide range of architectures, with the only common denominator being the use of more than one processing element. For the purpose of this thesis, such systems are assumed to comprise many similar processors, connected by a reliable communication mechanism, co-operating to solve a single task or problem. Many different styles of parallel computers have been developed, and figure 2.1 shows a number of common configurations (the blocks represent processors, memory, communication networks, or host processors). Indeed, there is sufficient variety [Duncan, 1990] that a number of different taxonomies have been developed. Flynn [1972] categorised machines based upon the number of instruction and data streams:

SISD (single instruction, single data) – the classic von Neumann architecture, encompassing most modern uniprocessors. Examples include the DEC Alpha AXP architecture [Sites, 1992], the SPARC family [Sun Microsystems, 1988], and the PowerPC [May et al., 1994].

SIMD (single instruction, multiple data) – vector/array processors (often referred to as data-parallel machines). Examples include the AMT DAP, Thinking Machines' CM-200, and the MasPar MP-1 (all of which have been surveyed by MacDonald [1992]).

MISD (multiple instruction, single data) – no practical examples of this class exist.

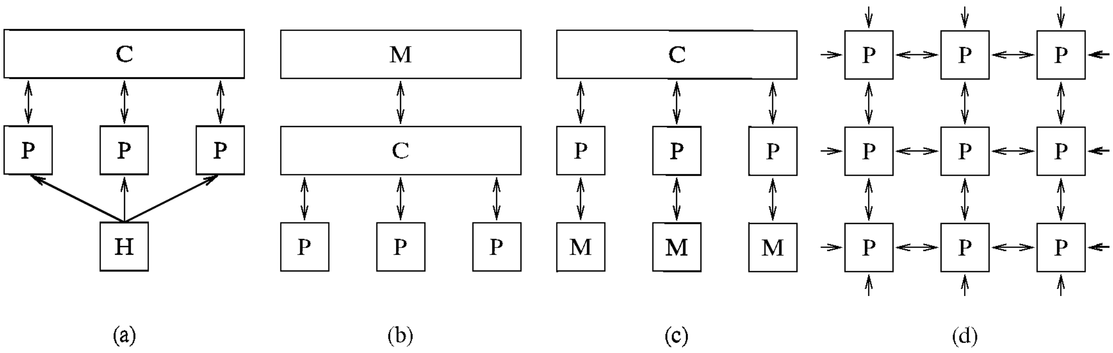


Figure 2.1: Four examples of parallel architectures: (a) vector processor (SIMD); (b) classic shared memory (GMSV); (c) loosely-coupled message passing (DMMP); (d) constant-valence message passing (DMMP, but with the memory components not shown)

MIMD (multiple instruction, multiple data) – most commercial multiprocessors and collections of workstations fall into this category. Examples include both shared-memory machines, such as the KSR and Sequent Symmetry, and message-passing systems, including Thinking Machines' CM-5, the NCUBE range, and systems based on the Inmos Transputer. (Oren and Ramanathan [1993] include an overview of each of these machines in their survey paper.)

Johnson [1988, figure 1, page 45] noted that the last category, MIMD, was too coarse, and divided it into the following classes:

	shared variables	message passing
global memory	GMSV shared memory	GMMP
distributed memory	DMSV hybrid	DMMP message passing

One failing common to both of these taxonomies, however, is that they convey no information with regards to a machine's "size". The Erlangen classification system, developed by Händler [1982], uses the triple (K, D, W) as a representation, where K is the number of processors, D is the number of ALUs (Arithmetic Logic Units), and W is the word length of each ALU. If pipelining is used, the notation is extended to $(K \times K', D \times D', W \times W')$, where the multipliers are the pipeline depths (macro-, instruction- and arithmetic-pipelining respectively). The system also allows representations to be combined using the following operators:

+ indicates the existence of more than one structure that operates independently in parallel.

***** indicates the existence of sequentially ordered structures where all data is processed through all structures.

v indicates that a certain system may have multiple configurations.

Skillicorn [1988] extended this idea to include descriptions of the interconnection topology (including both processor-to-processor and processor-to-memory networks). Indeed, Bönniger, Esser and Krekel [1993] also traded conciseness for accuracy by increasing the number of items to 350 (split across 14 groups). Schlesinger and Kuehn [1993] have taken this concept to its natural limit by developing an architecture-description language. Another approach is adopted by Culler et al. [1993], whose LogP model uses the performance characteristics of the communication mechanism as the primary attributes:

L - an upper bound on the latency involved with communicating a word-length message from source to destination.

o - the overhead attributed to the transmission or reception of each message (during which time a processor can engage in other activities.)

g - the minimum gap allowed between consecutive message transmission or reception. The reciprocal gives the per-processor bandwidth.

P - the number of processors.

2.2.2 Amdahl's law and the corollary of modest potential

While intuition would suggest that n co-operating processors should be n-times faster than a single processor, Amdahl [1967] argued the case against parallel processing as a means of achieving large scale computations. He showed that the maximum theoretical speedup is merely the reciprocal of the percentage of time spent performing serial computation (this limiting factor is known as the serial fraction.) Indeed, most problems are unlikely to experience even a 100-fold improvement, and this insight resulted in a degree of scepticism regarding the viability of massive parallelism. However, Gustafson [1988] addressed these concerns by pointing out that there are, in fact, two distinct approaches to parallel processing: *fixed size, reduced time*, used whenever user acceptance is important or there are real-time constraints; and, secondly, *bounded time, increased size*, where the increased power is used to improve either the accuracy or problem size of the computation.

Gustafson then showed that Amdahl's law only applied to the case where the serial fraction is independent of the number of processors, i.e. the fixed-size, reduced time approach. By considering the alternative method, a new law of *scaled speedup* was defined such that the speedup is approximately equal to the number of processors used (thereby confirming the original intuition).

As a final cautionary note, Snyder [1986, page 291], taking the bounded-time approach for an $\mathcal{O}(n^4)$ algorithm as an example, showed that it would require 100 million processors to increase the problem size by two orders of magnitude – this lead to the *corollary of modest potential*:

> "Because its benefit is so modest, the whole force of parallelism must be transferred to the problem, not converted to "heat" in implementational overhead."

2.3 Architecture independence through functional programming

2.3.1 The software crisis and parallel languages

While parallel processing seems to offer high performance at a low cost, the recent failure of the Thinking Machines Corporation [Markoff, 1994] would suggest that it is not a commercially-viable option. Skillicorn [1990, page 38] states that the major problem is:

> "There is currently no way to develop software for parallel computers and expect it to have a long lifetime."

The situation is akin to the original software crisis of the 1960s, and, as then, either the hardware or software components (or both) need to be improved.[1] A number of promising examples of the former approach exist, including the MIT Alewife [Agarwal et al., 1991], the Stanford DASH [Lenoski et al., 1992], and the Tera Computer Company's Multi-Threaded Architecture [Smith, 1990]. However, this thesis takes the latter path, with McColl [1995, page 42] providing the necessary motivation:

> "This *software-first* approach has a great deal of merit given that hardware is changing rapidly and that the cost and time required to produce software makes architecture-independence in software a major goal."

This opinion is widely held, as illustrated by the long list of *architecture-independent* programming languages: Dino, High Performance Fortran, Lucid, Orca, Proteus, PCN, Sisal, Split C, SR, etc. (Cheng [1993] has surveyed over 30 different parallel languages, including those listed here, in addition to a wide selection of communication libraries, and performance, debugging, and visualisation tools.) Although each of these languages has achieved a degree of success, Cook, Pancake and Walpole [1994, section 6] have recently stated that:

> "Parallel programming is difficult, under-supported, and unlikely to achieve impressive speedups on most applications."

Peyton Jones [1989, section 2.3, page 176] argues that this is a direct result of the underlying programming model, and re-applies Backus's *fat and weak* criticism (see section 2.3.2) to parallel languages, stating that:

> "A parallel imperative program specifies in detail many *resource-allocation decisions* which the parallel functional program does not mention at all."

Note that this thesis does not claim that parallel functional programming is the only solution, merely that it is a promising approach to the problem of developing architecture-independent software.

2.3.2 Can programming be liberated from the von Neumann style?

In 1977, the ACM Turing award was presented to Backus [1978], the developer of Fortran and BNF. During the lecture, he echoed the work of Landin [1966] by criticising conventional programming languages:

> "Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor – the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful properties for reasoning about programs."

[1] The BSP (Bulk-Synchronous Parallel) model, first proposed by Valiant [1990], is an attempt to decouple these two approaches.

He then proceeded to advocate the use of functional programming to circumvent the imperative *intellectual bottleneck*, and, despite the recent advances in imperative programming [Stroustrup, 1991], many of these arguments still hold [Hughes, 1989; Hudak and Jones, 1994].

The key properties of functional languages are described below (for a more complete overview of of functional programming, [Hudak, 1989] and [Bird and Wadler, 1988] are highly recommended):

declarative "functional programming is often described as expressing *what* is being computed rather than *how*" [Hudak, 1989, page 361]. Essentially, the programmer is free to concentrate on the underlying algorithm without having to deal with unimportant details, such as sequencing and memory management.

formal semantics the lambda calculus provides the theoretical foundations of functional languages [Barendregt, 1981; Hankin, 1994]. Furthermore, specifying the denotational semantics [Stoy, 1977; Schmidt, 1986] of such languages is straightforward – indeed, the meta-language used is syntactically very similar to the lambda calculus.

referential transparency due to the absence of side-effects, any sub-expression can be replaced by others possessing the same value without changing the final value of the mathematical expression [Bird and Wadler, 1988, page 2]. This is often summarised as: "equals can be replaced by equals" [Hudak, 1989, page 362].

In addition, most modern functional programming languages will also provide the following features:

algebraic data types a sum-of-product type using *constructors* to differentiate between each possible product [Bird and Wadler, 1988, pages 204–219]. Recursive and mutually recursive data types are permitted.

higher-order functions "functions are treated as first-class values in a [functional] language – allowing them to be stored in data structures, passed as arguments and returned as results" [Hudak, 1989, section 2.1, pages 382–383]. Through the use of such functions, it is possible to develop powerful combining forms, such that "it is as though the programming language can be extended with new control structures whenever desired" [Hughes, 1989, section 3, pages 99–101].

syntactic sugar any syntactic construct added solely for the purpose of improving programmability. Examples include pattern matching, guarded expressions, and list comprehensions [Hudak et al., 1992].

type inference the process of deducing a program's type attributes from its syntax, as typified by the Hindley–Milner system [Milner, 1978].

This combination of features provides an expressiveness and freedom from trivial details (such as sequencing and memory management) that is, as yet, unrivalled by modern imperative languages, such as C++ [Stroustrup, 1991]. Furthermore, the two traditional failings of functional languages, namely their inefficiency and poor input/output facilities, have been addressed by recent advances in compiler design [Peyton Jones, 1992; Plasmeijer and van Eekelen, 1993a] and category theory [Wadler, 1992; Peyton Jones and Wadler, 1993] respectively.

Non-strict evaluation

Often referred to as call-by-need, normal order reduction [Peyton Jones, 1987, page 25], or lazy evaluation, non-strictness is a property of an evaluation strategy such that an expression is only evaluated when its value is actually needed. Hughes [1989, page 98] argues that, in combination with higher-order functions, lazy evaluation "can contribute greatly to modularity, the key to successful programming." However, implementing non-strictness is complicated [Peyton Jones, 1987] and relies on the existence of analysis techniques for removing unnecessary laziness [Beemster, 1994; Peyton Jones and Partain, 1994; Burn, 1991]. However, recent benchmark results prompted Hartel et al. [1996, section 6.3.2] to note the following:

> "As the Glasgow Haskell compiler shows, if the compiler can exploit strictness at the right points, the presence of lazy evaluation need not be a hindrance to high performance. This implementation is actually faster than most of the strict implementations."

Even so, the functional programming community is split over the issue of strict versus non-strict languages, with Standard ML [Harper, Milner and Tofte, 1990] and Haskell (see the following section) representing the two camps. To help settle the matter, it is worth considering what Hill [1994, chapter 1, page 1] has to say about laziness in the context of parallelism:

> "We have found that non-strictness opens the door to particular techniques and parallel algorithms, in just the same way that it has opened the eyes of functional programmers over the last decade."

Therefore, for the purpose of this thesis, it will be assumed that the increased convenience and expressiveness of non-strictness outweigh the (perceived) operational deficiencies.

Haskell

Haskell is a modern functional programming language whose features include "higher-order functions, non-strict semantics, static polymorphic typing, user-defined datatypes, pattern matching, list comprehensions, a module system, and a rich set of primitive datatypes" [Hudak, Peyton Jones, Wadler and others, 1992, page 1]. A number of free compilers, interpreters, tools, and libraries exist, and, in a recent benchmark experiment [Hartel, 1994], the Glasgow Haskell compiler [Peyton Jones et al., 1993] consistently outperformed the best compilers for other lazy languages, including Clean [Nöcker et al., 1991], and Fast [The FAST project team, 1993]. Even when compared against strict languages, GHC managed to finish second out of twenty five [Hartel et al., 1996, section 6.2.1].

Appendix B contains a number of example Haskell definitions, including `map`, `foldl`, `fib`, `primes`, and `queens`.

2.3.3 Graph reduction

Graph reduction [Wadsworth, 1971, chapter 4] is, arguably, the standard approach to implementing functional languages [Peyton Jones, 1987, parts II and III]. It provides a framework for both controlling the order of reduction, e.g. normal order versus applicative order, and managing the sharing of common sub-expressions. Combinators [Turner, 1979] are typically used as the abstract machine language, with the instruction set being optimally generated for each program [Hughes, 1984].

Clean [Nöcker, Smetsers, Plasmeijer and van Eekelen, 1991] is one of the few modern languages not to use graph reduction, instead being based upon *term graph rewriting* [Sleep, Plasmeijer and van Eekelen, 1993]. The primary advantage of this approach is that a formal proof of *soundness* has been developed [Barendregt, van Eekelen, Kennaway and others, 1987]. Despite these theoretical differences, when comparing the ABC machine [Plasmeijer and van Eekelen, 1993a] (the abstract machine used by Clean), with, for example, the STG machine (a graph-reduction system), it is clear that the implementation strategies are very similar.

For a brief period, specialised hardware for running functional languages was considered, with examples including Cobweb [Hankin, Osmon and Shute, 1985], the AMPS (Applicative Multi-Processors) project [Keller, Lindstrom and Patil, 1979], ALICE (Applicative Language Idealised Computing Engine [Darlington and Reeve, 1981]), and FLAGSHIP [Watson and Watson, 1987]. However, all of these offerings could not compete against commercially-developed traditional uniprocessors, and have therefore become obsolete.

2.3.4 Parallel functional programming: an introduction

Burge [1975] was probably the first to recognise the potential advantages of parallel functional programming. In addition to the high level of abstraction offered by the paradigm, the absence of side effects offers the following benefits in a parallel context [Peyton Jones, 1989, section 2.2, pages 175–176]:

- functional languages are *implicitly parallel*, i.e. it is possible for a compiler to automatically detect what sub-expressions can be safely evaluated in parallel (see section 2.4.1).

- the programmer does not have to specify the low-level synchronisation of variables as sharing is automatically handled by the run-time system.

- ideally, the language retains its original denotational semantics (modulo resource allocation), therefore the same formal reasoning techniques can be applied. This also means that all programs are guaranteed to be deadlock free (unless, of course, the sequential program also fails to terminate).

- a program's result will be independent of implementation details such as scheduling, partitioning, and load balancing. This also implies that parallel programs can be debugged on a sequential architecture.

- being based on the lambda calculus, there is no architectural bias.

Concerning the first two points, critics would argue that a human programmer must specify the low-level behaviour if performance is to be a primary goal (recall the corollary of modest potential from section 2.2.2). Peyton Jones [1989, section 2.3, pages 176–177] counters by noting that there has always been resistance to abstraction:

> "For example, in the beginning all programs were written in assembly language, and compilers were distrusted because they were unlikely to do as good a job of register allocation as a human programmer."

However, he does concede that parallel functional programming relies on the existence of low- and mid-level parallel imperative technology, and it is probably in these areas that most work needs to be done.

For a more detailed overview of parallel functional programming, both [Hammond, 1994] and [Peyton Jones, 1989] are recommended.

2.4 User-level annotations and expressions

When examining a particular approach to parallel functional programming it is important to differentiate between what is being expressed and the notation used. For instance, consider the following code fragments, both of which have similar operational behaviours:

$f\ (g_1\ arg_{11} \ldots arg_{1a_1}) \ldots (g_n\ arg_{n1} \ldots arg_{na_n})$ where f :: speculation $a_1 \rightarrow \cdots \rightarrow$ speculation $a_n \rightarrow a$ $f = \ldots$	sandwich $f\ job_1 \ldots job_n$ where $f = \ldots$ $job_1 = g_1\ arg_{11} \ldots arg_{1a_1}$ $\vdots$ $job_n = g_n\ arg_{n1} \ldots arg_{na_n}$
Burton's type annotations [Burton, 1987]	Vree's sandwich expression [Vree, 1989]

For the purpose of this section, the primary focus is on the intended behaviour rather than any syntactic differences, with the predominant abstractions being:

implicit specification it is left to the compiler to identify and harness those portions of the program which will benefit from parallel evaluation.

bulk data types primitive operations for manipulating a group of objects *as a whole* provide the main sources of parallelism.

skeletal parallelism a skeleton is an algorithmic template, with both a denotational and operational reading, into which problem-specific routines are slotted.

low-level task control the programmer is given full control over the creation, placement, and scheduling of threads of computation.

Each of these paradigms is examined in turn in sections 2.4.1 to 2.4.4.

2.4.1 Implicit specification

In the absence of programmer-supplied hints, the compiler has to rely on abstract interpretation (or, when this fails, run-time profiling) to both detect potential parallelism, and to ascertain if the overhead is justified. The final product of the analysis phase is an explicitly parallel program, using one or more of the abstractions described in the following sections. The main tools used are *strictness* and *complexity* analysis, with the former identifying those expressions which are sure to be evaluated [Burn, 1991], and the latter generating cost models of the reduction of these expressions [Maheshwari, 1990]. Of the two, strictness analysis is probably the most evolved, as it has applications outside the world of parallel functional programming [Howe and Burn, 1994].

The advantages offered by this paradigm are the reduced programmer burden coupled with architecture independence. The viability of this approach has been demonstrated by a number of systems, including serial combinators [Goldberg, 1988b] and evaluation transformers [Burn, 1989]. However, at present, such systems only introduce unstructured **par** combinators, leading to sub-optimal performance.

merging	$\{1,2,9\} \cup \{1,3,6\}$	=	$\{1,2,3,6,9\}$
	$\langle 1,2,9\rangle \mathbin{+\!\!+} \langle 1,3,6\rangle$	=	$\langle 1,2,9,1,3,6\rangle$
	$\mathit{zip\_with}\ (+)\ \langle 1,2,9\rangle\ \langle 1,3,6\rangle$	=	$\langle 2,5,15\rangle$
selection	$\{x \in \{1,2,9\} \mid x \leq 5 \bullet x\}$	=	$\{1,2\}$
permutation	$\mathit{reverse}\ \langle c,a,t\rangle$	=	$\langle t,a,c\rangle$
reduction	$\mathit{fold}_{\mathit{left}}\ (+)\ 0\ \langle 1,3,6\rangle$	=	10
scanning	$\mathit{scan}_{\mathit{left}}\ (+)\ 0\ \langle 1,3,6\rangle$	=	$\langle 0,1,4,10\rangle$
apply to all	$\mathit{map}\ (1+)\ \langle 1,3,6\rangle$	=	$\langle 2,4,7\rangle$

Table 2.1: Some examples of collection-oriented operations

2.4.2 Bulk data types

The fundamental component of a data-parallel language is the monolithic *apply-to-all* function which simultaneously acts on a large collection of data. Sipelstein and Blelloch [1991, page 510] note that:

> "A collection-oriented language is characterised by two features: the kinds of collections it supports and the operations permitted on those collections."

Example collections include arrays, bags, sets, mappings, and trees [Maaßen, 1992], with the operations including merging, selection, permutation, reduction, scans, and the ubiquitous map – see table 2.1 for a number of Haskell-style examples. Some of the operations require an ordering to be imposed on the elements of the collection, and this is typically either array-based or uses a (possibly unique) key. The data placement, pattern of communication, and synchronisation are usually implicitly specified by whichever operation is used, although a number of languages allow the programmer full control over some of these areas [Bala, Ferrante and Carter, 1993]. In order to illustrate some of the issues touched upon here, the following sections look at two example languages, Paralation Lisp and data-parallel Haskell. A number of other equally important examples exist, including NESL [Blelloch et al., 1993] and Sisal [Skedzielewski, 1991; Oldehoeft and Cann, 1988].

Paralation Lisp

A paralation [Sabot, 1988] is a collection of fields, with each field containing an index which identifies the *site*, or location, of the field (the index does not have to be unique.) The index-to-site relationship is, by default, arbitrary, but the programmer can enforce a particular *shape* on the paralation, including rings and grids. There are four levels of locality defined by the model: the elements in a single field are guaranteed to be near (*elementwise locality*) – this applies even if one of the fields is itself a paralation (*inherited locality*); the next closest items are fields from the same paralation with similar indices (*shape locality*); then comes any fields within the same paralation; and finally, fields from different paralations are the most distant. The degree of synchronisation required is minimal, as any function which is sensitive to the order of evaluation is defined to be invalid under the model [Sabot, 1988, pages 19–21].

Ignoring the underlying computational model, Paralation Lisp only requires three basic operations (excluding creation and testing for equality), `elwise`, `match`, and `move`, as detailed in table 2.2. In the following examples, a field is represented as $\mathit{value}_{\mathit{index}}$ rather

`elwise` *paralations function*	concurrently applies *function* to each set of fields with matching indices taken from the paralations. In its simplest form, `elwise` is equivalent to the map operation
`match` *paralation*$_1$ *paralation*$_2$	creates a partial, multi-valued index mapping based on the comparisons of all the elements of the two paralations
`move` *paralation mapping default combine*	returns a paralation whose contents are generated by applying the *mapping* to the specified paralation. The *default* and *combine* operators handle the special cases of zero or many elements being mapped to one location

Table 2.2: Paralation Lisp's data-parallel constructs

than the more usual (*index*, *value*) presentation:

$$\begin{aligned}
\texttt{elwise}\ [1_0, 3_1, 6_2]\ (1+) &= [2_0, 4_1, 7_2] \\
\texttt{elwise}\ [1_0, 2_1, 9_2]\ [1_0, 3_1, 6_2]\ (+) &= [2_0, 5_1, 15_2] \\
\texttt{match}\ [4_0, 7_1, 3_2]\ [3_0, 4_1, 5_2, 4_3] &= \{0 \mapsto 1, 0 \mapsto 3, 2 \mapsto 0\} \\
\texttt{move}\ [5_0, 9_1, 12_2]\ \{0 \mapsto 2, 1 \mapsto 0, 1 \mapsto 2\}\ \delta\ \oplus &= [9_0, \delta_1, (5 \oplus 9)_2]
\end{aligned}$$

Data-parallel Haskell

A POD [Hill, 1994, page 18] is a collection of index/value pairs, where the index element of each pair is unique within the POD. The main operation on the data is a restricted form of list comprehension [Hudak et al., 1992, page 16], with the map operation being defined as follows:

$$map_{pod}\ f\ pod = [(index, f\ x) \mid (index, x) \leftarrow pod]$$

Multi-POD comprehensions are similar, but, to make it clear where the index set should be taken from, all secondary PODs are introduced via the '$\Leftarrow$' operator:

$$\begin{aligned}
add_{pod}\ pod_1\ pod_2 &= [(index, x + y) \mid (index, x) \leftarrow pod_1, (index, y) \Leftarrow pod_2] \\
fetch_{pod}\ function\ pod &= [(index, y) \mid (index, x) \leftarrow pod, (function\ index, y) \Leftarrow pod] \\
send_{pod}\ function\ pod &= [(function\ index, x) \mid (index, x) \leftarrow pod]
\end{aligned}$$

The functions $fetch_{pod}$ and $send_{pod}$ demonstrate how the comprehensions can concisely specify arbitrary communication patterns, and as such are similar to a combined `match` and `move` operation under Paralation Lisp. In line with its pedigree, data-parallel Haskell does not provide any mechanism to specify a PODs *shape*, and synchronisation is automatically handled by the run-time system.

2.4.3 Skeletal parallelism

The *skeleton* paradigm was so named by Cole [1989], who described the use of a set of fixed algorithmic templates, into which problem-specific routines can be slotted. Typical skeletons include divide-and-conquer (and it's small scale equivalent, the processor farm), pipelines, loops and trees [Burkhart, Korn, Gutzwiller, Ohnacker and Waser, 1993], with each skeleton having several associated components [Darlington et al., 1993]: a *declarative meaning*, which is the programmer's primary reference when developing skeletal algorithms; one or more *implementation templates*, for efficiently performing the required computation; *performance models* to predict the run-time costs of a particular instantiation of a skeleton; and a set of *transformation rules*, which, in combination with the cost model, allow a program to be optimised for a particular architecture. These components are examined in sections 2.4.3 to 2.4.3,

The distinction between data parallelism and skeleton-oriented programming is unclear, as the former could be considered a sub-paradigm of the latter.

Declarative meaning

A skeleton is typically represented by its functional-language definition, with this model serving two main purposes: it acts as the primary reference for the programmer, and it allows the programs to be tested on a sequential architecture. Some example definitions are given below:

Haskell

```
pipe :: [a -> a] -> (a -> a)
pipe [f]    = f
pipe (f:fs) = f . (pipe fs)

divide_and_conquer :: (a -> Bool) -> (a -> b) -> (a -> [a]) -> ([b] -> b) -> a
                                                                            -> b
divide_and_conquer is_trivial solve split combine values
  | is_trivial values = solve values
  | otherwise = combine [divide_and_conquer is_trivial solve split combine
                                         part | part  <-  split values]
```

Implementation templates

The actual implementation of the model deals with all the low-level issues such as data placement, synchronisation and scheduling. In order to cater for different architectures, or even slight variations within a particular configuration, a number of implementation templates will be necessary to achieve architecture independence. The specification of a template is usually given in terms of the component processes and the interconnection network used. For example, figure 2.2 shows the model for the P^3L dedicated-farm skeleton [Pelagatti, 1993, pages 92, 97, and 139].

Performance models

In order to make resource-allocation decisions it is essential to have an accurate performance model of both the parallel and sequential parts of the program. There are a number of factors that must be considered when developing the model, including: the available resources, such as the number of processors; the overheads associated with the implementation of the skeleton; the performance of the problem-specific code; the volume of data to be produced/consumed; and so on. Obviously some of these parameters will have to be estimated, either via abstract analysis or run-time profiling. As an example, Bratvold

```
M ⇒ $i = 0$
    while input $(f, x)$
        select worker
        send worker $(f, x, i)$
        increase $i$
W ⇒ input $(f, x, i)$
    result $= f\ x$
    send collector (result, $i$)
C ⇒ ...
```

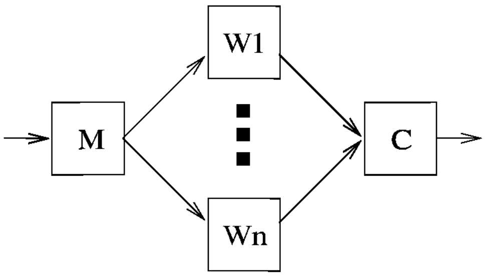


Figure 2.2: Operational template of P^3L's dedicated-farm skeleton

[1994, pages 105–106] uses the following equations to characterise the **pipeline** skeleton:

1. $L = \sum_{1 \leq i \leq s}(L_i + c_i)$
2. $T_C = (max_{1 \leq i \leq s}\ T_{Ci}) + L - L_{max}$

L	total latency of the pipeline
s	number of stages
L_i	latency of stage i
C_i	communication costs of sending a value from stage i to stage $i+1$
T_C	completion time
T_{Ci}	completion time for stage i
L_{max}	largest latency

Typical applications of cost models include discriminating between sequential and parallel implementations [Danelutto, Pelagatti and others, 1992, section 4.1], and deciding when to stop unfolding a divide-and-conquer problem [Darlington et al., 1993, section 4].

Transformation rules

Transformation rules declare two expressions or operational templates to be semantically equivalent, and, using a cost calculus as a guide, allow a transformation system to optimise a program with respect to a particular architecture [Pelagatti, 1993]. In addition, a precondition may have to be satisfied before the rule can be applied. The following rules are taken from Bratvold [1994, appendix B] and Darlington et al. [1993, section 5]:

$$\begin{array}{rcl}
\texttt{map}\ f\ (\texttt{map}\ g\ l) & \Longleftrightarrow & \texttt{map}\ (f\ .\ g)\ l \\
\texttt{map}\ (\texttt{divide\_and\_conquer}\ t\ s\ d\ c) & \Longrightarrow & \texttt{pipe}\ (\texttt{rept}\ q\ (\texttt{map}'\ n\ c))\ .\ \texttt{map}\ s\ . \\
 & & \texttt{pipe}\ (\texttt{rept}\ q\ (\texttt{foldr1}\ (+\!\!+)\ .\ \texttt{map}\ d))
\end{array}$$

Notice that the second transform is unidirectional, and uses an architecture-specific constant, q, which encodes the optimal depth of the pipeline. Transformations not only apply to program constructs, but to the implementation templates as well.

2.4.4 Low-level annotations

Unlike the other approaches to parallelism, low-level languages rely on the programmer to identify and control all aspects of the parallel program. This can lead to improved performance, but at the price of increased programmer effort and reduced portability.

Traditionally, annotations have been used to control the following operational properties of functional programs:

thread identification threads are the basic unit of work of a system, with each responsible for evaluating a single expression.

degree of evaluation by default, a thread will reduce an expression to head-normal form, which may not entail sufficient work to justify the overhead of thread creation.

thread and data placement the thread-to-processor mapping and data distribution encode load balancing and locality information.

order of evaluation scheduling annotations provide fine control over the sequence of computation, and can be used to augment, or override, the default strategy.

Thread identification

Probably the best-known annotation in parallel functional programming is the *spark* construct [Clack and Peyton Jones, 1986] – most commonly manifesting as either the "!" character or the *par* combinator. This indicates that the decorated expression *can* be evaluated (usually to normal form) in parallel with the current computation:

Haskell

```
dsum low high | (high == low) = high
              | otherwise     = (dsum    low      middle) + {! worth_it}
                                (dsum (middle + 1)  high)
  where middle   = (high + low) 'div' 2
        worth_it = (high - low)   >   50
```

where {!exp_{cond}} [Peyton Jones, 1989, page 181] is syntactic sugar for *let* x = *exp in if* exp_{cond} *then* x{!} *else* x. Another variation on the theme is Mattson Jr.'s speculative spark, {-# PCT *potential* #-} [1993a, page 66], where *potential* is an estimate of the probability that the expression will be needed.

Degree of evaluation

In order to justify the overhead of thread creation, it may be necessary to increase the work associated with an expression. Typically, this is done by overriding the default reduction strategy and evaluating to, for example, *irreducible normal form* [Kewley and Glynn, 1990, page 330]. In the presence of constructors, there are a wide spectrum of possible evaluation orderings [Burn, 1991, page 114], some of which may generate further threads of computation: (the following example uses the STG′ language from chapter 4 as standard Haskell does not support the necessary operations)

STG′ code

```
force_tree  = [] \r [tree] -> force_tree' tree tree ;
force_tree' = [] \r [tree original_tree]
  -> case tree of {Branch tree1 tree2 -> letpar tree1' = force_tree tree1 in
                                         letpar tree2' = force_tree tree2 in
                                         original_tree ;
                   Leaf   a           -> original_tree ; };
```

These transformers are either hand coded or automatically generated – Mirani and Hudak [1995, section2] support both approaches by using Haskell's class system to provide default definitions for any missing instances.

NeighbourP n proc_id	returns the ID of the *n*th neighbour of the specified processor
ChannelP var	returns the ID of the processor on which the argument variable is stored
ITOP integer	converts an integer into a processor ID, allowing user defined mapping functions to be written
CurrentP	returns the ID of the current processor
RandomP	generates a random processor ID

Table 2.3: Concurrent Clean's topology functions

Thread and data placement

The simplest example of process mapping is provided by Concurrent Clean's *Self* and *Par* annotations [Nöcker, Smetsers, Plasmeijer and van Eekelen, 1991] – threads spawned as a result of the former will be evaluated locally, and cannot be migrated to other processors, while those identified by the latter will probably be run on a remote processor. Within the same language, finer control is provided by the *P AT* exp_{PROCID} directive, where the expression is of primitive type *PROCID*. The primitive operations for manipulating such expressions are shown in table 2.3. In order to make use of the *ITOP* function, the mapping between integers and processor identifiers must be defined for each target architecture. To provide support for the various annotations, Concurrent Clean extends the type system so that it includes process types [Plasmeijer and van Eekelen, 1993b].

Para-functional Haskell [Hudak, 1991, section 5.3.3, pages 171–175] provides similar facilities, but for placing data rather than tasks. Also, by using an *operating system monad* to structure access to them, the inherent non-determinism is restricted to a purely operational level [Mirani and Hudak, 1995, section 4].

Caliban [Kelly, 1989] uses the *moreover* clause to both identify the threads and specify the required topology. This takes as its argument a conjunction of assertions, where an assertion is either an *arc* statement, or a *network-forming expression*. The statement *arc a b* indicates that process *a* derives its input from the output of process *b*, and that it is safe to run both processes concurrently, e.g.

```
main = (f . g . h) d
  where f = map ((+) 2), g = map ((*) 3), h = map sqrt, d = from 1
  moreover (arc @f @g) /\ (arc @g @h) /\ (arc @h d)
```

By using higher-order functions to manipulate *arc* definitions, a *network-forming expression* can concisely define a complex process structures. This is illustrated by the following definition of the *pipeline* function:

```
pipeline :: [a -> a] -> a -> a
pipeline fs x = (fold (.) id fs) x
  moreover (chain arc (map (@) fs)) /\ (arc @(last fs)  x)

chain :: (Bool -> Bool -> Bool) -> [(a -> b)] -> Bool
chain relation   [f]     = True
chain relation (f : fs) = (relation f1 (head fs)) /\ (chain  relation   fs)
```

Order of evaluation

In place of the usual *spark* and evaluation-override annotations, para-functional Haskell provides *schedule* expressions [Mirani and Hudak, 1995, section 2]:

> "Schedules define partial orders on *events* of which there are two kinds for every expression: (1) a *demand* for the expression's evaluation, and (2) a *wait* for the completion of the expression's evaluation. Many demands may be made for an expression's evaluation, but only the first one will have any effect."

A schedule, therefore, consists of either an event, or the concatenation or concurrence of two other schedules, denoted by $s_1 \; . \; s_2$ and $s_1 \parallel s_2$, respectively. The operational effect of these operators is similar to the traditional *seq* and *par* combinators. For example, consider the following function applications:

1) $f \; a \; b$ `schedule` (`demand` a $\parallel$ `demand` b $\parallel$ `demand` f)
2) $f \; a \; b$ `schedule` ((`demand` a . `wait` a) $\parallel$ (`demand` b . `wait` b)) . `demand` f

The first expression reduces both the function application and the arguments in parallel, while the second one concurrently evaluates the two arguments, and proceeds with the application only after both threads have completed.

2.5 Prototyping parallel functional languages

A would-be designer of a parallel functional system is faced with three major obstacles:

1. due to the large number of dimensions involved and to the lack of a common benchmarking system, it is extremely difficult determine which components are central to the performance of the system.

2. having selected and integrated the components, the cost of developing an efficient implementation for just one platform is considerable.

3. once the base implementation is complete, experimentation with any but the most trivial of subsystems may require significant effort.

What is needed is a system to rapidly develop and test ideas before committing to a full-scale implementation. Indeed, Hiromoto [1994, section 5] argues for a:

> "Incremental, cyclic and comparative approach in the evaluation of [parallel functional] languages, compilers and machine architectures."

The traditional software-engineering solution to this problem is the development of a prototype [Balzer et al., 1988, page 8]:

> "Prototyping is the process of constructing software for the purpose of obtaining information about the adequacy and appropriateness of the designers' conception for a software product...
>
> ...a prototype is distinguished from a production system by typically being more quickly developed, more readily adapted, less efficient and/or complete, and more easily instrumented and monitored."

Chapter 3 outlines a possible prototyping framework, while the remainder of this section examines the relevant literature.

2.5.1 The problem with performance comparisons

Jain [1991] motivates the need for performance evaluations as follows:

> "A performance evaluation is required when a computer designer wants to compare a number of alternative designs and find the best design... Even if there are no alternatives, performance evaluation of the current system helps in determining how well it is performing and if any improvements need to be made."

However, when Langendoen [1993, table 3.2, page 57] attempted to compare the performance of a number of parallel implementations, the following problems were encountered:

> "Unfortunately, few results are actually reported for each machine, and, worse, different algorithms have been used to solve the same problem. Only the notorious `nfib` program, a one-liner to compute the number of function calls per second, has been coded in similar style and measured on most parallel reduction machines."

Furthermore, Kaser et al. [1992, table 2, page 342] are probably the only others to attempt to directly compare different implementations (in this case, EQUALS, the $\langle v, G \rangle$-machine, and, GAML). They encountered similar problems: only two of the systems ran on the same architecture, and one of these did not support garbage collection (potentially 30% of the total run time). In the end, they resorted to comparing relative speedups.

Are these experiences surprising? Tables 2.4 and 2.5 summarise a number of existing implementations, including the test programs used to generate the performance results. Even disregarding the differences in the architecture, evaluation strategy, and source of parallelism, it is fairly clear that only a minority of systems have run even similar tests (fewer still have also included the source code and provided actual timings, rather than relative speedups.) Even in a purely sequential context, Partain [1993, section 1.1] is highly critical of the current state of performance evaluation:

> "The quantitative measurement of systems for lazy functional programming is a near-scandalous subject. Dancing behind a thin veil of disclaimers, researchers in the field can still be found quoting *nfibs/sec* (or something equally egregious), as if this refers to anything remotely interesting."

2.5.2 Is a standard benchmark suite the solution?

Consider, for example, PARKBENCH (PARallel Kernels and BENCHmarks [Hockney et al., 1993]), which has the following objectives:

1. to establish a comprehensive set of parallel benchmarks that is generally accepted by both users and vendors of parallel systems.

2. to provide a focus for parallel benchmark activities and avoid unnecessary duplication of effort and proliferation of benchmarks.

3. to set standards for benchmarking methodology and result-reporting together with a control database/repository for both the benchmarks and the results.

4. to make the benchmarks and results freely available in the public domain.

system	source	strict	reference
Alfalfa	implicit	yes	[Goldberg and Hudak, 1987]
Graphinators	implicit	no	[Hudak and Mohr, 1988]
BBN ML	implicit	no	[George, 1989]
HDG-machine	implicit	no	[Kingdon et al., 1991]
EQUALS	implicit	no	[Kaser et al., 1992]
SISAL	data-parallel	yes	[Oldehoeft and Cann, 1988]
NESL	data-parallel	yes	[Blelloch et al., 1993]
DP Haskell	data-parallel	no	[Hill, 1994]
Gamma	skeletal	yes	[Kuchen and Gladitz, 1992]
WYBERT	skeletal	no	[Langendoen, 1993]
Skeletal ML	skeletal	yes	[Bratvold, 1994]
$\langle v, G\rangle$-machine	low-level	no	[Augustsson and Johnsson, 1989]
GAML	low-level	no	[Maranget, 1991]
PAM	low-level	no	[Loogen et al., 1991]
BBN Haskell	low-level	no	[Mattson Jr., 1993a]
STAR:DUST	low-level	no	[Ostheimer, 1993]
pD	low-level	yes	[Schreiner, 1994]
GUM Haskell	low-level	no	[Trinder et al., 1996]

Table 2.4: Comparing implementations of parallel functional languages – language issues

Other C/Fortran-centric parallel suites include: SPLASH [Singh, Weber and Gupta, 1992], Genesis [Addison et al., 1993], and the NAS benchmarks [Bailey, 94].

With regards to functional programming, the `nofib` [Partain, 1993] suite (see appendix C) is probably the first attempt at providing a standard set of benchmark programs (a large subset of the collection had already been used by Hartel and Langendoen [1993].) The "pseudoknot" benchmark, based on a single float-intensive program, is also worthy of note, even if only for the unprecedented scale of collaboration achieved. However, ignoring the inherent problems of benchmarking [Bailey, 1991; Jain, 1991], there are a number of additional problems posed when moving to a parallel environment:

- there is no standard syntax for expressing parallelism. Indeed, even the general paradigm is still a matter for debate.
- each implementation embodies a large number of (potentially arbitrary) design decisions – isolating the effect of each would be very difficult, if not impossible.
- each architecture has different communication properties, and normalising the results would, again, be difficult.
- there is no consensus as to what metrics would be useful.

In summary, an implementation, with respect to a benchmark suite, appears to be a monolithic black box. This not only limits the useful experiments that can be devised, but also requires that a complete and optimised implementation exists. The problem is further compounded by the number of variables – consider, for instance, tables 2.4 and 2.5 – not least of which is the parallel architecture itself. It is therefore unlikely that any

	arch.	P	benchmarks	code	times
Graphinators	SIMD	8K	sum, matmult	yes	no
NESL	SIMD	16K	linefit, median, matmult	yes	yes
DP Haskell	SIMD	1K	map	yes	yes
SISAL	GMSV	10	sieve, simple, kernel batcher-sort, psphot	some	yes
$\langle v, G\rangle$-machine	GMSV	16	nfib 30, queens 10, euler	yes	no
BBN ML	GMSV	16	nfib 20, queens 8, sieve tak 18 12 6	yes	yes
GAML	GMSV	8	nfib 30, queens 10, euler sieve	yes	yes
Gamma	GMSV	6	mergesort, minimum	no	yes
EQUALS	GMSV	6	nfib 30, queens 10, euler sieve, matmult, qsort	no	yes
BBN Haskell	GMSV	122	euler, sieve, primes nfib', queens', tautology	yes	yes
WYBERT	GMSV	4	nfib, queens, det, wang puzzle, wave, comp-lab	some	yes
pD	GMSV	20	linear, resultants	yes	yes
GUM Haskell	GMSV	6	fac, loadtest, bulktest	yes	no
Alfalfa	DMMP	36	queens 6	no	yes
HDG-machine	DMMP	4	nfib 20, queens 6 tak 18 12 6	yes	yes
PAM	DMMP	12	nfib 24, matmult, towers map, fold, one, qsort	yes	yes
Skeletal ML	DMMP	34	ray, match, area	yes	yes
STAR:DUST	DMMP	24	nfib, qsort	some	yes
GUM Haskell	DMMP	8	fac, loadtest, bulktest	yes	no

Table 2.5: Comparing implementations of parallel functional languages – benchmarks. The *code* field indicates whether the source code of the benchmark programs was supplied, while the *time* column differentiates between systems that only supply speedup figures and those that provide the total elapsed times.

meaningful comparison or conclusions can be made, based on ad-hoc performance data (consider again table 2.5).

2.5.3 Existing approaches to developing functional implementations

The Haskell approach

One solution to the problems outlined previously, is to standardise one or more components of the language, compiler, and, architecture triple. For example, the Haskell language was designed to "reduce unnecessary diversity in functional programming languages" [Hudak et al., 1992, page iv]. Also, with regards to compiler implementation, GHC aims "to provide a modular foundation that other researchers can extend and develop" [Peyton Jones et al., 1993, section 2]. Both of these ideas have already been adopted by a section of the non-strict parallel functional community – pH [Nikhil et al., 1995, section 1, page 1], a Haskell derivative extended to include explicit parallelism, has as one of its goals:

> "To share infrastructure (compilers, systems, application programs), and to facilitate interesting research topics, such as comparing lazy evaluation *vs.* lenient evaluation..."

This is certainly a move in the right direction. However, by necessity, these compilers are written primarily for speed and efficiency, possibly at the expense of clarity – based on personal experience, this is certainly true of GHC! Moreover, the system will be sufficiently complex that familiarisation and development will take a significant amount of time.

Simulating multiprocessor architectures for compiled graph reduction

Before building the GRIP multi-processor [Peyton Jones, Clack, Salkild and Hardie, 1987], Deschner [1990] developed a "highly flexible simulation system" to explore task partitioning, memory usage, scheduling, topology, and run times. The simulator took as input a precedence graph and a description of the hardware configuration. The former is automatically generated by tracing the sequential execution of the test program, while the latter comprises: the number of processors, the task-pool size, the partitioning and scheduling policies, and the costs associated with some basic operations.

Hammond and Peyton Jones [1992, section 5.2], when analysing the performance of the final hardware implementation, acknowledge the accuracy of the simulator:

> "Somewhat surprisingly, in the absence of throttling, the choice of FIFO or LIFO [scheduling] strategy has at most a marginal impact on GRIP. We first realised this as a result of Deschner's simulation experiments, and then verified it on GRIP..."

An executable specification of the HDG machine

The HDG machine [Kingdon, Lester and Burn, 1991] was both specified and tested as a Miranda script. The authors note that using a functional language enabled them to write the simulator more quickly, debugging was easier, and the resulting definitions bore a strong resemblance to the traditional state-transition model of abstract machines. Indeed, Burn [1989, section 6, page 391] concludes that:

> "This has turned out to be such a powerful technique that we would highly recommend it to other machine designers."

Ginger

Joy and Axford [1992] describe the Ginger language, "which sits somewhere between the low-level FLIC [Functional Language Intermediate Code] and high-level Miranda or Haskell." The interpreter is combinator based, supports both strict and non-strict evaluation, and provides explicit parallelism via placed sparks and data-parallel lists [Axford and Joy, 1991]. A simulator is used to "facilitate research and teaching into parallelism," and this is capable of modelling both shared- and distributed-memory machines. No details of the simulation parameters are provided.

Simulating shared-memory graph reduction

Bennett [1993] uses simulation to explore the behaviour of a parallel functional system on shared-memory machines – primarily focusing on the caching mechanism. The simulator enables both theoretical and existing configurations to be explored, something that would have been impossible if using a design-build approach. The results lead to the design of a scalable cache mechanism which takes advantages of the memory reference characteristics of parallel functional programs.

A graphical winnowing system for Haskell

Hammond, Loidl and Partridge [1995a] use simulation to explore the impact of language and implementation on task granuality. The simulator is based on GHC, and models a distributed-memory machine. A visualisation tool helps to analyse the results, which have been used to uncover a previously unknown relationship between a program's run time and its heap-granuality profile (a histogram of the memory used by each thread of execution) [Hammond, Loidl and Partridge, 1995b]. In addition, and rather unusually, they also state that the simulation has confirmed the experimental results of a real system. However, they do point out the main problem with simulation:

> "There is, of course, a danger that the design of the simulator may obscure real artifacts or introduce false one."

2.6 Summary

Non-strict higher-order functional programming languages are elegant, concise, mathematically sound and contain few environment-specific features. Considering that sequential compiler technology has recently begun to compare with that of their imperative counterparts, they then become obvious candidates for harnessing high-performance parallel architectures. The validity of this approach has been established by a number of experimental compilers. However, a would-be designer of a parallel functional system is faced with three major obstacles:

1. due to the large number of dimensions involved and to the lack of a common benchmarking system, it is extremely difficult determine which components are central to the performance of the system.

2. having selected and integrated the components, the cost of developing an efficient implementation for just one platform is considerable.

3. once the base implementation is complete, experimentation with any but the most trivial of subsystems may require significant effort.

What is needed is a system to rapidly develop and test ideas before committing to a full-scale implementation.

Chapter 3

A framework for prototyping parallel functional intermediate languages

3.1 Introduction

This chapter describes an approach to the design of an explicitly parallel intermediate language for use during the compilation of non-strict higher-order functional programming languages. The framework is based upon the development of both a denotational and operational model for the intermediate language, which are then used to produce specifications for the parser and code generator. Haskell animations of these components aid with both debugging and informal validation. (For a more detailed and example-driven description of the prototyping system see [Ben-Dyke and Axford, 1995], which is reproduced in appendix A.)

3.2 A three-phase compilation system

As illustrated by figure 3.1, the prototyping framework is modelled on a traditional three-phase[1] compilation system [Santos, 1995, figure 2.1, page 6]. The phases are as follows:

the source language typically a Haskell-like non-strict functional programming language supporting higher-order functions and abstract data types. Parallelism will either be explicitly specified, as is the case with para-functional Haskell and Caliban, or abstract-analysis techniques will be employed to automatically detect the potential parallelism (see section 2.4). The *translation* rules convert from the source language to the intermediate representation.

the intermediate representation the sequential STG language [Peyton Jones, 1992] is used for the purpose of this study. As well as converting to and from the intermediate language via the *translation* and *code-generation* rules, the *optimisation* rules convert between equivalent language terms, with the aim of improving efficiency.

[1] Both WYBERT [Langendoen, 1993, figure 5.1, page 96] and Clean [Plasmeijer and van Eekelen, 1993a, figure 8.1, page 253] have four phases, but such systems can be considered as comprising two distinct compilation processes.

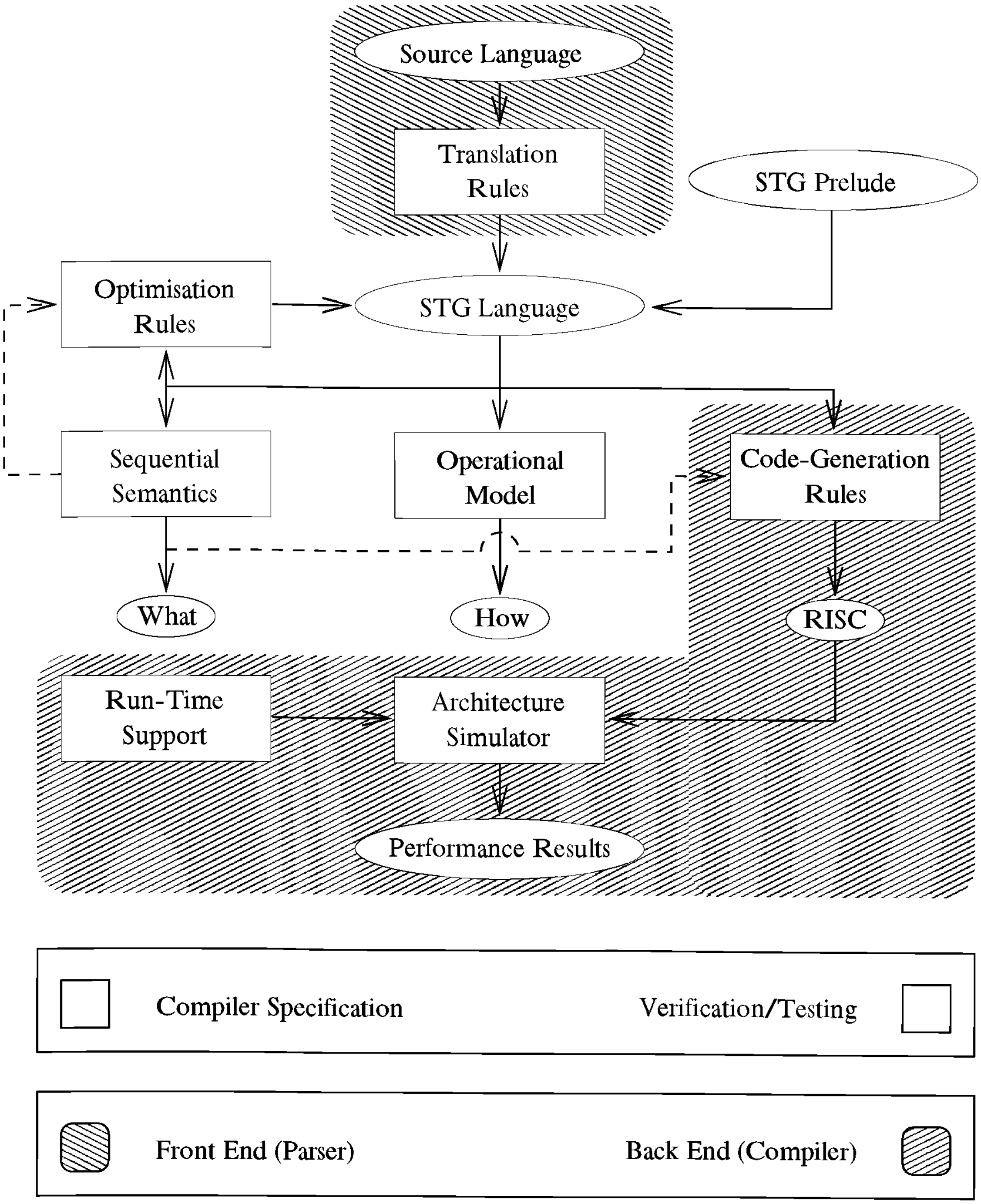


Figure 3.1: An overview of the prototyping framework

the target language the Alpha AXP instruction set [DEC, 1992] is the primary interface between the compiler and the architecture simulator. Section 8.2 discusses the reasons for selecting this over the high-level language C [Kernighan and Ritchie, 1978]. The *code-generation* rules convert from the STG representation into the RISC format.

3.3 Translation, optimisation, and code generation

The three rule sets shown in figure 3.1 – translation, optimisation, and code generation – in combination with the run-time support, serve as the specification of the compilation system. These can therefore be considered as the final outputs of the prototyping system:

translation rules will typically serve one of two possible roles: de-sugaring, i.e. the removal of any syntactic construct added solely for the purpose of improving programmability; and explicitly identifying the parallelism inherent in the source program. While the lexing and parsing of high-level languages is well understood [Watson, 1989; Peyton Jones, 1987], automatic parallelisation [Jones and Hudak, 1993; Burn, 1991] is still a major research topic – therefore, the front end of the compiler will not be discussed in this thesis.

optimisation rules form an important part of most modern compilers, and this is especially true of functional systems [Gill, 1992; Beemster, 1994]. Due to the similarity between the Core [Santos, 1995, section 2.2] and STG languages, all of the optimisation rules, heuristics, and algorithms presented by Santos [1995] have STG-language equivalents. However, in a parallel context, another major rule group is often required – these deal with architecture-specific optimisations, as illustrated by the skeletal transformations described in section 2.4.3.

compilation rules generate the low-level machine code, and therefore have to deal with such issues as register allocation [Fraser and Hanson, 1992; Boquist, 1995], heap representations [Shao and Appel, 1994], control flow [Bernstein, 1985], and stack frames [Douence and Fradet, 1995]. As noted by Shao and Appel [1995], it is important to take advantage of the information provided by the intermediate language's static semantics (see section 4.5):

> "Our measurement shows that a combination of several type-based optimisations reduces heap allocation by 36%; and improves the already-efficient code generated by the old non-type-based compiler by about 19%..."

(See chapter 8.)

the run-time system covers such sequential technology as garbage collection, optimised library routines, error handling. In addition, certain parallel components are also necessary, including termination-detection algorithms, the implementation of any skeletal sub-systems, and interfaces to the communication network. (See chapter 8.)

3.4 Structuring the design

While not forming part of the final output, the various semantics models of the three phases, once developed, are arguably the most important components of the framework. Henderson [1986, section 7, page 249] notes that:

> "A formal language can be effective as a tool for communication of designs on a larger scale and to suggest the way in which software design should proceed using formal methods."

the sequential semantics two descriptions are used, with the first being a denotational semantics [Stoy, 1977], which assigns values to programs. This provides the reference model against which the compilation rules can be tested and the translation rules validated (assuming that the source language also has a denotational semantics). Furthermore, as outlined in section 5.4, the development of the semantics forces the designer to concentrate on a number of important issues, including the order and degree of evaluation, non-determinism, and run-time errors.

The second description, a Hindley–Milner type-inference algorithm, restricts the set of valid language expressions. This simplifies the compilation rules (as well as enabling a number of advanced optimisations [Shao and Appel, 1995]), and does away with the need for run-time type checking.

the operational model based on a state-transition system, the operational model provides a concise high-level description of the intended behaviour of the compilation rules. This approach has been widely used to develop a number of abstract machines, including both Tim and the STG machine. Having constructed the model, and tested it against the simpler denotational description, it should be easier to develop the actual compilation rules.

the architecture simulator provides the framework for testing both the correctness and performance of the compilation rules. Note that any generated results are potentially inaccurate, and should therefore only be used to motivate design choices.

the STG-language prelude offers a source of examples and test cases. Appendix B.1 includes some typical prelude definitions.

In addition to driving the design process, the development of the (semi-formal) models mean that it is, in theory, possible to prove the correctness of the rules discussed in section 3.3. However, while important, this subject is beyond the scope of this thesis.

3.5 Animating the compiler

As the semantic descriptions are used to validate the compiler's front and back ends, it is important that there is a degree of confidence in the models themselves. One possible solution to this problem is suggested by Henderson [1986, section 7, page 249]

> "The executable prototype introduces a realistic element of validation of the design sufficiently early in the development process that there is some likelihood of eventual cost saving due to the early determination of design flaws."

Therefore, using the functional programming language Haskell [Hudak, Peyton Jones, Wadler and others, 1992], executable versions of the specifications are developed. The prescriptive approaches to the animation process are described in sections 4.3.5, 4.5.3, 4.7.4, and 4.8.10 – dealing with abstract syntax, Hindley–Milner type-inference rules, denotational semantics, and state-transition rules respectively.

3.6 Summary

Most parallel implementations of functional programming languages have at least one point of commonality: the use of an intermediate form. Typically, these abstract representations explicitly identify all parallel components but without the background noise of syntactic and (potentially arbitrary) implementation details. We suggest that that this is a good point at which to start to draw comparisons, and the problem now becomes one of isolating and testing the effect of a particular design feature. To this end, this chapter outlined a framework for rapidly prototyping such intermediate languages. Based on the traditional three-phase compiler model, the design process is driven by the development of semantic descriptions of the source, intermediate, and target language (and architecture). Executable versions of the specifications help to both debug and informally validate these models.

Chapter 4

The sequential STG$'$ language

4.1 Introduction

This chapter describes the STG$'$ language, a variant of the *Shared Term Graph (STG) language*, in terms of its abstract and concrete syntax, denotational semantics, and operational semantics. A Hindley–Milner style type-inference algorithm is also presented, which serves to restrict the language and produces information useful to a compilation system.

The chapter starts with a discussion of the utility of a sequential language in a parallel prototyping system in section 4.2, and the the abstract and concrete syntaxes are covered in sections 4.3 and 4.4. In section 4.5 the type-inference algorithm is presented, and the problem of how to record the resulting type annotations is addressed in section 4.6. The denotational and operational semantics of the language are then dealt with in sections 4.7 and 4.8, before the chapter is summarised in section 4.9.

4.2 Why use a sequential language?

Gelernter and Carriero [1992, page 97] state that a complete programming model consists of two orthogonal components, a computation model and a coordination model:

> "The computation model allows programmers to build a single computational activity: a single-threaded, step-at-a-time computation. The coordination model is the glue that binds separate activities into an ensemble."

It follows that when developing a coordination model it will be necessary to couple it with an existing (sequential) computational model. However, Gelernter and Carriero argue for the complete separation of these two components on the grounds of portability and heterogeneity. In principle, they are correct, but in practice little is lost by creating a mixed-model language, and it is likely that there will be a performance gain due to the close coupling of the two.

Having decided that a computation model will be needed, the selection of the STG language over the other suitable candidates – Tim [Fairbairn and Wray, 1981; Chitnis, Satpathy and Oberoi, 1995], a continuation-passing system [Appel, 1992], functional quads [Traub, 1991], or the ABC machine [Plasmeijer and van Eekelen, 1993a] – has to be justified:

1. the STG language is "a very austere purely-functional language" [Peyton Jones, 1992, section 4] making the conversion from a high-level functional language to the

intermediate form particularly simple. In addition, STG language expressions are concise, yet easy to read.

2. the STG-machine [Peyton Jones and Salkild, 1989] provides the language with an operational reading, the efficiency of which has been demonstrated by the performance of the Glasgow Haskell compiler in a recent benchmark test [Hartel, 1994].
3. there exists of a large body of literature relating to the STG language, covering a wide range of topics, from semantics [Peyton Jones and Launchbury, 1991] to parallel implementation [Hill, 1993].
4. the Glasgow Haskell compiler is capable of dumping the STG language equivalent of a Haskell program, via the `-ddump-stg` command-line switch, providing a ready supply of example code.

4.3 Abstract syntax

The abstract syntax[1] of the STG′ language is given in figure 4.1, with the exception of identifiers, which are discussed in section 4.3.1 (appendix B presents a number of example STG′ programs.) The significant differences between the STG′ and STG languages are:

introduction of algebraic data-type declarations as the operational semantics made no use of the Haskell-style sum-of-products declarations, their definition was omitted from the original STG report. But, in order to develop both type-inference and compilation rules, such information is vital.

removal of named defaults from case expressions as noted by Peyton Jones [1992, section 5, rule 8], the presence of named defaults complicates the operational semantics, and similar difficulties arose when developing the type-inference and compilation rules

introduction of unboxed and strict let expressions the `let#` and `letstrict` expressions compensate for the removal of the named defaults from the literal and algebraic `case` expressions respectively.

The minor changes include the removal of a number of extraneous symbols, some renaming, and the provision of the *bind* production. These have the net effect of simplifying the development and presentation of the syntax-driven algorithms described within this thesis.

Even though the STG-machine is not considered until chapter 6, the operational reading of the language expressions is given in table 4.1. Note that evaluation necessitates the creation of one or more continuations to which the resulting constructor, literal or primitive expressions will return.

Algebraic data types are discussed in section 4.3.2, `case` expressions in section 4.3.3, and the new `letstrict` and `let#` expressions in section 4.3.4. As for the other production rules, these are as presented by Peyton Jones [1992], to whom the interested reader is referred. Finally, the problem of animating the abstract syntax is dealt with in section 4.3.5.

[1] The terminology used within this section is primarily based on that used by Watt [1991]

Program	$program$	$\longrightarrow$	$typedecls\ bindings$	
Type declarations	$typedecls$	$\longrightarrow$	$typedecl_1 \ldots typedecl_t$	$(t \geq 0)$
	$typedecl$	$\longrightarrow$	$\texttt{data}\ \chi\ \alpha_1 \ldots \alpha_v\ \texttt{=}\ condecls$	$(v \geq 0)$
	$condecls$	$\longrightarrow$	$condecl_1 \ldots condecl_n$	$(n \geq 1)$
	$condecl$	$\longrightarrow$	$cons\ \tau_1 \ldots \tau_f$	$(f \geq 0)$
Monotype	τ	$\longrightarrow$	$\pi \mid \nu$	
Boxed type	π	$\longrightarrow$	$\alpha \mid \tau_1 \rightarrow \tau_2 \mid \chi\ \pi_1 \ldots \pi_v$	
Unboxed type	ν	$\longrightarrow$	$\texttt{Int\#} \mid \ldots \mid \texttt{Float\#}$	
Bindings	$bindings$	$\longrightarrow$	$bind_1 \ldots bind_n$	$(n \geq 1)$
	$bind$	$\longrightarrow$	$var = lambda\_form$	
	$simplebind$	$\longrightarrow$	$var = exp$	
Lambda form	$lambda\_form$	$\longrightarrow$	$vars_{free}\ \pi\ vars_{args} \rightarrow exp$	
Update flag	π	$\longrightarrow$	$\texttt{u} \mid \texttt{r}$	
Expression	exp	$\longrightarrow$	$\texttt{let}\ bindings\ exp$	
		$\mid$	$\texttt{letrec}\ bindings\ exp$	
		$\mid$	$\texttt{let\#}\ simplebind\ exp$	
		$\mid$	$\texttt{letstrict}\ simplebind\ exp$	
		$\mid$	$\texttt{case}\ exp\ \texttt{of}\ alts\ [default]$	
		$\mid$	$var_{fun}\ atoms$	
		$\mid$	$cons\ atoms$	
		$\mid$	$primitive\ atoms$	
		$\mid$	$literal$	
Alternatives	$alts$	$\longrightarrow$	$lalt_1 \ldots lalt_n$	$(n \geq 1)$
		$\mid$	$aalt_1 \ldots aalt_n$	$(n \geq 1)$
	$lalt$	$\longrightarrow$	$literal \rightarrow exp$	
	$aalt$	$\longrightarrow$	$cons\ vars \rightarrow exp$	
Default	$default$	$\longrightarrow$	$\_ \rightarrow exp$	
Variables	$vars$	$\longrightarrow$	$var_1 \ldots var_n$	$(n \geq 0)$
Atoms	$atoms$	$\longrightarrow$	$atom_1 \ldots atom_n$	$(n \geq 0)$
	$atom$	$\longrightarrow$	$var \mid literal$	

Figure 4.1: Abstract syntax of the STG′ language

Construct	Operational reading
function application	tail call
`let(rec)` expression	heap allocation
`let#` expression	evaluation and register assignment
`letstrict` expression	evaluation and heap allocation
`case` expression	evaluation
constructor application	return to algebraic continuation
primitive application	return to continuation[a]
literal expression and literal-variable application	return to literal continuation

[a] Primitive functions will either return a literal or constructor value, depending upon the primitive in question.

Table 4.1: The operational reading of STG′ language expressions

4.3.1 Identifiers

While the exact format of the different types of identifiers is left unspecified, the naming scheme is in line with Haskell's policy [Hudak et al., 1992, pages 6–9]: variables, *var*, and type variables, α, are represented by identifiers beginning with lower-case letters; constructors, *cons*, and type constructors, χ, are either identifiers which start with a capital letter, or are a sequence of non-alphanumeric characters (`:`, `:+`, and `:=` are examples of this form); primitives, *primitive*, are similar to variables, except the identifier will end with a hash symbol; finally, literals, *literal*, are represented by the usual constants (integers, floating-point numbers, ASCII characters etc.). The main exception is the representation n-ary tuples, which are encoded as `Tup0`, `Tup2`, `Tup3` etc.

4.3.2 Algebraic data-type declarations

A data-type declaration[Hudak et al., 1992, pages 27–28] defines a new sum-of-products type, consisting of one or more constructors. The following example defines booleans, lists and trees:

STG′ code

```
data Bool   = True   | False ;

data List a = Nil    | Cons a (List a) ;

data Tree a = Leaf a | Branch (Tree a) (Tree a) ;
```

Using these declarations it is possible to define enumerated, recursive and (polymorphic) composite types [Bird and Wadler, 1988, pages 204–219].

4.3.3 Named defaults and case expressions

In the original STG language, algebraic `case` expressions containing named defaults served two distinct roles. Firstly, they provided a way of avoiding allocation of the result of the scrutinised expression whenever the result matched any of the non-default alternatives. For example, in the following example, `r` and `r'` compute identical values, but `r` will not create a closure for the value of `(f a)` if it matches the pattern `S x`:

STG code

```
r  = [] \r [a] -> case  f a of    { S x    -> g x ;
                                    t      -> h t };      {- named  default -}

r' = [] \r [a] -> let { result = [] \u [] -> f a ; } in

                  case  result of { S x    -> g x ;
                                    _      -> h result }; {- simple default -}
```

Secondly, they allow a value to be forced to head-normal form, presumably encoding the result of a strictness-analysis phase:

STG code

```
enumFromTo = [] \r [n m] -> case enumFrom n of  {

   n_to_inf -> let { predicate = ... } in             {- named  default -}

               takeWhile predicate n_to_inf };
```

One `case` expression may use the named default for both of these purposes. However, analysis of the STG representation of the `nofib` benchmark suite (see appendix C) has shown that named defaults are only ever used to encode strictness information. Therefore, in practice, there are just two distinct uses of the algebraic `case` expression: alternative selection based on the de-construction of the value of the scrutinised expression; and forced evaluation combined with heap allocation of the result. The second usage bears more resemblance to variable binding than to selection. Moreover, supporting both types of behaviour leads to complications in the operational semantics [Peyton Jones, 1992, section 5, rule 8], type-inference rules and compilation system. For example, it would be difficult to develop a concise type rule which rejected the following function definition:

STG code

```
seq = [] \r [x y] -> case x of {x' -> y x' };              {- named  default -}
```

For these reasons, named defaults were removed from algebraic `case` expressions, with the new `letstrict` expression taking over the role of strictness encoding (see the following section).

The use of named defaults in literal `case` expressions is slightly simpler, as unboxed values are never directly heap allocated. Based on analysis of the `nofib` benchmark suite, named defaults are always unaccompanied and serve to bind the result of a computation to a variable, as illustrated below:

STG code

```
... case minusInt# [x', y'] of
     {
     xy -> case plusInt#  [xy, 1#] of                   {- named  default -}
            {
            xy' -> ... expression using xy and xy' ...   {- named  default -}
            }
     }
```

To keep the `case` expression symmetrical, named defaults were also removed from the literal version (`let#` binds literal expressions to variables.)

4.3.4 Unboxed and strict let expressions

Having removed named defaults from both literal and algebraic `case` expressions, it became necessary to determine if any important functionality had been lost. The answer, in the case of literal defaults, was a definite yes – there was now no way to bind a temporary literal value to a variable (short of defining a new function whose arguments were the value plus the free variables of the remaining computation). To this end the `let#` expression was introduced, the use of which is illustrated below:

STG′ code

```
... let# xy  = minusInt# [x', y'] in
    let# xy' = plusInt#  [xy, 1#] in ... expression using xy and xy' ...
```

The right-hand side expression must evaluate to one of the primitive unboxed types.

With respect to algebraic defaults, the situation is less straightforward as it is still possible to achieve the same results through the use of an additional `let` expression (see the previous section for an example). Yet the named algebraic default accounted for approximately seven percent of the total dynamic bindings (i.e. any variable that is introduced by a `let(rec)` expression or named algebraic default) of the optimised `nofib` benchmark suite. The `letstrict` expression was therefore introduced, as shown here:

STG′ code

```
enumFromTo = [] \r [n m] -> letstrict n_to_inf  = enumFrom n in
                            let     { predicate = ...      } in
                            takeWhile predicate n_to_inf ;
```

The right-hand expression must evaluate to an algebraic data type (see section 4.5.1).

4.3.5 Animating the abstract syntax

Most of the algorithms and transition rules used during prototyping are syntax driven, so the animation of the abstract syntax is arguably the most important aspect of the entire system. Fortunately, using Haskell's algebraic data types and type synonyms, the task is a simple one.

For each group of production rules a new data type is created, and each production rule within the group becomes a constructor of the new type. So, for example, the unboxed-type production rules ($\nu \longrightarrow$ **Int#** | … | **Float#**) translate to:

Haskell

```
data UnboxedType = UnboxedInt | ... | UnboxedFloat
```

The choice of type and constructor names should reflect the group and individual production rules respectively, but some mangling may be necessary to arrive at a unique name (a restriction of the Haskell language).

In general, a constructor will have one argument type for each constituent non-terminal symbol, unless there are a variable number of the same symbol, in which case a *List* type is used. This is illustrated by the boxed-type rules ($\pi \longrightarrow \alpha \mid \tau_1 \rightarrow \tau_2 \mid \chi\ \pi_1 \ldots \pi_v$):

Haskell

```
data BoxedType   =  BoxedVar   TypeVariable
                 |  BoxedFun   MonoType      MonoType
                 |  BoxedCon   Constructor   [BoxedType]
```

In addition to the non-terminal symbols, extra arguments may be added to a constructor to facilitate parts of the prototyping system. A case in point is the *exp* group of rules, to each of which has been added an *ExpressionId* field, providing a unique key with which to look up expression-specific information (see section 4.6):

Haskell

```
data Expression  = Let   ExpressionId Bindings   Expression             | ...
                   Case  ExpressionId Expression Alts (Maybe Default) | ...
                   Value ExpressionId Literal
```

The previous example also illustrates the use of the `Maybe` type to represent optional non-terminal symbols, such as the *default* symbol in a `case` expression. The data declaration for this type is: `data Maybe a = Just a | Nothing`.

Variables, constructors and all other identifiers are represented using the `String` synonym.

4.4 Concrete syntax

It may seem strange that an intermediate language would ever make use of a concrete syntax, but such a representation is useful in two important situations: when reporting an error; and during testing, where parsing a textual description of an STG′ program is quicker, more convenient, and less prone to error than hand coding the Haskell representation. So saying, the exact details of the concrete syntax used are not important, and only the input and output routines are considered here. To simplify the parser, keywords, such as `let` and `of`, are not allowed to be used as variable names – a production compiler may well lift this restriction.

With regards to the conversion of the abstract syntax to text, the simplest solution would be the use of derived instances of the *Text* type class [Hudak et al., 1992, 147–148] for each of the production-rule data types. Unfortunately, the resulting output is awkward and does not match the format used by the parser. Hand coding is the only alternative:

Haskell

```
lambdaformShow (LambdaForm free_vars uflag args exp)
   =  "[" ++ free_vars' ++ "] " ++ uflag' ++ " [" ++ args' ++ "] ->" ++ exp'
  where
  free_vars' =  variablesShow free_vars
  uflag'     = updateflagShow uflag
  args'      =  variablesShow args
  exp'       = expressionShow exp
```

Developing a robust parser is more taxing, but most of the complexity can be avoided by using Happy [Gill and Marlow, 1993]:

> “Happy is a parser generator system for Haskell, similar to the tool ‘yacc’ for C. Like ‘yacc’, it takes a file containing an annotated BNF specification of a grammar and produces a Haskell module containing a parser for the grammar.”

As an example, here is the Happy specification of the *lambda_form* production rule (of the concrete syntax):

Happy

```
LambdaForm ::                     { LambdaForm }
LambdaForm  : '[' Arguments ']'  UpdateFlag '[' Arguments ']'
                  right_arrow    Expression  {  LambdaForm $2 $4 $6 $9 }
```

Notice the symmetry between this rule and the previous display routine.

4.5 Language restrictions, type inference and free variables

In this section, the problem of restricting STG′ language programs to ensure the validity of the STG machine is addressed. Section 4.5.1 enumerates the required restraints and illustrates the need for a type-inference system. The advantages and limitations of static typing are then outlined in section 4.5.2, while section 4.5.3 outlines a Hindley–Milner style type system for the STG′ language. Finally, an algorithm for generating the free-variable annotations of a binding ($var = lambda\_form$) is presented in section 4.5.4.

4.5.1 The STG language and the STG machine

To simplify the design of the STG machine, and thereby improve its efficiency, Peyton Jones [1992, section 4] and Peyton Jones and Launchbury [1991, section 7.1] explicitly placed the following (informal) restrictions on STG language programs:

1. global (top-level) bindings, and `let` and `letrec` expressions cannot bind a variable of unboxed type.

2. all constructors and primitives are saturated (have the correct number of arguments).

3. polymorphic functions cannot manipulate unboxed values.

Also, Beemster [1994] has shown that the STG machine cannot force the evaluation of a partial application due to its *aggressive take* (the method by which a function's arguments are fetched, as embodied by rules 17 and 17a in section 5.6 of the STG report). Essentially, the following expression will not terminate under the STG machine:

STG code

```
... let { f = [] \r [a b c] -> ... expression using a, b, and c ... ; }
    in case f x y of
        {
        f' -> ... expression using f' ...          {- named  default -}
        }
```

This leads to the following additional limitation:

4. **case** expressions can only scrutinise values whose type is either unboxed or algebraic.

(section 6.3.1 shows how this restriction may be removed.) Finally, three additional requirements are needed:

5. the top-level variable *main* is defined, and is bound to an expression of type *Dialogue*.

6. the operational decorations (i.e. update flags and free-variable information) are correct.

7. all of the patterns from a **case** expression's alternatives are unique i.e. only one alternative will ever be applicable for any given result.

Looking at the first five rules, it is clear that detailed type information is required if the validity of a program is to be verified. There are two possible sources for this data: *type-inference*, the attributes are automatically derived using a system of type rules; and *type annotations*, the abstract syntax is extended to include type information, thereby delegating responsibility for type inference to the previous stage (conversion from the source language to the intermediate form). The latter is the approach adopted by the Glasgow Haskell compiler, although the type information is recorded in a database similar to that described in section 4.6.

Whichever approach is taken, some form of type inference will be required. For the sake of generality, it was decided to frame this problem in the context of the STG$'$ language. The traditional approach to typing in a functional programming language is to use a Hindley–Milner style algorithm [Milner, 1978; Damas and Milner, 1982] – both ML and Haskell employ this technique and this is also the solution adopted for the STG$'$ language.

With regards to the sixth requirement, section 4.5.4 presents an algorithm for checking or generating the free-variable information, while section 4.2 of the STG report discusses the problem of setting the update flag.

4.5.2 The advantages of static typing

A language is said to be *statically typed* [Schmidt, 1994, page 6] if a type inference algorithm exists which can calculate the type attributes of a program without evaluating the program. As the algorithm presented in section 4.5.3 is purely syntax driven, the STG$'$ language is statically typed. The principle benefits of static typing are improved debugging, and an increase in the number of optimisations that can be performed by a compilation system [Shao and Appel, 1995; Hall, 1994; Gill and Peyton Jones, 1994]. It is the latter property that is of primary importance in the context of the prototyping system.

The primary drawback of adopting a static system is that it becomes difficult, if not impossible, to use the intermediate form as a target for dynamically-typed source languages. The majority of modern functional programming languages are statically typed [Hudak et al., 1992; Harper et al., 1989], so this becomes an acceptable limitation. Indeed, Cardelli and Wegner [1985, page 474] suggest that

> "In general, we should strive for strong typing and adopt static typing whenever possible."

4.5.3 Hindley–Milner type inference for the STG′ language

The type-inference system presented in this section is based on the work of Peyton Jones and Wadler [1992], which, in turn, is based on the Hindley–Milner algorithm [Milner, 1978; Damas and Milner, 1982]. Both ML and Haskell use variants of this algorithm. For both an overview of this technique and a discussion of alternative approaches [Reynolds, 1985; Schmidt, 1994; Cardelli and Wegner, 1985] are highly recommended.

Note that no attempt is made to relate the type algorithm to any of the other semantic descriptions, neither is it proved that the algorithm assigns the most general type to an expression.

Limitations of the inference rules

As a side effect of using the algorithm outlined in this section, the following additional language restrictions are imposed:

8. a variable bound by a `letrec` expression must have the same type for all occurrences in the right-hand sides of the bindings.

9. lambda-bound and pattern-defined variables must take the same type for all occurrences in the body of the function or algebraic alternative.

Such limitations are common, as typified by Haskell's *monomorphism restriction* [Hudak et al., 1992, pages 40–41]. A number of attempts to remove these restrictions [Kfoury, Tiuryn and Urzyczyn, 1993; Henglein, 1993] have met with limited success, as the problem is, in general, undecidable.

Terminology

The notation adopted here is based on that used by Peyton Jones and Wadler [1992] and is only briefly introduced here, as a full account is given in appendix D.

The abstract syntax of types, part of which was included in figure 4.1, is shown in figure 4.2. Following the usual conventions, function types of the form $\tau_1 \rightarrow (\tau_2 \rightarrow (\cdots \rightarrow \tau_n)\cdots)$ will be written as $\tau_1 \rightarrow \tau_2 \rightarrow \cdots \rightarrow \tau_n$, and brackets will only be used if one of the argument types is itself a function type.

An *environment* is a finite mapping, usually from identifiers to types, either explicitly constructed, e.g. $\{var_1 \mapsto \tau_1, \ldots, var_n \mapsto \tau_n\}$, or created by combining two existing environments. Two forms of merge operations are used: $env_1 \oplus env_2$, which is only defined if the domains are distinct; and $env_1 \overset{\rightarrow}{\oplus} env_2$, where an identifier will take its value from the second environment if it is defined by both. An identifier's value is retrieved by treating the mapping as a set of tuples and testing for membership i.e. $(id, value) \in env$.

Polytype	σ	$\longrightarrow$	$\forall\alpha_1 \ldots \alpha_n.\tau$	type signature
		$\mid$	τ	simple type
Monotype	τ	$\longrightarrow$	π	boxed type
		$\mid$	ν	unboxed type
Boxed type	π	$\longrightarrow$	α	type variable
		$\mid$	$\tau_1 \rightarrow \tau_2$	function type
		$\mid$	$\chi\ \pi_1 \ldots \pi_n$	parameterised data type
Unboxed type	ν	$\longrightarrow$	`Int#`	integer
		$\mid$	`Float#`	floating-point number
		$\mid$	`Char#`	character

Figure 4.2: Abstract syntax of types

Environment	Notation	Type
constructor environment	CE	$cons \mapsto (n, \sigma)$
primitive envrionment	PE	$primitive \mapsto (n, \sigma)$
general variable environment	GVE	$var \mapsto \sigma$
local variable environment	LVE	$var \mapsto \tau$
type-constructor environment	TCE	$\chi \mapsto (n_\alpha, n_{cons}, \langle cons_1, \ldots, cons_{n_\alpha} \rangle)$
total environment	TE	(CE, PE, GVE, LVE)

Table 4.2: Summary of the environments used during type inference

The environments used by the type rules are summarised in table 4.2, where: n is either the arity of a function or a constructor; n_α is the number of type variables needed to saturate an algebraic type; n_{cons} the number of constructors; and $\langle cons_1, \ldots, cons_{n_\alpha} \rangle$ the constructors themselves.

Algorithm overview

As all of the rules are included in appendix D, only a brief overview of the algorithm is given here.

initial rule the *PROGRAM* rule serves as the starting point of the algorithm, a simplified version of which is shown in figure 4.3. Notice how the primitive environment, *PE*, is passed as an argument to the inference algorithm, with figure 4.4 providing some example entries. The algorithm proceeds as follows:

1. generate the constructor environment, *CE*.
2. initialise the total environment, *TE*.
3. infer the type of the top-level definitions, treating the entire program as one large `letrec` expression.
4. ensure that the fourth restriction presented in section 4.5.1 is met.

$$PROGRAM\quad \frac{\begin{array}{ll}(1) & \overset{\text{typedecls}}{\vdash}\ \ typedecls : (TCE, CE)\\ (2) & TE = (CE, PE, \emptyset, \emptyset, \emptyset)\\ (3) & TE \overset{\text{recbinds}}{\vdash} bindings : GVE' \\ (4) & (main, Dialogue) \in GVE'\end{array}}{PE \overset{\text{program}}{\vdash} typedecls\ bindings : GVE'}$$

Figure 4.3: The simplified *PROGRAM* type rule

$$\begin{array}{rcl} negateInt\# & : & \texttt{Int\#} \rightarrow \texttt{Int\#}\\ plusInt\# & : & \texttt{Int\#} \rightarrow \texttt{Int\#} \rightarrow \texttt{Int\#}\\ lessthanInt\# & : & \texttt{Int\#} \rightarrow \texttt{Int\#} \rightarrow Bool \end{array}$$

Figure 4.4: Example type signatures of primitive functions

constructor-environment generation apart from checking that the definitions are well formed, the main purpose of the type-declaration rules is to produce the constructor environment. This is primarily achieved by the *CONDECL* rule, shown in figure 4.5, which:

1. verifies that each of the constructor's argument types are well formed.
2. using the function type as a convenient representation, generates a polytype description of the constructor.
3. generates a environment whose sole entry associates the constructor with the description from step 2.

bindings top-level definitions and `let(rec)` expressions are the only way to introduce variables with polymorphic type signatures, as illustrated by the *BINDS* rule shown in figure 4.6. In general, the type of the right-hand side is first inferred (step 1) and then generalised (step 2), and the resulting signature added to the general variable environment, *GVE* (step 3).

expressions this group of rules forms the heart of the algorithm, with each rule deriving the monotype associated with one of the expression constructs. As an example, the *CONS-EXP* rule, shown in figure 4.7, proceeds as follows:

$$CONDECL\quad \frac{\begin{array}{ll}(1) & TCE \overset{\text{monotype}}{\vdash} \tau_i \quad (0 \leq i \leq f)\\ (2) & \overset{\text{gen}}{\vdash} \tau_1 \rightarrow \cdots \rightarrow \tau_f \rightarrow \tau_\chi : \sigma\\ (3) & CE = \{cons \mapsto (\,f, \sigma)\}\end{array}}{TCE; \tau_\chi \overset{\text{condecl}}{\vdash} cons\ \tau_1 \ldots \tau_f : (CE, cons)}$$

Figure 4.5: The *CONDECL* type rule

$$BINDS\quad \frac{\begin{array}{ll}(1) & TE \overset{\text{bind}}{\vdash} bind_i : (var_i, \tau_i) \\ (2) & TE \overset{\text{gen}}{\vdash} \tau_i : \sigma_i \\ (3) & GVE = \oplus_{i \leq n}\{var_i \mapsto \sigma_i\}\end{array}}{TE \overset{\text{binds}}{\vdash} bind_1 \ldots bind_n : GVE}$$

Figure 4.6: The *BINDS* type rule

$$CONS\text{-}EXP\quad \frac{\begin{array}{ll}(1) & (cons, (n, \sigma)) \in CE \\ (2) & TE \overset{\text{spec}}{\vdash} \sigma : \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \chi\ \pi_1 \ldots \pi_v \\ (3) & TE \overset{\text{atom}}{\vdash} atom_i : \tau_i \quad (0 \leq i \leq n)\end{array}}{TE \overset{\text{exp}}{\vdash} cons\ atom_1 \ldots atom_n : \chi\ \pi_1 \ldots \pi_v}$$

Figure 4.7: The *CONS-EXP* type rule

1. lookup the constructor's type signature and arity, n, in the constructor environment, CE.
2. create a fresh instance of the polytype.
3. match the inferred types of the arguments with the monotype from step 2. Also the number of arguments has to match the arity, n, from step 1, so satisfying the second restriction presented in section 4.5.1.

generalisation and specialisation as the previous examples illustrate, the *GEN* and *SPEC* rules are used to convert monotypes to polytypes and vice versa. The *generic instance* of a monotype τ is $\forall \alpha_1 \ldots \alpha_n . \tau$, where each α_i is a free type variable of τ, which is also free in the current environment. Similarly, an *instance* of a type signature is simply the right-hand side monotype with all occurrences of the type variables replaced with fresh ones

unification Robinson's unification algorithm [Robinson, 1965] is used to determine if two monotypes are compatible (see step 3 of the *CONS-EXP* type rule for an example of where unification is used.) If unification succeeds, the algorithm returns the most general type that matches both of the arguments, as well as a set of substitutions. These substitutions represent the restrictions (on free type variables) that have had to be made in order to resolve the two types, and they must be applied to the current environment to ensure consistency. For this application, unification fails if a substitution of the form $\alpha \mapsto \ldots \alpha \ldots$ would be required to unify the two types. This is commonly referred to as the *occurs check*, which prevents the introduction of infinite types

Animating the algorithm

As the development of Hindley–Milner style algorithms using functional programming languages is well documented in the literature [Hancock, 1987; Peyton Jones and Lester, 1991; Jones, 1994], only a brief overview is given here.

Making use of the syntax-driven nature of the rules, the first step of the process is to construct type signatures for each of the rule groups. There are three general forms that the signatures can take, with examples of each being shown below:

Haskell

```
atomInferType       :: Atom       -> TypeState ->  MonoType
expressionInferType :: Expression -> TypeState -> (MonoType, TypeState)
bindingsInferType   :: Bindings   -> TypeState -> (GeneralVariableEnv,
                                                   TypeState)
```

where `TypeState` is the Haskell representation of the total environment, TE, with the addition of some miscellaneous extras, such as a unique name supply for specialising polytypes. Similarly, `GeneralVariableEnv` corresponds to the general variable environment, GE, and `MonoType` to the abstract syntax of monotypes (see figure 4.2). The definition of these types is not discussed here, but the general technique for doing so is illustrated in section 4.8.10.

Each functions is made up of a series of pattern matched definitions, with one branch for every constructor associated with the primary data type:

Haskell

```
expressionInferType (Let     exp_id binds exp) type_state = ...
   .
expressionInferType (Value   exp_id   literal) type_state = ...
```

The body of the definition will depend upon the rule it implements, and, as an example, the Haskell implementation of the *CONS-EXP* rule (see figure 4.7) is given below:

Haskell

```
expressionInferType (Cons exp_id cons atoms) type_state

  | (envIsDefined cons constructor_env) && (cons_arity == length atoms)

  = (substApply subst result_type, substApplyToEnv type_state2)

  where
  constructor_env           = typestateGetConsEnv          type_state
  (arity,         polytype) = envGet              cons     constructor_env
  (monotype,   type_state1) = polytypeSpecialise  polytype type_state
  (arg_types,  result_type) = monotypeSplitFun    monotype
  (atom_types, type_state2) = atomsInferType      atoms    type_state1
  OK           subst        = monotypesUnify      arg_types atom_types
```

The order of the definitions closely follows that of the original rule: **`envGet`** is a library function and its use corresponds to the first step of the rule i.e. $(cons, (n, \sigma)) \in CE$; while **`polytypeSpecialise`** and **`atomsInferType`** are themselves type rules, and complete the second and third steps respectively. Of the remaining expressions, only **`monotypeUnify`** is of significance, ensuring, as it does, that the types of the atoms match those specified by the constructor's declaration.

The guard expression checks that the constructor is defined within the constructor environment, CE, and that it is also fully saturated (see restriction 2 in section 4.5.1). These are both implicit conditions of the *CONS-EXP* rule.

4.5.4 Free variables

From an operational perspective, free-variable information is essential whenever either a closure or a continuation has to be created – it identifies which variables are *live* and need to be saved so that the computation can be re-started (either when the closure's value is demanded, or when the continuation is returned to). The STG$'$ language's lambda-form annotation cover the first usage, but the second is unsupported. Moreover, incorrect annotations will lead to problems – therefore, an algorithm is needed which can both generate the missing data and check the decorations.

By nature, free-variable algorithms are simple [Peyton Jones, 1987, page 14], and the only complication to developing an algorithm for the STG$'$ language is the restriction imposed by Peyton Jones [1992, section 4.1.2]: free variables should not include any variables bound at the top level of the program. The solution is to pass the top-level variables as an argument to all of the algorithm's rules, as illustrated by the $\mathcal{FV}_{program}[\![\,]\!]$ rule:

$$\boxed{\mathcal{FV}_{program} \left[\!\!\left[\begin{array}{lcl} var_1 & = & lambda_1 \\ & \vdots & \\ var_n & = & lambda_n \end{array} \right]\!\!\right] \begin{array}{l} = \{\} \\ = \bigcup_{i \leq n} \mathcal{FV}_{lambda}[\![lambda_i]\!]\ \{var_1, \ldots, var_n\} \end{array} \qquad \begin{array}{r} \text{(definition)} \\ \text{(derived)} \end{array}}$$

To ensure that the language's lexical scoping rules are followed, the set of global variables, g, has to be trimmed whenever a new variable is defined, as done by the rule handling algebraic alternatives:

$$\begin{array}{rl} \mathcal{FV}_{aalt}[\![cons\ var_1 \ldots\ var_n \rightarrow exp]\!]\ g = & \mathcal{FV}_{exp}[\![exp]\!]\ g' \setminus vars_{bound} \\ & \text{where } g' = g \setminus vars_{bound} \\ & \text{and} \quad vars_{bound} = \{var_1, \ldots, var_n\} \end{array}$$

The animation process is straightforward, as shown by the Haskell implementation of the previous rule:

Haskell

```
aaltFreeVars :: AlgebraicAlt -> Variables -> Variables

aaltFreeVars (AlgebraicAlt cons vars_bound exp) globals

  = expressionFreeVars exp globals' \\ vars_bound

  where globals' = globals \\ vars_bound
```

The complete set of rules is presented in appendix E.

4.6 Annotations are not enough?

The previous section presented two algorithms, both of which generate information that may be of use to a compiler. In this section, the problem of encoding this data is addressed, with there being two obvious solutions:

extend the abstract syntax this is the approach used to record the free variables and update flag of a lambda-form. When considering the amount of generated data, it becomes clear that this method is not a general solution, as the language constructs would quickly be obscured by operational annotations. Furthermore, each algorithm would have to return a modified version of the original program, complicating all aspects of the system.

use an attribute database a database makes it possible to unobtrusively record the required information, such that the addition of new algorithms will not entail the modification of existing routines. The main problem, apart from the introduction of hidden state, is that some form of key is required to access the data.

Throughout this report, the existence of a program-specific database is assumed. Two main types of key are used: identifiers, such as variable and constructor names, and unique labels (typically integers) attached to language constructs – the *ExpressionId* field presented in section 4.3.5 is an example of the latter type of key.

When accessing the database, the algorithm which generated the required information should be clearly identified. So for example, the free variables of an expression would be referred to as $\mathcal{FV}[\![exp]\!]$, and the type of a variable as $\vdash var : \tau$. Notice the omission of both the rule name and the formal arguments in each reference. The actual mechanism used to store and retrieve the information will not be considered unless it impacts upon the topic under discussion.

4.7 Denotational semantics

The denotational semantics presented in this section is essentially the non-strict model described by Peyton Jones and Launchbury [1991, section 3.2], with only a few minor modifications. The reader interested in further information on the motivation and theoretical underpinnings of denotational semantics is referred to [Schmidt, 1986], while Stoy's seminal work [Stoy, 1977] and Tennent's short introduction [Tennent, 1976] are also both highly recommended.

4.7.1 Domain equations

The domains used by the valuation functions are defined using the following recursive equations:

$$\begin{array}{lcl}
I\# & = & \text{The set of fixed-precision integers} \\
F\# & = & \text{The set of fixed-precision floating-point numbers} \\
Id & = & \text{The set of all identifiers} \\
Cons & = & \mathbf{Val}^* \\
Fun & = & \mathbf{Val} \to \mathbf{Val} \\
\mathbf{Val} & = & I\# \cup \cdots \cup F\# \cup Id \cup (Cons + Fun)_\perp \\
\mathbf{Env} & = & Id \to \mathbf{Val}
\end{array}$$

4.7.2 The meta-language

The notation used here follows the standard conventions [Schmidt, 1986, pages 52–53] and is summarised in table 4.3. Note that even though the meta-language bears a strong resemblance to the lambda calculus, it should not be confused with it.

4.7.3 Valuation functions

Figure 4.8 shows the valuation functions for well-formed programs and bindings, figure 4.9 deals with expressions, default alternatives and atoms, and figure 4.10 handles `case` alternatives. To improve readability, the injection, projection and domain membership functions have been omitted.

The conversion of literals to the corresponding domain values by the $\mathcal{L}[\![\;]\!]$ function, is

operation	result's domain	description
$\lambda x.e$	$A \to B$	function construction, such that for all $a \in A$, $[a/x]e$ has a unique value in B
$(e_1\ e_2)$	B	function application such that $e_1 \in A \to B$, and $e_2 \in A$
$let\ x = e_1\ in\ e_2$	**Val**	local definition
$case\ e\ of$ $[]\ pattern_1 \to e_1$ $[]\ \ldots$ $[]\ pattern_n \to e_n$	A	conditional selection, such that for $1 \leq i \leq n$, $e_i \in A$
$\langle e_1, \ldots, e_n \rangle$	**Val***	short form of $\{1 \mapsto e_1, \ldots, n \mapsto e_n\}$
$fix(\lambda x.e)$	A	the fixed-point operator, such that $\lambda x.e \in A \to A$

Table 4.3: The meta-language of the denotational semantics

$$
\begin{aligned}
\mathcal{P}rogram[\![program]\!] &: \mathbf{Val} \\
\mathcal{P}rogram[\![typedecls\ bindings]\!] &= \mathcal{E}[\![\texttt{letrec}\ bindings\ \texttt{main}]\!]\ \emptyset \\
\\
\mathcal{B}inds[\![binds]\!] &: \mathbf{Env} \to \mathbf{Env} \\
\mathcal{B}inds[\![bind_1 \ldots bind_n]\!]\ \rho &= \textstyle\bigoplus_{i \leq n} \mathcal{B}ind[\![bind_i]\!]\rho \\
\\
\mathcal{B}ind[\![bind]\!] &: \mathbf{Env} \to \mathbf{Env} \\
\mathcal{B}ind[\![var = lambda\_form]\!]\ \rho &= \{var \to \mathcal{LF}[\![lambda\_form]\!]\ \rho\} \\
\\
\mathcal{LF}[\![lambda\_form]\!] &: \mathbf{Env} \to \mathbf{Val} \\
\mathcal{LF}[\![vars_{free}\ \pi\ var_{arg_1} \ldots var_{arg_n} \to exp]\!]\ \rho &= \lambda \epsilon_1 \ldots \epsilon_n.(\mathcal{E}[\![exp]\!]\ (\rho \overset{\rightarrow}{\oplus} \{var_{arg_1} \mapsto \epsilon_1, \ldots \\
&\qquad var_{arg_n} \mapsto \epsilon_n\}))
\end{aligned}
$$

Figure 4.8: Denotational semantics of STG′ programs and bindings

$$
\begin{array}{rcl}
\mathcal{E}\llbracket exp \rrbracket & : & \mathbf{Env} \rightarrow \mathbf{Val} \\
\mathcal{E}\llbracket \texttt{let}\ bindings\ exp \rrbracket\ \rho & = & \mathcal{E}\llbracket exp \rrbracket\ (\rho \overset{\rightarrow}{\oplus} \mathcal{B}inds\llbracket bindings \rrbracket\ \rho) \\
\mathcal{E}\llbracket \texttt{letrec}\ bindings\ exp \rrbracket\ \rho & = & let\ \rho' = fix\ (\lambda\rho'.\mathcal{B}inds\llbracket bindings \rrbracket\ (\rho \overset{\rightarrow}{\oplus} \rho')) \\
 & & in\ \mathcal{E}\llbracket exp \rrbracket\ (\rho \overset{\rightarrow}{\oplus} \rho') \\
\mathcal{E}\llbracket \texttt{let\#}\ var = exp_{rhs}\ exp \rrbracket\ \rho & = & case\ (\mathcal{E}\llbracket exp_{rhs} \rrbracket\ \rho)\ of \\
 & & \qquad \bot \quad \rightarrow \quad \bot \\
 & & \qquad \epsilon \quad \rightarrow \quad \mathcal{E}\llbracket exp_{body} \rrbracket\ (\rho \overset{\rightarrow}{\oplus} \{var \mapsto \epsilon\}) \\
\mathcal{E}\llbracket \texttt{letstrict}\ var = exp_{rhs}\ exp_{body} \rrbracket\ \rho & = & case\ (\mathcal{E}\llbracket exp_{rhs} \rrbracket\ \rho)\ of \\
 & & \qquad \bot \quad \rightarrow \quad \bot \\
 & & \qquad \epsilon \quad \rightarrow \quad \mathcal{E}\llbracket exp_{body} \rrbracket\ (\rho \overset{\rightarrow}{\oplus} \{var \mapsto \epsilon\}) \\
\mathcal{E}\llbracket \texttt{case}\ exp\ alts\ default \rrbracket\ \rho & = & \mathcal{A}lts\llbracket alts \rrbracket\ \rho\ (\mathcal{E}\llbracket exp \rrbracket\ \rho)\ (\mathcal{D}\llbracket default \rrbracket\ \rho) \\
\mathcal{E}\llbracket var_{fun}\ atom_1 \ldots atom_n \rrbracket\ \rho & = & let\ \ fun = \rho\ var_{fun} \\
 & & \qquad arg_i = \mathcal{A}tom\llbracket atom_i \rrbracket\ \rho,\ \ (0 \leq i \leq n) \\
 & & in\ (\cdots((fun\ arg_1)\ arg_2)\cdots arg_n) \\
\mathcal{E}\llbracket cons\ atom_1 \ldots atom_n \rrbracket\ \rho & = & \langle cons, \mathcal{A}tom\llbracket atom_1 \rrbracket\ \rho, \ldots, \mathcal{A}tom\llbracket atom_n \rrbracket\ \rho\rangle \\
\mathcal{E}\llbracket literal \rrbracket\ \rho & = & \mathcal{L}\llbracket literal \rrbracket \\
 & & \\
\mathcal{D}\llbracket default \rrbracket & : & \mathbf{Env} \rightarrow \mathbf{Val} \\
\mathcal{D}\llbracket \_ \rightarrow exp \rrbracket\ \rho & \rightarrow & \mathcal{E}\llbracket exp \rrbracket\ \rho \\
 & & \\
\mathcal{A}tom\llbracket atom \rrbracket & : & \mathbf{Env} \rightarrow \mathbf{Val} \\
\mathcal{A}tom\llbracket var \rrbracket\ \rho & = & \rho\ var \\
\mathcal{A}tom\llbracket literal \rrbracket\ \rho & = & \mathcal{L}\llbracket literal \rrbracket
\end{array}
$$

Figure 4.9: Denotational semantics of STG$'$ expressions, defaults and atoms

$$\mathcal{A}lts[\![alts]\!] \ :\ \mathbf{Env} \to \mathbf{Val} \to \mathbf{Val} \to \mathbf{Val}$$

$$\begin{array}{l}
\mathcal{A}lts \left[\!\!\left[\begin{array}{lcl} cons_1\ var_{11}\ldots var_{1a_1} & \to & exp_1 \\ & \vdots & \\ cons_n\ var_{n1}\ldots var_{na_n} & \to & exp_n \end{array}\right]\!\!\right] \rho\ \epsilon_{exp}\ \epsilon_{default} \\
= case\ \epsilon_{exp}\ of \\
\quad \begin{array}{lcl}
[]\ \bot & \to & \bot \\
[]\ \langle cons_1, \epsilon_{11}, \ldots, \epsilon_{1a_1}\rangle & \to & \mathcal{E}[\![exp_1]\!]\ (\rho \overset{\to}{\oplus} \{var_{11} \mapsto \epsilon_{11}, \ldots, var_{1a_1} \mapsto \epsilon_{1a_1}\}) \\
[]\ \ldots & & \\
[]\ \langle cons_n, \epsilon_{n1}, \ldots, \epsilon_{na_n}\rangle & \to & \mathcal{E}[\![exp_n]\!]\ (\rho \overset{\to}{\oplus} \{var_{n1} \mapsto \epsilon_{n1}, \ldots, var_{na_n} \mapsto \epsilon_{na_n}\}) \\
[]\ else & \to & \epsilon_{default}
\end{array}
\end{array}$$

$$\mathcal{A}lts \left[\!\!\left[\begin{array}{lcl} literal_1 & \to & exp_1 \\ & \vdots & \\ literal_n & \to & exp_n \end{array}\right]\!\!\right] \rho\ \epsilon_{exp}\ \epsilon_{default} = case\ \epsilon_{exp}\ of\ \begin{array}{lcl}
[]\ \bot & \to & \bot \\
[]\ literal_1 & \to & \mathcal{E}[\![exp_1]\!]\ \rho \\
[]\ \ldots & & \\
[]\ literal_n & \to & \mathcal{E}[\![exp_n]\!]\ \rho \\
[]\ else & \to & \epsilon_{default}
\end{array}$$

Figure 4.10: Denotational semantics of STG′ `case` alternatives

illustrated by the following example:

$$\begin{array}{lcl}
\mathcal{L}[\![literal]\!] & : & \mathbf{Val} \\
\mathcal{L}[\![1]\!] & \mapsto & 1 \\
 & \vdots &
\end{array}$$

Primitive functions are the equivalent of lambda-calculus δ-rules [Barendregt, 1981], and, as such, care should be taken with their definition. The following rule serves as an example of the $\mathcal{E}[\![primitive\ atoms]\!]$ set of rules:

$$\begin{array}{lcl}
\mathcal{E}[\![plusInt\#\ atom_1\ atom_2]\!]\ \rho & = & let\ \epsilon_1 = \mathcal{A}tom[\![atom_1]\!]\ \rho \\
 & & \qquad \epsilon_2 = \mathcal{A}tom[\![atom_2]\!]\ \rho\ in\ \epsilon_1 +_{I\#} \epsilon_2
\end{array}$$

4.7.4 Programming denotational semantics

There have been two main approaches taken to animating denotational semantics: the first prescribes hand-coding the rules directly into a general-purpose programming language, such as ML [Jouvelot, 1986] or even Pascal [Allison, 1983]; while the second advocates the use of a meta-language, as typified by Navel [Michaelson, 1993] or Wand's prototyping system [Wand, 1984]. In keeping with the rest of this thesis, the former approach is advocated here, with the work of Jouvelot [1986] serving as a useful template. So, for example, an element from the **Val** domain is defined as:

Haskell

```
data ValElement = BottomElement
                | IHashElement  Int
                | FHashElement  Float
                |  ConsElement  [ValElement]
                |   FunElement  (ValElement -> ValElement)
                |    IdElement  String
```

and the *plusInt#* primitive becomes:

Haskell

```
expressionDenotes (Primitive exp_id "plusInt#" (Atoms [atom1, atom2])) rho

  = let IHashElement e1 = atomDenotes atom1 rho
        IHashElement e2 = atomDenotes atom2 rho in IHashElement (e1 + e2)
```

Notice that Haskell's strong typing automatically detects missing injection and projection operations.

This approach has also been used to animate a continuation-passing semantics for APOSTLE, an object-oriented language for parallel and distributed discrete-event simulation [Wonnacott and Bruce, 1996]. This work is described in [Booth, Bruce and Ben-Dyke, 1996, section 3 and 4.2, and appendices A and B].

4.8 Graph reduction and the sequential STG machine

The STG machine [Peyton Jones and Salkild, 1989; Peyton Jones, 1992] is just one of a large number of abstract machines for performing graph reduction [Wadsworth, 1971, chapter 4] (arguably the most efficient approach to evaluating non-strict functional languages). Its selection over the other candidates, however, can be justified by the models comparative efficiency and wide-spread usage.

This section provides an overview of the sequential STG machine, starting with a more detailed examination of the merits of this system in section 4.8.1. Section 4.8.2 then presents the notation used throughout the remainder of this chapter, while sections 4.8.3 through 4.8.9 look at the state-transition model of the abstract machine.

4.8.1 Why use the STG machine?

Considering the large number of viable alternatives, including the ABC machine [Plasmeijer and van Eekelen, 1993a], TIM [Fairbairn and Wray, 1981; Chitnis, Satpathy and Oberoi, 1995], or a G-machine derivate, such as the $\langle v, G\rangle$-machine [Johnsson, 1991] or the GAML system [Maranget, 1991], what are the reasons for selecting the STG machine?

1. the STG$'$ language is based on the STG language, which serves as the abstract machine code for the STG machine (see chapter 4).

2. the efficiency of the STG machine has been demonstrated by GHC, the Glasgow Haskell compiler, which ranked number one in a recent benchmark study [Hartel, 1994]. The relationship between the specification and its implementation is explored in section 4.8.10.

3. a number of important optimisations can be realised as simple source-to-source transformations [Howe and Burn, 1994; Peyton Jones and Launchbury, 1991, section 5.1], thereby avoiding the need to provide special machine support.

4. the self-updating model of thunks [Peyton Jones, 1992, section 3.1.2] and the uniform representation of closures [Peyton Jones, 1992, section 3.1.3] (which gives rise to the *tagless* nature of the model) allows for the seamless integration of threads, remote references, and skeletons into the basic model. For example, Mattson Jr. [1993a, figures 4.2 and 4.3, pages 78 and 79] uses *black* and *grey holes* to provide automatic thread-level synchronisation.

5. the STG machine has served as the core technology for a number of parallel implementations. These cover a range of platforms, including message-passing [Hwang and Rushall, 1992], shared-memory [Mattson Jr., 1993a], vector [Hill, 1994], and hybrid [Chakravarty, 1994] architectures.

6. Sestoft [1994] has derived a simplified STG machine, offering hope for the development of a correctness proof for the full version.

Note that some of the arguments presented here echo those from section 4.2. With regards to the exact differences between the various models of graph reduction, the taxonomy proposed by Douence and Fradet [1995] is highly recommended.

4.8.2 Terminology

Peyton Jones [1992, section 5] specifies the STG machine in terms of a state-transition system. While the presentation does bear a resemblance to Plotkin's structured operational semantics [Hennessy, 1990], the exact relationship is not clear. In fact, Hill [1994, chapter 6, page 94] questions the theoretical foundations of the work, saying:

> "The operational semantics given here is a minor abstraction of the assembly code tinkering that was required to implement DP Haskell. However it does provide a clean definition of the implementation of the language"

Rather than attempting to develop a formal description of the STG machine, the original semantics is adopted here, complete with the aforementioned limitations. Furthermore, relating the operational model with the denotational semantics presented in section 4.7 is outside the scope of this thesis.

A state-transition system comprises: a definition of the state in terms of its components, an initial state, a set of state-transition rules, and a set of final states. Each of these items is discussed in the following sections (the presentation is biased towards the modelling of abstract machines for language interpreters.) Section 4.8.10 describes the Haskell animation of such systems.

State as a tuple

The state is used to represent all elements of the system to be modelled, so, for example, a microprocessor's state would include the register file, main memory, and instruction pipeline (see section 7.3). As the component make-up is likely to remain constant with respect to time, it is sensible to represent the state as a tuple of values, $(code, component_1, \ldots, component_n)$, although it is often convenient to omit the brackets and commas, i.e. $code\ component_1 \cdots component_n$. The ordering of the fields is not significant, but is invariable. The *code* component is the primary driving force behind the evaluation process and serves a role similar to that of a microprocessor's instruction stream.

component	notation	description
n-ary Tuples	$(a_1, \ldots, a_n)$	general representation
Sets	$\emptyset$	empty set
	as or $\{a_1, \ldots, a_n\}$	general representation
	$a \in as$ or $a \notin as$	membership tests
	$as \cup bs$, $as \cap bs$, and $as \setminus bs$	standard set operations
Sequences	$\langle\rangle$	empty sequence
	xs or $\langle x_1, \ldots, x_n\rangle$	general representation
	$x : xs$	item addition
	$xs \,!\, i$	indexing
	$xs +\!\!+\, ys$	concatenation
	$\mathit{length}\ xs$	sequence length
Environments	$\{\}_{env}$	empty environment
	ρ, σ, or $\{a_1 \mapsto x_1, \ldots, a_n \mapsto x_n\}_{env}$	general representation
	$(a, x) \in \rho$ or $\rho\ a$	value lookup
	$\mathit{dom}(\rho)$	domain extraction
	$\rho_1 \overrightarrow{\oplus} \rho_2$	extension (ρ_2 has precedence)
	$\rho_1 \oplus \rho_2$	merging
Stacks	Stacks use the same notation as sequences, although sub-. scripts may be used to differentiate between the two. For example, the empty stack is often represented as $\langle\rangle_{stack}$	
Heaps	h or $\begin{bmatrix} a_1 \mapsto c_1 \\ \cdots \\ a_n \mapsto c_n \end{bmatrix}$	general representation
	$h[a \mapsto c]$	heap access or extension
	$\mathit{size}_{heap}(h)$ and $\mathit{size}_{closure}(c)$	size of heaps and closures
Sum of products	$\mathit{Cons}_1\ \mathit{component}_{11} \ldots \mathit{component}_{1a_1}$	general representation
	$\cdots$	
	$\mathit{Cons}_n\ \mathit{component}_{n1} \ldots \mathit{component}_{na_n}$	

Table 4.4: Example state components

Components

The state-transition system is based upon the matching and manipulation of components. Typically, a component will either be: a standard mathematical entity, such as a set, sequence, tuple, or variable; an abstract type, including environments, stacks and heaps; or a sum-of-products type, akin to Haskell's algebraic data types. In fact, the notation used is similar to that of Haskell, as illustrated by table 4.4. For an overview of the semantics of pattern matching, [Peyton Jones and Wadler, 1987] is recommended. The *code* component is typically an algebraic type, with each constructor representing a different mode of operation.

The initial state

The initial state is used to bootstrap the abstract machine. This is the only point at which external values can be referenced as there is no support for input or output; for example, the program to be evaluated may be incorporated into the state without specifying its exact origin.

State-transition rules

The simplest form a state-transition rule can take is: $pattern_{source} \Longrightarrow pattern_{target}$. If a state matches a rule's source pattern, then a transition occurs and a new state is constructed as prescribed by the rule's target pattern. Rules can also include explicit guard conditions and auxiliary definitions:

$$\boxed{\begin{array}{lllll} & (pattern_{code}, & pattern_1, & \ldots, & pattern_n) \\ \multicolumn{5}{l}{\text{such that } condition_a \cdots condition_m} \\ \Longrightarrow & (code', & component'_1, & \ldots, & component'_n) \\ \multicolumn{5}{l}{\text{where } definition_p \cdots definition_z} \end{array}}$$

All of the implicit and explicit conditions (patterns and guards respectively) have to hold for the rule to match a given state.

Peyton Jones [1992, section 5, page 33] restricts the rule set by require that any given state matches, at most, one transition rule. If the definitions, conditions, and component specifications are purely functional in nature, then the resulting system is obviously deterministic. However, by relaxing this restriction, a number of important behaviours can be specified (see sections 9.3.2 and 6.2.2).

The final state

Starting with the initial state, the state transitions will continue until one of the following situations arise: the current state does not match any of the rules, suggesting either an error, or omission, in the rule set or initial state; or the state matches a final-state pattern, indicating successful completion of the evaluation. The specification of a valid final state is similar to that of a transition rule, but without the target state. It is possible that the system never terminates.

4.8.3 The abstract state of the STG-machine

The STG machine uses the following state to model graph reduction:

$$\boxed{\begin{array}{lrrrrrr} & (code, & argument\ stack, & return\ stack, & update\ stack, & heap, & global\ env) \\ \equiv & (code, & as, & rs, & us, & h, & \sigma) \end{array}}$$

Following the presentation of Peyton Jones [1992, section 5], the individual components of the state are specified in table 4.5. The relationship between the *code* field and the state-transition rules is illustrated in figure 4.11. Chapter 8 deals with the implementation of both the state components and the transition rules.

	specification	description	rules
code	$Eval\ exp\ \rho$	evaluate *exp* in environment ρ	all but 16–17A
	$Enter\ a$	closure application	1, 2, 15, 17, 17A
	$Return_\tau\ result$	return value of type τ to continuation	5–14 and 16
argument stack	stack of *values*	structure for passing parameters	1, 2, 15, 16–17A
return stack	stack of *continuations*	structure for storing control information	4–4B, 6–8′, 12′, 13
update stack	stack of *update frames*	update mechanism	15–17A
heap	heap of *closures*	boxed-value storage	2, 3, 8′, 15–17A
global environment	$\sigma\ var = a$	closure address of the top-level bindings	5
value	$Addr\ a$	closure address	1, 3, 8′, 15–17A
	$Int\ k$	literal integer	9–14
continuation	$Case_\tau\ alts\ \rho$	`case` expression	4, 6, 7, 11, 13
	$Forced_\tau\ var\ \rho$	`letstrict` or `let#` expression	4A, 4B, 8′, 12′
update frame	(as, rs, a)	update marker	15–17A
closure	$(lambda\_form, values)$	boxed representation of values	2, 3, 8′, 15–17A

Table 4.5: The state components of the STG machine

rule	description
1	variable application
2	non-updatable closure entry
3	`let(rec)` expression
4	`case` expression
4A	`letstrict` expression
4B	`let#` expression
5	constructor application
6	constructor return (explicit alternative)
7	constructor return (default alternative)
8′	constructor return (`letstrict` exp.)
9	literal value
10	literal variable application
11	integer return (explicit alternative)
12′	integer return (`let# expression`)
13	integer return (default alternative)
14	primitive application (integer addition)
15	updatable-closure entry
16	constructor update
17	partial-application update (I)
17A	partial-application update (II)

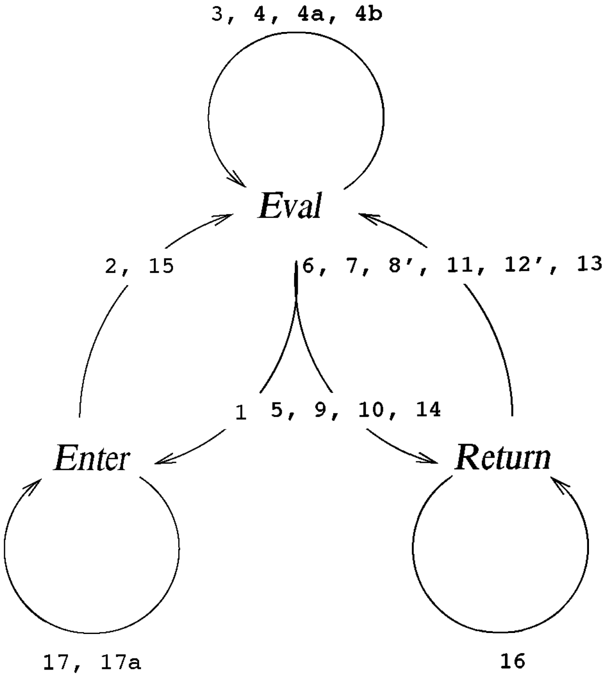


Figure 4.11: The relationship between the STG-machine rules and the *code* component

$$
(4\text{A})\quad
\begin{array}{|lllllll|}
\hline
 & \mathit{Eval}\ (\texttt{letstrict}\ (\mathit{var} = \mathit{exp}_{\mathit{rhs}})\ \mathit{exp}_{\mathit{body}})\ \rho & \mathit{as} & \mathit{rs} & \mathit{us} & h & \sigma \\
\Longrightarrow & \mathit{Eval}\ \mathit{exp}_{\mathit{rhs}}\ \rho & \mathit{as} & \mathit{return} : \mathit{rs} & \mathit{us} & h & \sigma \\
\text{where} & \mathit{return} \ = \ \mathit{Forced}_{\chi\ \pi_1 \ldots \pi_n}\ \mathit{var}\ \mathit{exp}_{\mathit{body}}\ \rho' & & & & & \\
 & \mathit{exp}_{\mathit{rhs}} \text{ is of type } \chi\ \pi_1 \ldots \pi_n & & & & & \\
 & \mathit{dom}(\rho') \ = \ \mathcal{FV}[\![\mathit{exp}_{\mathit{body}}]\!] & & & & & \\
\hline
\end{array}
$$

Figure 4.12: The STG-machine rule for evaluating `letstrict` expressions

$$
(8')\quad
\begin{array}{|llllllll|}
\hline
 & \mathit{Return}_{\chi\ \pi_1 \ldots \pi_n}\ c\ \mathit{ws} & \mathit{as} & (\mathit{Forced}_{\chi\ \pi_1 \ldots \pi_n}\ \mathit{var}\ \mathit{exp}_{\mathit{body}}\ \rho) : \mathit{rs} & \mathit{us} & h & \sigma \\
\Longrightarrow & \mathit{Eval}\ \mathit{exp}_{\mathit{body}}\ \rho' & \mathit{as} & \mathit{rs} & \mathit{us} & h' & \sigma \\
\text{where} & \rho' \ = \ \rho \overset{\rightarrow}{\oplus} \{\mathit{var} \mapsto a\} & & & & & \\
 & h' \ = \ h[a \mapsto (\mathit{vs}\ \texttt{r}\ \{\} \rightarrow c\ \mathit{vs}, \mathit{ws})] & & & & & \\
 & \mathit{vs} \text{ is a sequence of arbitrary distinct variables} & & & & & \\
 & \mathit{length}(\mathit{vs}) \ = \ \mathit{length}(\mathit{ws}) & & & & & \\
\hline
\end{array}
$$

Figure 4.13: The STG-machine rule for returning to a `letstrict` continuation

4.8.4 The STG′ language and the STG machine

In order to adapt the STG machine to work with the STG′ language, two new rules were introduced (rules 4A and 4B) and two existing rules were altered (rules 8 and 12). These rules are shown in figures 4.12 through 4.14, with the exception of rule 4B, which is a slight variation of rule 4A (dealing with the `let#` expression instead of the `letstrict` expression). The global environment has also been extended to allow access to the program's attribute database.

4.8.5 The initial state

The initial state [Peyton Jones, 1992, section 5.1] takes as its only parameters an STG′ program and its attribute database (see section 4.6). The state is constructed so that the code is set to evaluate the variable `main`, all stacks are empty, the heap contains closure's representing all of the program's top-level bindings, and the global environment contains the addresses of each of these closures. For example, the program shown in figure 4.15 (see section B.2 for the definition of `fib.wrk`) would result in the creation of the following initial state:

$$
(12')\quad
\begin{array}{|llllllll|}
\hline
 & \mathit{Return}_{\texttt{Int\#}}\ k & \mathit{as} & (\mathit{Forced}_{\texttt{Int\#}}\ \mathit{var}\ \mathit{exp}_{\mathit{body}}\ \rho) : \mathit{rs} & \mathit{us} & h & \sigma \\
\Longrightarrow & \mathit{Eval}\ \mathit{exp}_{\mathit{body}}\ \rho' & \mathit{as} & \mathit{rs} & \mathit{us} & h & \sigma \\
\text{where} & \rho' \ = \ \rho \overset{\rightarrow}{\oplus} \{\mathit{var} \mapsto k\} & & & & & \\
\hline
\end{array}
$$

Figure 4.14: The STG-machine rule for returning to a `let#` continuation

STG′ code

```
const.Int.* = [] \r [x y] -> case x of { Int x' ->
                             case y of { Int y' ->
                             let# xy = timesInt# [x', y']
                             in Int [xy] ; }; } ;

main = [] \u [] -> let { z = [] \u [] -> fib.wrk 20# ; }
                   in const.Int.* z z ;
```

Figure 4.15: An example STG′ program

code	argument stack	return stack	update stack	heap	globals
$\mathit{Eval}\ \texttt{main}\ \{\}_{env}$	$\langle\rangle_{stack}$	$\langle\rangle_{stack}$	$\langle\rangle_{stack}$	h_{init}	σ

$$\text{where}\quad \begin{aligned} \sigma &= \{\texttt{const.Int.*} \mapsto a_1, \texttt{main} \mapsto a_2\} \\ h_{init} &= \begin{bmatrix} a_1 \mapsto (\texttt{r}\ x\ y \to \texttt{case}\ldots, \langle\rangle) \\ a_2 \mapsto (\texttt{u} \to \texttt{let}\ldots, \langle\rangle) \end{bmatrix} \end{aligned}$$

4.8.6 Variable application, closures, and entry methods

A closure typically represents a variable of boxed type, π, and to access its value it is necessary to invoke the closure's entry method. To illustrate this, the first few transitions of the initial state presented in the previous section are as follows (ignoring the fact that **main**'s closure is updatable):

		$\mathit{Eval}\ \texttt{main}\ \{\}_{env}$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	h_{init}	σ
(rule 1)	$\Longrightarrow$	$\mathit{Enter}\ a_2$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	h_{init}	σ
(rule 2)	$\Longrightarrow$	$\mathit{Eval}\ \left(\texttt{let}\ \begin{matrix} z = \texttt{u} \to \texttt{fib.wrk}\ 20 \\ \texttt{const.Int.*}\ z\ z \end{matrix}\right) \{\}_{env}$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	h_{init}	σ
(rule 3)	$\Longrightarrow$	$\mathit{Eval}\ (\texttt{const.Int.*}\ z\ z)\ \{z \mapsto a_3\}_{env}$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	h_1	σ
	where	$h_1 = h_{init}[a_3 \mapsto (\texttt{u} \to \texttt{fib.wrk}\ 20, \langle\rangle)]$					
(rule 1)	$\Longrightarrow$	$\mathit{Enter}\ a_1$	as_1	$\langle\rangle$	$\langle\rangle$	h_1	σ
	where	$as_1 = \langle a_3, a_3\rangle$					
(rule 2)	$\Longrightarrow$	$\mathit{Eval}\ (\texttt{case}\ x\ \mathit{alts}_1)\ \{x \mapsto a_3, y \mapsto a_3\}_{env}$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	h_1	σ
	where	$\mathit{alts}_1 = \texttt{Int}\ x' \mapsto \texttt{case}\ldots$					

Notice how the second application of rule 1 results in **const.Int.***'s arguments being pushed onto the stack, which are then removed and bound upon entry to the function's closure (see the local environment of the last state).

Even though the operational description only defines one type of closure and one standard entry method, a complete implementation would support a richer mixture (see, for example, sections 6.4.3 and 6.2).

4.8.7 Returning values

With the exception of the variable main, the evaluation of an expression is always initiated by either a case, let#, or letstrict expression. Before the new evaluation begins, each of their associated rules pushes a continuation onto the return stack . This is then removed and invoked when the new expression's head-normal form is reached, thereby returning control to the original expression. To illustrate this process, the example from the previous section is continued below:

		$Eval\ (\texttt{case}\ x\ alts_1)\ \{x \mapsto a_3, y \mapsto a_3\}_{env}$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	h_1	σ
	where	$alts_1 = \texttt{Int}\ x' \mapsto \texttt{case}\ldots$					
(rule 4)	$\Longrightarrow$	$Eval\ x\ \{x \mapsto a_3\}_{env}$	$\langle\rangle$	rs_1	$\langle\rangle$	h_1	σ
	where	$rs_1 = \langle Case_{\texttt{Int}}\ alts_1\ \{y \mapsto a_3\}_{env}\rangle$					
(rule 1)	$\Longrightarrow$	$Enter\ a_3$	$\langle\rangle$	rs_1	$\langle\rangle$	h_1	σ
	$\Longrightarrow^*$	$Return_{\texttt{Int}}\ \texttt{Int}\ 21\,891$	$\langle\rangle$	rs_1	$\langle\rangle$	h_2	σ
(rule 6)	$\Longrightarrow$	$Eval\ (\texttt{case}\ y\ alts_2)\ \{x' \mapsto 21\,891, y \mapsto a_3\}_{env}$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	h_2	σ
	where	$alts_2 = \texttt{Int}\ y' \mapsto \texttt{let\#}\ldots$					
(rule 4)	$\Longrightarrow$	$Eval\ y\ \{y \mapsto a_3\}_{env}$	$\langle\rangle$	rs_2	$\langle\rangle$	h_2	σ
	where	$rs_2 = \langle Case_{\texttt{Int}}\ alts_2\ \{x' \mapsto 21\,891\}_{env}\rangle$					
	$\Longrightarrow^*$	$Return_{\texttt{Int}}\ \texttt{Int}\ 21\,891$	$\langle\rangle$	rs_2	$\langle\rangle$	h_2	σ
(rule 6)	$\Longrightarrow$	$Eval\ \left(\texttt{let\#}\ \begin{array}{l} xy = timesInt\#\ x\ y \\ \texttt{Int}\ xt \end{array}\right)\ \rho_1$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	h_2	σ
	where	$\rho_1 = \{x' \mapsto 21\,891, y' \mapsto 21\,891\}_{env}$					
(rule 4b)	$\Longrightarrow$	$Eval\ (timesInt\#\ x\ y)\ \rho_1$	$\langle\rangle$	rs_3	$\langle\rangle$	h_2	σ
	where	$rs_3 = \langle Forced_{\texttt{Int\#}}\ xy\ (\texttt{Int}\ xy)\ \{\}_{env}\rangle$					
(rule 14)	$\Longrightarrow$	$Return_{\texttt{Int\#}}\ 479\,215\,881$	$\langle\rangle$	rs_3	$\langle\rangle$	h_2	σ
(rule 12')	$\Longrightarrow$	$Eval\ (\texttt{Int}\ xy)\ \{xy \mapsto 479\,215\,881\}_{env}$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	h_2	σ
(rule 5)	$\Longrightarrow$	$Return_{\texttt{Int}}\ \texttt{Int}\ 479\,215\,881$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	h_2	σ

Notice how the local and update-frame environments are constantly trimmed to remove redundant entries (see section 6.3.3).

4.8.8 The update mechanism

The update mechanism maintains the laziness of the STG machine by ensuring that an expression will be reduced to head-normal form at most once. The update flag of the STG′ language indicates which expressions need to be updated, and upon entry to an updatable closure both the argument and return stacks are reset. An update is then triggered whenever there are insufficient values on either stack to satisfy an access. Consider, for example, the evaluation of the variable x (the details of which were omitted from the previous section's description):

		$Eval\ x\ \{x \mapsto a_3\}_{env}$	$\langle\rangle$	rs_1	$\langle\rangle$	h_1	σ
(rule 1)	$\Longrightarrow$	$Enter\ a_3$	$\langle\rangle$	rs_1	$\langle\rangle$	h_1	σ
(rule 15)	$\Longrightarrow$	$Eval\ (\texttt{fib.wrk}\ 20)\ \{\}_{env}$	$\langle\rangle$	$\langle\rangle$	us_1	h_1	σ
	where	$us_1 = \langle(a_3, \langle\rangle, rs_1)\rangle$					
	$\Longrightarrow^*$	$Return_{\texttt{Int}}\ \texttt{Int}\ 21\,891$	$\langle\rangle$	$\langle\rangle$	us_1	h_1	σ
(rule 16)	$\Longrightarrow$	$Return_{\texttt{Int}}\ \texttt{Int}\ 21\,891$	$\langle\rangle$	rs_1	$\langle\rangle$	h_2	σ
	where	$h_2 = h_1[a_3 \mapsto (v\ \texttt{r} \rightarrow \texttt{Int}\ v, \langle 21\,891\rangle)]$					

All future entries of the a_3 closure will return the new value without having to repeat the costly evaluation. With regards to partial applications, an update is triggered when a function needs more arguments than are available on the stack (rule 17 or 17A).

When dealing with closures that are known to reduce to constructor applications (using type information) clearing both the argument and return stacks upon entry is unnecessary, and a shorter update frame could be used. The compilation rules to support this idea have not yet been developed.

4.8.9 The final state

Evaluation is complete whenever the machine is in the *Return* mode and all three stacks, *as*, *rs*, and *us*, are empty. The last state of the example shown in section 4.8.7 would be the final state of that evaluation.

4.8.10 Animating state-transition systems

The primary motivation behind the animation process is that of debugging the state-transition system. This includes testing both the correctness of the model (see sections 4.7 and 5.4), and, if feasible, its efficiency (see section 6.2.2). The structure adopted here is based on that outlined by Diller [1994a], although Haskell, rather than Miranda [Holyer, 1991], is used as the target language. The steps are as follows:

1. *create a type signature for each state component.* If the component is not already supported, an abstract type, and associated operations, will have to be developed.

2. *define an algebraic type to represent the abstract state(s).* Despite the tuple representation used throughout this chapter, Haskell's algebraic types are better suited to the role. For states with a large number of components, access and update routines need to be developed to support the implementation of the transition rules.

3. *specify the initial and final states.*

4. *develop a partial ordering for the rule set.* To improve efficiency, Haskell's built-in pattern matching facilities are used during the implementation of the transition rules. The semantics dictate a sequential left-to-right depth-first evaluaton of nested patterns [Hudak et al., 1992, figure 3, page 22]. Hence, the ordering of the rules has to be considered carefully.

5. *encode each transition rule.* The state-transition rule and its Haskell implementation are very similar, with the latter only requiring some additional plumbing to correctly order accesses and updates to the components.

The first and last pairs of rules are discussed in sections 4.8.10 and 4.8.10 respectively, and section 4.8.10 looks at the third step, the animation of the initial and final states. The sequential STG machine is used as the primary example throughout this material. Also, as it forms a key part of the prototyping system, section 4.8.10 validates the STG animation against the Glasgow Haskell compiler.

The state, its components, and abstract data types

By using Haskell's class and module system [Hudak et al., 1992, sections 4 and 5, pages 24–55] to develop abstract data types for the most common components (see table 4.4) the required type signatures can often be generated immediately:

Haskell

```haskell
type ArgumentStack = Stack Value
type ReturnStack   = Stack Continuation
type UpdateStack   = Stack UpdateFrame
type MainHeap      = Heap  Address  Closure
type GlobalEnv     = Env   Variable Address
```

The sum-of-products components are the only exception, and these can be directly converted into data declarations:

Haskell

```
data STGCode = Eval      Expression  LocalEnv    |
               Enter     Address                 |
               ReturnCon Constructor Values      |
               ReturnLit Literal
```

An algebraic type is also used to encode the abstract state:

Haskell

```
data STGState = STGState STGCode  ArgumentStack ReturnStack UpdateStack
                         MainHeap GlobalEnv     Extras
```

As well as holding miscellaneous plumbing information, including a unique name supply and possibly a stream of random numbers, the `Extras` field is used to instrument the transition system.

By using a data type instead of a tuple, it is possible for the state and component types to be recursive. For example, the following definitions could be used to bring the operational model into line with the physical implementation of closures (see chapter 8):

Haskell

```
data Closure     = Closure LambdaForm LocalEnv EntryMethod
type EntryMethod = Address -> Closure -> STGState -> STGState
```

One of the major problems with the animation method is that any change to one or more of the underlying types, particularly that of the abstract state, can require that all associated definitions be updated. Fortunately, Haskell's static type system will identify all of the inconsistencies. Furthermore, by using access and update functions to manipulate the state where ever possible, most of the changes can be localised:

Haskell

```
stgstateGetArgumentStack ::             STGState -> ArgumentStack
stgstateSetMainHeap      :: MainHeap -> STGState -> STGState
```

Another possible approach would be to pass each state component as an individual argument to each state-transition rule. However, this would require a continuation-passing system, making step-based tracing and/or debugging difficult. Furthermore, as noted previously, the ADT approach provides better encapsulation, thereby localising the changes that have to be made when the state is extended or changed.

The initial and final states

The initial state is realised as a Haskell function, the arguments of which equate to the external parameters of the abstract machine. The body is simply a collection of component instantiations:

Haskell

```
stgstateInitialise :: TypeEnvironment -> PrimitiveEnv -> Program -> STGState

stgstateInit type_env primitives program = STGState code as rs us heap ge ex
  where
  code         = Eval (envFindAddress "main" envEmpty globalenv)
  (as, rs, us) = (stackCreate "as", stackCreate "rs", stackCreate "us")
  (ge,   heap) = bindsAllocate (programGetBinds program) ge heapInitialise
  ex           = extrasInititialise type_env primitives
```

Notice how non-strictness has been used to "tie a knot" during the creation of the global environment, ge. Also, as the compiler will automatically resolve the various dependencies, the ordering of the declarations is unimportant.

The final-state predicate is constructed in exactly the same way as the transition rules, except, rather than returning a new state, the result is either True or False

State-transition rules

The entire rule set could be encoded as a single Haskell function, using a guarded binding for each specific rule. However, as the semantics dictate a left-to-right depth-first evaluation of guards and patterns [Hudak et al., 1992, figure 3, page 22], the ordering of the bindings would implicitly define the rule hierarchy (which is typically flat as rules tend not to overlap – see section 4.8.2). This can cause complications when modifying the rules, so it is suggested that the ordering is clearly defined through the use of dispatch functions:

Haskell

```
stgstateTransform stgstate@(STGState code as rs us heap ge ex)
  | stgstateTriggerGC heap = stgstateInitiateGC stgstate
  | otherwise              = step code (stgstateIncReductions stgstate)
  where
  step (Eval      expr local_env) = codeEval      expr local_env
  step (Enter            address) = codeEnter     address
  step (ReturnCon con     values) = codeReturnCon con values
  step (ReturnLit        literal) = codeReturnLit literal
  step  _                         = codeUndefined
```

This technique has the advantage of allowing support functions to be developed along side the appropriate rule. This would not be possible with the one-function approach as GHC does not allow diffuse bindings. It is also easier to instrument the system, as illustrated by the stgstateIncReductions ticky-ticky function.

Note that non-determinism, whether introduced by overlapping rules or explicitly specified in a single rule, can be simulated by, for example, extending the Extras field to include a stream of random numbers. The dispatch function then selects a rule based on the next value in the stream.

The coding of the rules is usually straightforward, and the following functions implement rule 9 (evalLiteral) and rule 3 (evalLet):

Haskell

```
evalLiteral :: Literal -> LocalEnv  -> STGState -> STGState
evalLiteral    literal    local_env    stgstate
  = stgstateSetCode (ReturnLit literal) (stateIncEventLit stgstate)

evalLet :: Bool -> Bindings -> Expression -> LocalEnv -> STGState -> STGState
evalLet recursive binds expression original_env state
  = stateSetCode  (Evaluate expression  local_env) $
    stateIncEventLet                               $
    stateSetMainHeap heap' state
  where
  (binds_env, heap')  = bindsAllocate binds rhs_env old_heap
  local_env           = envMerge original_env binds_env
  rhs_env | recursive = local_env
          | otherwise = original_env
  heap                = stateGetMainHeap state
```

Both of these definitions give rise to one of the main problems affecting the animation. Due to the non-strict semantics of the language, the evaluation of the ticky-ticky counts will be deferred as their values are not needed in the calculation of the new state. Each subsequent step will again defer evaluation, creating a series of linked closures whose length is proportional to the number of transitions made. To prevent this unwanted space leak, infrequently-accessed values have to be artificially forced via dummy `case` expressions.

Benchmarking the STG machine

Following the guidelines laid down by Jain [1991, chapter 25, pages 413–436], this section discusses the verification and validation of the animation of the STG machine – both are essential to having confidence in the output of the animation. Obviously, it is first necessary to consider what the outputs are likely to be:

final result the terminal value of the *code* component is usually $\mathit{Return}_{\mathcal{T}}\ \mathit{value}_{tau}$ (ignoring errors), and $\mathit{value}_{\mathcal{T}}$ is taken to be the final result of the computation.

accumulated totals and statistics this data is stored in the `Extras` field, and primarily records event counts in much the same way as GHC's `ticky-ticky` profiling system. The state components can also act as data sources. For example, the `MainHeap` data type records the number and the size of the stored closures.

traces snapshots of the abstract state are dumped to a file, with the frequency and level of detail controlled by command-line options.

In addition to the usual model-verification techniques [Jain, 1991, section 25.1, pages 413–420], the denotational semantics (see section 4.7) serves as a reference against which the animation's final result can be checked. Furthermore, the close correspondence between the specification and its implementation further simplifies the debugging process.

In order to validate a model it is necessary to have either expert intuition, real-system measurements, or theoretical results [Jain, 1991, section 2.5.2, pages 420–423]. For the sequential STG machine, the outputs can be compared against the `tick-ticky` profiles generated by GHC [AQUA Team, 1993, section 9, page 36] (attempting to predict the run time would be difficult, see section 6.2.2.)

Tables 4.6 through 4.9 present the percentage errors between the estimated and observed values for the `fib`, `primes`, `queens`, and `hamming` benchmark programs (see sections B.2 to B.5 for the STG′-language versions). The measured values include: *closures* and *words*, a record of the number of heap allocations (rules 3, 8′, 16, 17, 17A) and the total memory used; *entries*, a count of the invocations of closure entry routines (rules 2 and 15); *updates*, the number of thunks updated (rules 17 and 17A); and *returns*, a tally of the non-unary constructor returns (rules 6–8′).

The percentage errors range between −4·29% and 13·33%, although the errors tend to zero as the number of transitions increases (with the exception of the *returns* estimates for both the `hamming` and `primes` benchmarks). The discrepancies are largely due to the animation not modelling GHC's input-output mechanism.

4.9 Summary

The STG′ language provides the computational model upon which a parallel intermediate language can be built, and this chapter has defined the language in terms of its:

abstract syntax this is the internal representation used to encode programs, and provides the main structure around which most of the language-processing algorithms are developed.

fib	5	10	15	20
reductions/k	1	9	105	1 160
entries	1·31	0·10	0·00	0·00
returns	0·99	0·08	0·00	0·00

fib -O	5	10	15	20	25
reductions/k	0	2	28	306	3 399
entries	13·33	1·12	0·10	0·00	0·00

Table 4.6: The `fib` benchmark results

primes	50	100	200	300	400
reductions	96 374	348 835	1 293 335	2 826 486	4 930 210
closures	−0·03	−0·01	−0·00	−0·00	−0·15
words	−0·05	−0·01	−0·00	−0·00	−0·15
entries	−0·05	−0·01	−0·00	−0·00	−0·00
updates	−0·02	−0·00	−0·00	−0·00	−0·00
returns	−0·08	−0·02	−0·00	−0·00	−0·00

primes -O	50	100	200	300	400
reductions	79 032	286 485	1 063 671	2 325 964	4 058 756
closures	0·05	0·01	0·00	0·00	−0·20
words	0·05	0·01	0·00	0·00	−0·20
entries	0·02	0·00	0·00	0·00	0·00
updates	0·10	0·02	0·00	0·00	0·00
returns	−0·01	−0·00	−0·00	−0·00	−0·00

Table 4.7: The `primes` benchmark results

language restrictions by imposing restrictions upon the set of valid programs it is possible to ensure the language has an efficient operational semantics. To enforce these rules, a static Hindley–Milner type-inference algorithm has been presented.

denotational semantics this set-based valuation function maps a program directly onto its meaning, and is uncluttered by operational issues.

operational semantics the STG machine, specified as a state-transition system, provides an operational model of the interpretation of the STG$'$ language, and complements the denotational specification.

`queens`	4	5	6	7	8
reductions	8 426	38 630	188 174	902 002	4 568 372
closures	−0·65	−0·18	−0·04	−0·01	−0·01
words	−0·04	−0·21	−0·23	−0·18	−0·13
entries	−1·75	−0·55	−0·15	−0·04	0·10
updates	0	0	0	0	0
returns	−0·85	−0·08	−0·05	0·01	0·01

`queens -O`	4	5	6	7	8
reductions	4 322	16 863	75 102	343 181	1 686 356
closures	−1·98	−0·91	−0·31	−0·08	−0·02
words	−2·77	−1·32	−0·50	−0·14	−0·03
entries	−4·29	−1·74	−0·55	−0·16	−0·04
updates	1·14	0·54	0·23	0·08	0·02
returns	−1·78	−0·24	−0·15	0·04	0·03

Table 4.8: The `queens` benchmark results

`hamming`									
cut off	500			750			1000		
primes	20	50	80	20	50	80	20	50	80
reductions/k	195	702	1 316	255	936	1 759	306	1 147	2 165
closures	0·01	0·00	0·00	0·01	0·00	0·00	0·00	0·00	0·00
words	0·01	0·00	0·00	0·01	0·00	0·00	0·00	0·00	0·00
entries	−0·86	−0·31	−0·17	−0·91	−0·33	−0·19	−0·94	−0·34	−0·20
updates	0·01	0·00	0·00	0·01	0·00	0·00	0·01	0·00	0·00
returns	1·11	0·39	0·22	1·16	0·41	0·23	1·19	0·42	1·63

`hamming -O`									
cut off	500			750			1000		
primes	20	50	80	20	50	80	20	50	80
reductions/k	159	565	1 056	207	752	1 410	249	921	1 734
closures	0·01	0·00	0·00	0·01	0·00	0·00	0·00	0·00	0·00
words	0·01	0·00	0·00	0·01	0·00	0·00	0·00	0·00	0·00
entries	−1·19	−0·43	−0·25	−1·25	−0·46	−0·27	−1·29	−0·48	−0·28
updates	0·01	0·00	0·00	0·01	0·00	0·00	0·01	0·00	0·00
returns	1·58	0·56	0·32	1·66	0·59	0·34	1·71	0·61	0·36

Table 4.9: The `hamming` benchmark results

Chapter 5

Expressing parallelism – static models

5.1 Introduction

In this chapter a number of guidelines are presented for adding support for parallelism into the sequential STG′ language, as described in chapter 4. Typically, this involves extending the abstract syntax, adding language restrictions, and developing a denotational model of the parallel components. The examples used to motivate each of the steps are, where possible, based on the constructs presented in section 2.4. While the issues of language design are not directly addressed, MacLennan's principles [1987, page 547] serve as a useful guide, and are thus reproduced in table 5.1 (the small-caps keywords on the left of the table will be used to refer to these principles throughout the remainder of this chapter).

The basic techniques for introducing parallelism are outlined in section 5.2, while sections 5.3 and 5.4 consider the issues of language restrictions and denotational semantics in the context of parallelism. The chapter is summarised in section 5.5.

5.2 Introducing parallelism into the STG′ language

In line with the rest of this thesis, the introduction of parallelism into the sequential language is syntax driven, admitting the following possibilities:

new production rule the addition of new rules allows the introduction of task-oriented expressions, ranging in power from simple spark expressions to comprehensive algorithmic skeletons.

new primitive function superficially, this has similar properties to the addition of a new production rule. However, this method tends to hide the parallelism from the top-level syntax and semantics, and is not recommended for any but the most routine of situations.

new primitive type extensions to the type system can be used to introduce bulk data types, or to improve the encapsulation of other parallel constructs. Both applications require the definition of new production rules (or primitive functions) with which to manipulate values of the new type.

alteration of an existing expression by either modifying the syntax, type rule or denotational semantics of one of the standard constructs, it is possible to radically

Abstraction	Avoid requiring something to be stated more than once; factor out the recurring pattern
Automation	Automate, mechanical, tedious, or error-prone activities
Defence in Depth	Have a series of defences so that if an error isn't caught by one it will probably be caught by another
Information Hiding	The language should permit modules designed so that (1) the user has all of the information needed to use the module correctly, and nothing more; and (2) the implementor has all the information needed to implement the module correctly, and nothing more
Labelling	Avoid arbitrary sequences more than a few items long; do not require the user to know the absolute position of an item in a list. Instead, associate a meaningful label with each item and allow the items to occur in any order
Localised Cost	Users should pay only for what they use; avoid distributed costs
Manifest Interface	All interfaces should be apparent (manifest) in the syntax
Orthogonality	Independent functions should be controlled by independent mechanisms
Portability	Avoid features that are dependent on a particular machine or small class of machines
Preservation of Information	The language should allow the representation of information that the user might know and that the compiler might need
Regularity	Regular rules, without exceptions, are easier to learn, use, describe, and implement
Security	No program that violates the definition of the language, or its own intended structure, should escape detection
Simplicity	A language should be as small and simple as possible. It should contain the minimum number of concepts with simple rules for their combination
Structure	The static structure of a program should correspond in a simple way to the dynamic structure of the corresponding computations
Syntactic Consistency	Similar things should look similar; different things different
Zero-One-Infinity	The only reasonable numbers are zero, one, and infinity

Table 5.1: MacLennan's language design principles

change the language. As an example, the presented system can be made strict simply by adjusting the semantics of function and constructor application.

hybrid definition it is suggested that any hybrid language be developed incrementally, in that each of the separate items is prototyped in isolation.

Sections 5.2.1 to 5.2.5 examine each of these approaches in greater depth.

5.2.1 New production rules

Introducing parallelism into the language via a new production rule is an attractive proposition for a number of reasons: firstly, all existing algorithms and descriptions will still be valid, and require only the addition of special cases to bring them into line with the new syntax; similarly, the test programs will still be well formed and have the same behaviour; and, finally, the construct is directly visible, making it difficult to overlook or ignore at any stage of the design process.

In general, the addition of a new production rule will proceed as follows;

1. *extend the abstract and concrete syntax.* The primary decision to be made at this stage concerns which production-rule group will be extended. As the concrete syntax will only be used to encode test programs, aesthetic considerations can be set aside, thereby simplifying one of the more difficult aspects of this step.

2. *generate example programs.* Sample programs not only serve as a useful source of test data for the various animations, but also provide an insight into potential pitfalls that may be encountered in the later stages. A random-program generator, along the lines of the hpg utility (see section C.1), may even be of some value.

3. *modify the type-inference and free-variable algorithms.* The main purpose of the type system is to imposes restrictions on the language so as to avoid complicating the run-time system. These limitations arise from consideration of the kind of values manipulated by the new constructs.

4. *update the denotational semantics.* Despite the limitations of set-based denotational descriptions, the development of such models focuses attention on the issues of non-determinism and the default order of evaluation.

The mechanics of the first point have already been outlined in sections 4.3.5 and 4.4, while sections 5.3 and 5.4 cover the last two points respectively. The remainder of this section therefore presents a number of examples, followed by a brief overview of the utility of the major groups of production rules.

The *par* combinator

The traditional production rule, *exp* ⟶ `par` *var exp*, dissociates the thread from the expression it will reduce, making local optimisations difficult. Moreover, the operational reading may well include memory allocation, thereby invoking the SYNTACTIC CONSISTENCY principle, such that the following rule is arguably superior:

$$\boxed{\quad exp \quad \longrightarrow \quad \texttt{letpar}\ \mathit{simplebind}\ exp \quad}$$

Note that this construct is cumbersome for sparking non-local variables, i.e. a formal argument or a pattern-matched variable. Surprisingly, this is an advantage as such usage

Group	Overview
program	useful for adding static or one-off definitions, such as communication channels or initial data mappings
bindings	similar to the *program* group, except allowing the creation of dynamic topologies
binding	appropriate in cases where there will be just one definition involved, or where there is no relationship between each of the bindings
exp	as the examples presented in section 5.2.1 demonstrate, this group is capable of encoding most forms of task-based parallelism

Table 5.2: Extending production-rule groups

should be considered carefully, reflecting the lack of control over the computational content. Plasmeijer and van Eekelen [1993b, section 25.3.5, pages 355–357] force this issue by extending the type system so that all possible sources of parallelism have to be clearly identified.

The dual of this operation, the *seq* combinator, is already represented by the `letstrict` expression. Furthermore, mutually-recursive threads can only be sparked after the corresponding `letrec` expression.

Skeletal operations

Moving on to consider skeletal operations, two similar options exist: (`farm` and `pipeline` are described in section 2.4.3)

$$\boxed{\begin{array}{lcl@{\qquad}lcll} exp & \longrightarrow & skeleton & exp & \longrightarrow & \texttt{farm}\; var_{fun}\; exp & \\ skeleton & \longrightarrow & \texttt{farm}\; var_{fun}\; exp & & \mid & \texttt{pipeline}\; var_{fun_1} \ldots var_{fun_n}\; exp & (n \geq 1) \\ & \mid & \ldots & & \mid & \ldots & \end{array}}$$

The leftmost system provides better encapsulation and, unless the number of skeletons is small, is the recommended solution. The transformations associated with each skeleton can be performed prior to, during, or after the usual STG-language optimisations (see section 3.3), and take exactly the same form:

$$\boxed{\begin{array}{lcl} \texttt{pipeline}\; var_{fun_1}\; var_{fun_2}\; exp & \Longrightarrow & \texttt{let}\; fun' = \ldots\; \texttt{r} \ldots \rightarrow compose\; var_{fun_1}\; var_{fun_2} \\ & & \quad \texttt{farm}\; fun'\; exp \end{array}}$$

These rules can cause the specification of the topology to become diffuse, such that the grouping of related definitions may be necessary i.e. *skeleton* $\longrightarrow$ `farm` *binds* var_{fun} *exp*.

Selecting a production-rule group

All of these examples have extended the *exp* production rules, but for each new addition, all of the groups outlined in table 5.2 should be considered. The groups not covered by this table are either inappropriate (the following section deals with extending the type declarations), have no obvious utility, or can be simulated by the represented groups. To illustrate this last point, consider the following rules: *atom* $\longrightarrow$ *var* | `par` *var* | *literal*, where, operationally, any par-annotated variable should be sparked. Any expression using this syntax can be trivially converted into an equivalent one which uses the `letpar` construct and the standard *atom* rule group, as demonstrated below:

$$\boxed{\begin{array}{lll} f\ var_1 \ldots (\texttt{par}\ var_i) \ldots var_n & \Longrightarrow & \texttt{letpar}\ \ (var_{i,\texttt{par}} = \texttt{r} \rightarrow var_i) \\ & & \qquad f\ var_1 \ldots var_{i,\texttt{par}} \ldots var_n \end{array}}$$

The latter approach not only increases the spark's prominence, but is more in keeping with the operational semantics [Peyton Jones, 1992, section 5, rules 1, 5, and 14]. The relationship between the *lambda_form* and *bind* groups is similar, but either is acceptable, depending on the context.

5.2.2 New primitive functions

The addition of a new primitive function is quick and simple, requiring no modifications to be made to the abstract or concrete syntax, and only entailing the following steps:

1. *add the type signature to the primitive environment, PE.*

2. *add a new valuation function to the denotational semantics.*

This simplicity has a price, in that such functions can only manipulate *atoms*. Furthermore, due to the low profile of the primitives, only uncomplicated computation should be introduced via this method – non-deterministic operations, or operations which affect the order of evaluation, are not appropriate!

5.2.3 New primitive types

Extensions to the type system can be used to introduce bulk data types, improve the encapsulation of other constructs, or to relax some of the language restrictions detailed in section 4.5. Whatever the purpose of the new type, the addition proceeds as follows:

1. *is the type boxed or unboxed?* Before incorporating the new type into the language, a time must be spent considering its machine representation. This helps to determine where to make the extensions in step 2, and to focus the selection of constructs in step 4.

2. *extend the syntax of types.* Based on the deliberations of step 1, new rules are added to the type system first outlined in figure 4.2. Although concerned with the language syntax, most of the points raised in section 5.2.1 also apply here. Furthermore, if the new type is to be allowed to appear in data-type declarations, the additions to the syntax of types must be mirrored in the language syntax (see step 4).

3. *modify the unification algorithm.* This controls how the new type interacts with type variables, i.e. whether values of this type can be manipulated by polymorphic functions. If a type is boxed there is little reason to disallow such interactions.

4. *extend the language syntax and primitive functions.* Facilities to create and manipulate instances of the type are next introduced into the language. Four categories of operations should be considered: value creation; conversion from or to existing types; transformations within the same type; and conditionals, including comparison operations.

$$
\begin{array}{llll}
U & \alpha \quad\quad\quad\quad \mathit{pod} & = & (\mathit{pod}, \{\alpha \mapsto \mathit{pod}\}) \\
U & \langle\!\langle \nu_1 \rangle\!\rangle \quad\quad\quad \langle\!\langle \nu_2 \rangle\!\rangle & = & (\langle\!\langle \nu_1 \rangle\!\rangle, \emptyset), \text{such that } (\nu_1 == \nu_2) \\
U & \langle\!\langle \chi_{\mathtt{tag}_1} \rangle\!\rangle\ \mathit{pod}_{11} \ldots \mathit{pod}_{1n} & & \\
 & \langle\!\langle \chi_{\mathtt{tag}_2} \rangle\!\rangle\ \mathit{pod}_{21} \ldots \mathit{pod}_{2n} & = & (\langle\!\langle \chi_{\mathtt{tag}_1} \rangle\!\rangle\ \mathit{pod}_1 \ldots \mathit{pod}_n, S_1 \oplus \cdots \oplus S_n) \\
 & & & \text{such that } (\chi_{\mathtt{tag}_1} == \chi_{\mathtt{tag}_2}) \\
 & & & \text{where} \\
 & & & (\mathit{pod}_1, S_1) = U\ \mathit{pod}_{11}\ \mathit{pod}_{21} \\
 & & & \quad\quad\quad\quad \vdots \\
 & & & (\mathit{pod}_n, S_n) = U\ (S_{n-1}\ \mathit{pod}_{1n})\ (S_{n-1}\ \mathit{pod}_{2n})
\end{array}
$$

Figure 5.1: An extended unification algorithm for Hill's PODs

5. *update the semantic descriptions.* For each new construct added by the previous stage, steps 2–4 of the method outlined in section 5.2.1 should be followed (or section 5.2.2 for primitive functions). It may be necessary to update the domain equations used by the denotational semantics.

The following examples serve as demonstrations of the method, and also highlight some of the potential applications.

Data-parallel Haskell

Hill has implemented data-parallel Haskell (see section 2.4.2) on the AMT DAP (Distributed Array Processor [MacDonald, 1992]), a SIMD machine using a flexible 64 by 64 grid of 1-bit processors. Operationally, as DAP vectors can only contain unboxed primitive data types [Hill, 1994, table 5.1], a POD is prevented from storing functions or unevaluated expressions. This restriction also impacts upon the representation of algebraic PODs, which are thus stored as tables of simpler PODs – the relationship between an algebraic type, χ, and its flattened representation, χ', is shown below:

$$
\begin{array}{lll}
\mathtt{data}\ \chi = \mathit{cons}_1\ \tau_{11} \ldots \tau_{1a_1} & \Longrightarrow & \mathtt{data}\ \chi' = \mathtt{flat}_\chi\ \chi_{\mathtt{tag}}\ \tau_{11} \ldots \tau_{ia_i} \ldots \tau_{na_n} \\
\quad\quad\quad \vdots & & \mathtt{data}\ \chi_{\mathtt{tag}} = \mathit{cons}'_1 \ldots \mathit{cons}'_n\ \chi_{\mathtt{not\_here}} \\
\quad\quad\quad \mathit{cons}_n\ \tau_{n1} \ldots \tau_{na_n} & &
\end{array}
$$

The first entry encodes the constructor tag, with the remaining entries representing each possible argument of every constructor associated with the data type. These restrictions are reflected in the extensions to the syntax of types of the STG′ language:

Boxed type	π	$\longrightarrow$	pod	POD vector
POD vector	pod	$\longrightarrow$	$\langle\!\langle \nu \rangle\!\rangle$	primitive vector
		$\mid$	$\langle\!\langle \chi_{\mathtt{tag}} \rangle\!\rangle\ \mathit{pod}_1 \ldots \mathit{pod}_n$	flattened algebraic type ($n \geq 0$)

The unification algorithm, U, is extended as shown in figure 5.1, with the first rule stating that PODs are first-class citizens with respect to polymorphism. Figure 5.2 shows the production rules added by Hill [1994, chapter 5] to support the new type – there are no conversion routines defined, only creation ($\langle\!\langle \cdots \rangle\!\rangle$), transformation ($\overline{\text{MAP}}_n$, $\overline{\text{INDICES}}$, $\overline{\text{SEND}}$, and $\overline{\text{FETCH}}$), and conditional ($\overline{\text{CASE}}$) operations. Notice that named defaults, as described in section 4.3.3, can be avoided by using the $\overline{\text{MAP}}_1$ construct.

Expression	exp	$\longrightarrow$	$\overline{\text{MAP}}_n\ (lambda\_form \mid var)\ exp_1 \ldots exp_n$	$(n \geq 1)$
		$\mid$	$\overline{\text{INDICES}}\ var$	
		$\mid$	$\overline{\text{SEND}}\ var\ var$	
		$\mid$	$\overline{\text{FETCH}}\ var\ var$	
		$\mid$	$\overline{\text{CASE}}\ exp\ \overline{\text{OF}}\ palts\ default$	
		$\mid$	$\langle\!\langle cons \rangle\!\rangle\ atoms$	
Parallel alternatives	$palts$	$\longrightarrow$	$lpalt_1 \ldots lpalt_n$	$(n \geq 1)$
		$\mid$	$vars\ apalt_1 \ldots apalt_n$	$(n \geq 1)$
	$lpalt$	$\longrightarrow$	$\forall\ literal \rightarrow exp$	
	$apalt$	$\longrightarrow$	$\forall\ cons \rightarrow exp$	
Atom	$atom$	$\longrightarrow$	$\langle\!\langle \ldots atom \ldots \rangle\!\rangle$	

Figure 5.2: Hill's extended syntax for a data-parallel STG language

Abstract syntax	ν	$\longrightarrow$	PID#	processor identification
Unification rule	U PID# PID#	$=$	(PID#, $\emptyset$)	
New production rules	exp	$\longrightarrow$	**randomPID#** $\mid$ **currentPID#** $\mid \ldots$	
New primitives	$neighborPID\#$	:	Int# $\rightarrow$ PID# $\rightarrow$ PID#	
	$itopPID\#$	:	Int# $\rightarrow$ PID#	

Figure 5.3: The PID# type for restricting access to non-deterministic topology functions

Processor identification

The topology operations detailed in table 2.3 are a potential source of non-determinism, and it is desirable to restrict the employment of their results to purely operational matters. This is achieved naturally through the use of the new unboxed type shown in figure 5.3. Operationally, variables of this type will be represented as unboxed integers, i.e. equivalent to values of type Int#. However, by restricting the constructs and functions that produce and consume values of this new type, the desired encapsulation is achieved.

Pipeline parallelism

Most existing implementations of the *pipeline* skeleton are limited by the static type system to using stages which all have the type $\alpha \rightarrow \alpha$ [Bratvold, 1994, section 3.4.1]. While it is possible to use algebraic data types to circumvent this restriction, this is a cumbersome and inefficient solution. The boxed type outlined in 5.4 offers a more flexible solution.

5.2.4 Altering existing expressions

This method is deceptively simple, as all of the necessary definitions and functions already exist, and only require modification. However, any change, whether it be to the syntax, language restrictions, or denotational semantics, may cause the test programs to either

Abstract syntax	π	$\longrightarrow$	$\overline{\underline{\pi_1 \to \pi_2}}$	pipeline specification
Unification rule	$U\ \alpha\ \overline{\underline{\pi_1 \to \pi_2}}$	$=$	$\left(\overline{\underline{\pi_1 \to \pi_2}}, \left\{\alpha \mapsto \overline{\underline{\pi_1 \to \pi_2}}\right\}\right)$	
	$U\ \overline{\underline{\pi_{11} \to \pi_{12}}}\ \overline{\underline{\pi_{21} \to \pi_{22}}}$	$=$	$\left(\overline{\underline{\pi_1 \to \pi_2}}, S_1 \oplus S_2\right)$ where $(\alpha_1, S_1) = U\ \alpha_{11}\ \alpha_{21}$ $(\alpha_2, S_2) = U\ (S_1\ \alpha_{12})\ (S_1\ \alpha_{22})$	
New primitives	*addstagePipe*	:	$(\alpha_1 \to \alpha_2) \to \overline{\underline{\alpha_2 \to \alpha_3}} \to \overline{\underline{\alpha_1 \to \alpha_3}}$	
	applyPipe	:	$\overline{\underline{\alpha_1 \to \alpha_2}} \to \mathit{List}\ \alpha_1 \to \mathit{List}\ \alpha_2$	
	emptyPipe	:	$\overline{\underline{\alpha_1 \to \alpha_2}}$	

Figure 5.4: Improving pipeline parallelism using a new boxed type

become invalid, fail to terminate, or yield different results. Re-checking the integrity of the collection can be time consuming, and the savings over the addition of a new rule or type may often be negligible. Moreover, most modifications will have to be made simultaneously, before testing can begin in earnest. Hence it is suggested that the alteration of existing expressions should be employed only when major changes are deemed necessary, and no other method is applicable.

The following three sections look at alterations to the abstract syntax, language restrictions, and denotational semantics respectively.

Abstract syntax

Alterations to the abstract syntax are limited to the following:

removal of a production rule the deletion of a rule effectively removes a capability from the language. Examples include the removal of: named defaults – see section 4.3.3; `letrec` expressions, such that recursion can only be defined at the top-level; and the *literal* alternative from the *atom* group, forcing all primitive values to be defined using the `let#` expression. This last change would bring the language more into line with the continuation-passing style advocated by Appel [1992, figure 2.1].

addition of a new non-terminal symbol to a production rule the introduction of an new field can increase the expressiveness of an existing construct. For example, the *lambda_form* could be extended to include a location directive (see section 2.4.4). As a special case, simple expression-based extensions can be avoided altogether by using the attribute database (see section 4.6). This shortcut can be stretched to include bindings and lambda forms, but the access routines become complex.

removal of a symbol from a production rule this is the inverse of the previous operation, and is used to delete extraneous symbols. As an example, Hill [1994, figure 5.2] simplified the *lambda_form* rule to $vars_{args} \to exp$, claiming that the missing operational information could be inferred easily. This is certainly true of the free-variable data, but the removal of the update flag does complicate the encoding of, for example, strictness and complexity information.

Note that re-ordering and replacing fields can be treated as combinations of deletions and additions. With the exception of production-rule deletion, the method outlined in section 5.2.1 should be followed, with the existing definitions serving as an additional guide.

Language restrictions

Alterations to the type rules will either tighten or relax the constraints placed on STG′ programs. The method is straightforward, involving only the addition or deletion of assertions, but the total effect can be considerable, as illustrated by the following examples.

The *APPLY-EXP′* rule, shown below, restricts the result of function application to algebraic values – a lifting algebraic type ($\texttt{data}\ \chi_{\texttt{Lift}}\ \alpha = \texttt{Lift}\ \alpha$) would have to be used to return functions or polymorphic variables:

$$\mathit{APPLY\text{-}EXP'}\quad \frac{\begin{array}{l} TE \overset{\text{var}}{\vdash} \mathit{var}_{\mathit{fun}} : \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \chi\ \alpha_1 \ldots \alpha_v \\ TE \overset{\text{atom}}{\vdash} \mathit{atom}_i : \tau_i \quad (0 \leq i \leq n) \end{array}}{TE \overset{\text{exp}}{\vdash} \mathit{var}_{\mathit{fun}}\ \mathit{atom}_1 \ldots \mathit{atom}_n : \chi\ \alpha_1 \ldots \alpha_v}$$

Relaxing any of the restrictions described in section 4.5.1 would require significant changes to be made to the operational semantics, and should therefore be avoided. The technique used in section 5.2.3 to improve the pipeline skeleton is a less powerful, yet workable, alternative. However, as an example, the following type rule removes the second restriction, allowing constructors to appear unsaturated:

$$\mathit{CONS\text{-}EXP'}\quad \frac{\begin{array}{l} (\mathit{cons}, (n, \sigma)) \in CE \\ TE \overset{\text{spec}}{\vdash} \sigma : \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \chi\ \pi_1 \ldots \pi_v \\ TE \overset{\text{atom}}{\vdash} \mathit{atom}_i : \tau_i \quad (0 \leq i \leq m \leq n) \end{array}}{TE \overset{\text{exp}}{\vdash} \mathit{cons}\ \mathit{atom}_1 \ldots \mathit{atom}_m : \tau_{m+1} \rightarrow \cdots \rightarrow \tau_n \rightarrow \chi\ \pi_1 \ldots \pi_v}$$

Denotational semantics

As outlined in section 4.7, the denotational semantics consists of three sets of definitions – the meta language, the domain equations, and the valuation functions – and changes can be made to each of these. Whichever group is targeted, the primary motivation behind any modification is likely to be concerned with the default order of evaluation. As an example, function application can be forced to model applicative-order reduction using the valuation function shown in figure 5.5. The transformation from a non-strict language into a strict language can be completed by redefining constructor application and the variable-binding functions. For the same reasons as outlined in section 5.2.2, non-determinism should not be introduced by this route.

$$\boxed{\begin{array}{lcl}
\mathcal{E}[\![\mathit{var}_{fun}\ \mathit{atom}_1 \ldots \mathit{atom}_n]\!]\ \rho & = & \mathit{let}\ \mathit{fun} = \rho\ \mathit{var}_{fun} \\
 & & \mathit{in\ case}\ \mathcal{A}\mathit{tom}[\![\mathit{atom}_1]\!]\ \rho\ \mathit{of} \\
 & & \qquad \bot \rightarrow \bot \\
 & & \qquad \epsilon_1 \rightarrow \mathit{case}\ \mathcal{A}\mathit{tom}[\![\mathit{atom}_2]\!]\ \rho\ \mathit{of} \\
 & & \qquad\qquad \bot \rightarrow \bot \\
 & & \qquad\qquad \epsilon_2 \rightarrow \ldots \\
 & & \qquad\qquad\quad \ddots \\
 & & \qquad\qquad\qquad \epsilon_n \rightarrow \ (\cdots((\mathit{fun}\ \epsilon_1)\ \epsilon_2)\cdots\epsilon_n)
\end{array}}$$

Figure 5.5: A valuation function for strict function application

5.2.5 Hybrid definitions

While it is possible to prototype a simple language using just one of the previous four methods, it is only by combining these strategies that more complex effects can be achieved. For example, the $\overline{\alpha_1 \rightarrow \alpha_2}$ boxed type and associated primitives (section 5.2.2) can increase the expressiveness of the `pipe` skeleton (section 5.2.1). Similarly, the `PID#` type provides support for a data-placement operation. However, apart from suggesting that each of the separate items be developed in isolation, hybrid definitions are beyond the scope of this thesis.

5.3 Language restrictions revisited

This section is concerned with the restrictions that need to be placed on any new language features, starting with an overview of the types of restraints that can be applied in section 5.3.1. The remaining two sections then look at extending the type-inference and free-variable algorithms presented in section 4.5.

5.3.1 Syntactic, algorithmic, and informal restrictions

There are three complementary approaches to limiting the set of valid language terms (DEFENCE IN DEPTH), and these are summarised below:

abstract syntax by controlling the way in which language terms can be constructed, it is possible to prevent undesirable phrases from being expressed. A good example of this is the abstract syntax of the POD type, defined in section 5.2.3, which requires no additional constraints to ensure that a given term is valid.

algorithmic restrictions often, complex restrictions cannot be enforced by the abstract syntax alone, and additional mechanised checks have to be made. The sequential STG′ language, for example, already makes use of free-variable and type-inference algorithms. In the context of parallel languages, new algorithmic techniques, such as shape analysis [Jay and Cockett, 1994], have great potential.

informal restrictions in situations where mechanisation is difficult, informal restrictions may be imposed on a language. Examples include requiring that: an operator is associative [Skillicorn, 1990, the reduce and directed-reduce operations, page 45]; or that the *head* function is never applied to an empty list [Hudak et al., 1992, `PreludeList`, page 106]. It is usually left to the programmer to ensure that these conditions are met – failure to do so may result in errors that are difficult for the run-time system to detect.

5.3.2 Type-inference rules

The type-inference algorithm presented in section 4.5 enforces the majority of restrictions required of well-formed sequential STG$'$ language programs. By extending the underlying rule set, most of the mundane restrictions that need to be placed on parallel extensions can also be checked. The rules contained in table 5.3 illustrate this last point, and also serve as an overview of the basic principles. While most cases should be straightforward, care has to be taken when dealing with constructs that force evaluation, as typified by the *par* combinator. Furthermore, extensions to the total environment, TE, may have to be made to accommodate new primitive types. Both of these issues are discussed in the following sections.

As a final note, by extending the type-inference algorithm it becomes obvious which constructs manipulate collections of values. Rather than using the standard `List` or `Tree` data types, these operations may benefit from the support of a specialised primitive type (see section 5.2.3). As an example, consider the *PIPE-SKELETON* rule from table 5.3.

Constrained types and the polymorphic *par* combinator

As mentioned in section 4.5.1, the STG machine has an *aggressive take* mechanism, such that a function can only be reduced when all of its arguments are present. This has obvious implications for any evaluation-forcing operation, which should thus only be used to reduce variables or expressions with algebraic or literal types. Even with a non-aggressive take, the return mechanism assumes that the exact type of the final result is known prior to evaluation – arbitrarily reducing polymorphic variables and expressions could cause problems. It follows that the *PIPE-SKELETON* rule from table 5.3 is too relaxed, while the *PAR-EXP* and *LETPAR-EXP* rules are suitably constrained.

Extending the type environment

With the introduction of new types or new binding mechanisms, the total environment, TE, may need to be extended. For instance, case alternatives for algebraic PODs only include the constructor tag, necessitating a POD-tag environment, PTE, of type $\chi_{\mathtt{tag}} \mapsto pod_1 \ldots pod_n$. The *ALG-PALTS* type rule, shown in figure 5.6, illustrates the use of this new entry. Note that it would be possible to simply use the constructor environment, CE, to store this information, and have the *ALG-PALT* rule access and return the required information.

With regards to animating these two rules, the *ALG-PALTS* needs to know the tag type before creating the local variable environment used by the *ALG-PALT* rule. As this information is inferred by the second rule, there is an obvious cyclic dependency. Two possible solutions exist: either Haskell's non-strict semantics can be used to resolve the conflict; or the first rule can peek at the left-hand side of the first alternative, and independently determine the type. The former strategy is the more elegant and concise, but will be sensitive to the strictness of all the routines which manipulate the environment.

1. simple types	*CURRENT-PID-EXP*	$\dfrac{}{TE \overset{\text{exp}}{\vdash} \texttt{currentPID} : PID\#}$
2. accessing the TE	*INDICES-EXP*	$\dfrac{(var, pod) \in TE}{TE \overset{\text{exp}}{\vdash} \overline{\text{INDICES}}\ var : \langle\!\langle \texttt{Int\#} \rangle\!\rangle}$
3. dependent types	*PAR-EXP*	$\dfrac{\begin{array}{l} TE \overset{\text{exp}}{\vdash} exp : \tau \\ (var, \chi\ \pi_1 \ldots \pi_v) \in TE \end{array}}{TE \overset{\text{exp}}{\vdash} \texttt{par}\ var\ exp : \tau}$
4. constrained types		
i. unboxed	*POD-ATOM*	$\dfrac{TE \overset{\text{atom}}{\vdash} atom : \nu}{TE \overset{\text{atom}}{\vdash} \langle\!\langle \ldots atom \ldots \rangle\!\rangle : \langle\!\langle \nu \rangle\!\rangle}$
ii. boxed	*PIPE-SKELETON*	$\dfrac{\begin{array}{c} \left(var_{pipe}, \overline{\pi_1 \rightarrow \pi_2}\right) \in TE \\ TE \overset{\text{exp}}{\vdash} exp : \texttt{List}\ \pi_1 \end{array}}{TE \overset{\text{skeleton}}{\vdash} \texttt{pipe}\ var_{pipe}\ exp : \texttt{List}\ \pi_2}$
5. extending the TE	*LETPAR-EXP*	$\dfrac{\begin{array}{l} TE \overset{\text{exp}}{\vdash} exp_{defn} : \chi\ \pi_1 \ldots \pi_v \\ LVE = \{var \mapsto \chi\ \pi_1 \ldots \pi_v\} \\ TE \overset{\rightarrow}{\oplus} LVE \overset{\text{exp}}{\vdash} exp : \tau \end{array}}{TE \overset{\text{exp}}{\vdash} \texttt{letpar}\ var = exp_{defn}\ exp : \tau}$
6. auxillary functions	see the *CASE-EXP* and *PROGRAM* rules in appendix D	

Table 5.3: A selection of type rules for parallel constructs

$$ALG\text{-}PALTS \quad \frac{\begin{array}{l}(\chi_{\mathtt{tag}}, pod_1 \ldots pod_n) \in PTE \\ LVE = \{var_1 \mapsto pod_1, \ldots, var_n \mapsto pod_n\} \\ TE \overset{\rightarrow}{\oplus} LVE \overset{\text{apalt}}{\vdash} apalt_1 : \chi_{\mathtt{tag}} \rightarrow pod \quad (1 \leq i \leq m)\end{array}}{TE \overset{\text{palts}}{\vdash} var_1 \ldots var_n \; apalt_1 \ldots apalt_m : \langle\!\langle \chi_{\mathtt{tag}} \rangle\!\rangle \; pod_1 \ldots pod_n \rightarrow pod}$$

$$ALG\text{-}PALT \quad \frac{\begin{array}{l}(cons, (0, \chi_{\mathtt{tag}})) \in CE \\ TE \overset{\text{exp}}{\vdash} exp : pod\end{array}}{TE \overset{\text{apalt}}{\vdash} \forall \; cons \rightarrow exp : \chi_{\mathtt{tag}} \rightarrow pod}$$

Figure 5.6: The *ALG-PALTS* and *ALG-PALT* type rules for PODs

5.3.3 Free variables

In general, developing the $\mathcal{FV}[\![]\!]$ rules for the parallel-language terms is straightforward – it is simply a matter of taking the union of the free variables of each non-terminal symbol that compose the construct:

$$\mathcal{FV}_{skeleton}[\![\mathtt{pipe}\ var_{pipe}\ exp]\!]\ g = \mathcal{FV}_{var}[\![var_{pipe}]\!]\ g \cup \mathcal{FV}_{exp}[\![exp]\!]\ g$$

The only complication involves binding operations, where care must be taken to filter out the local variables from the final answer:

$$\mathcal{FV}_{exp}[\![\mathtt{letpar}\ var = exp_{defn}\ exp]\!]\ g = \ \mathcal{FV}_{exp}[\![exp_{defn}]\!]\ g \cup (\mathcal{FV}_{exp}[\![exp]\!]\ g \setminus \{var\})$$

5.4 Denotational semantics and parallel languages

In the context of the prototyping framework, once developed, the denotational description of the entire language will serve as a guide during the development of the operational semantics, and as a reference model during the testing phase. In addition, the construction of the denotational semantics focuses the designers attention on the following areas:

order of evaluation for the semantics outlined in section 4.7, the order of evaluation is primarily determined by the occurrences of `case`, `letstrict`, and `let#` expressions. In the presence of scheduling constructs (see section 2.4.4) more complex orderings can be specified.

degree of evaluation in order to increase the amount of work performed by, for example, a `pipe` expression, it may be necessary to reduce its argument further than the usual head normal form. This behaviour should be reflected in the valuation function.

speculative evaluation and non-termination an expression which reduces to bottom, $\perp$, under the denotational semantics will probably fail to terminate in an actual implementation. The effect of non-termination on the run-time system can be grossly specified by the denotational model.

non-determinism despite the potential loss of referential transparency, the introduction of non-determinism can be useful, particularly when providing access to operational parameters, such as the system load or the processor identifier.

run-time errors and exception handling it is not uncommon for languages to support the use of informal restrictions by providing a primitive similar to Haskell's `error` operator [Hudak et al., 1992, pages 68 and 88]. Non-fatal errors can be supported through the use of user-level exception handlers.

Each of these issues, along with a brief summary of the different strategies, are explored in the following sections.

5.4.1 Order of evaluation

As the underlying mathematics supports no notion of 'order of evaluation', it seems unlikely that the denotational semantics can be applied to this problem. However, in the absence of side effects, the exact interleaving of computations [Hooman, 1991, sections 2.2 and 4.1] is not important, and only the dependencies need to be expressed [Bloss and Hudak, 1988]. Consider the following example:

$$\begin{array}{rcllcl} \mathcal{E}\llbracket \mathsf{seq}\ var\ exp \rrbracket\ \rho & = & case\ (\rho\ var)\ of & \bot & \rightarrow & \bot \\ & & & \epsilon & \rightarrow & \mathcal{E}\llbracket exp \rrbracket\ \rho \end{array}$$

Remembering that bottom, $\bot$, equates to non-termination, this valuation function captures the expected behaviour, i.e. seq x y will only return the value y if x represents a finite computation. Similarly, the par combinator can be modelled as follows (ignoring termination properties):

$$\mathcal{E}\llbracket \mathsf{par}\ var\ exp \rrbracket\ \rho \ = \ \mathcal{E}\llbracket exp \rrbracket\ \rho$$

Unfortunately, using this strategy, no satisfactory definition can be arrived at for a passive `wait` x y expression [Goldberg, 1988a, section 3.2], i.e. one which cannot initiate the reduction of x. If a seq-style valuation function is used, the possibility that x is never reduced is not expressed. However, in all of of the examples presented by Goldberg, the `waits` are always matched by preceding par operations – if this restriction could be enforced by the language, the proposed description would be valid.

While obtaining an accurate model is desirable, it is not essential, as the operational semantics is a better medium for expressing these concerns. As long as the denotational model is under constrained, i.e. the denotational reading is always as well defined as the operational result, the model can still be used for test purposes.

As a final note, it is expected that the number of constructs which change the default (non-strict) order of evaluation will be small. It was therefore decided not to use the continuation-passing style of denotational semantics [Raskovsky and Collier, 1980; Sethi, 1982; Schmidt, 1986, chapter 9], which would, arguably, make the descriptions harder to follow in most cases.

5.4.2 Degree of evaluation

While testing for bottom can be used to indicate that a value will be reduced to at least head normal form, constructs that manipulate algebraic data types may need to express more complex requirements [Burn, 1991, figure 5.1, page 114]. For example, consider the following definition of the pipe skeleton:

$$\begin{array}{lcl}
\mathit{Skeleton}\llbracket \texttt{pipe}\ \mathit{var}_{\mathit{pipe}}\ \mathit{exp}\rrbracket\ \sigma & = & \mathit{let}\ \mathit{function} = \sigma\ \mathit{var}_{\mathit{pipe}},\ \mathit{arguments} = \mathcal{E}\llbracket \mathit{exp}\rrbracket\ \rho \\
 & & \mathit{in}\ \xi_{\infty}^{(\texttt{List}\ \pi)}\ (\mathit{map}\ \mathit{function}\ \mathit{arguments}) \\
 & & \\
 & & \xi_{\infty}^{(\texttt{List}\ \pi)}\ ::\ \mathbf{Val} \rightarrow \mathbf{Val} \\
 & & \\
\xi_{\infty}^{(\texttt{List}\ \pi)}\ \epsilon & = & \mathit{case}\ \epsilon\ \mathit{of}\ \begin{array}[t]{lcl} [\,]\ \bot & \rightarrow & \bot \\ [\,]\ \langle\texttt{Nil}\rangle & \rightarrow & \langle\texttt{Nil}\rangle \\ [\,]\ \langle\texttt{Cons}, x, xs\rangle & \rightarrow & \langle\texttt{Cons}, x, \xi_{\infty}^{(\texttt{List}\ \pi)}\ xs\rangle \end{array}
\end{array}$$

The application of the $\xi_{\infty}^{(\texttt{List}\ \pi)}$ function [Burn, 1991, section 1.2, page 7] to the pipe's input, forces reduction to spine normal form [Kewley and Glynn, 1990, page 330]. For each data type there is potentially a large number of reduction strategies [Hammond, 1991, "the twenty-four names of Cons", section 8.3], and it may be worth considering automatically deriving the *evaluation transformers* from the type declarations.

5.4.3 Speculative evaluation and non-termination

When threads are used to only reduce essential expressions, non-termination of an individual thread is not a problem as the entire computation will, by definition, also fail to terminate. However, by permitting speculative evaluation [Mattson Jr., 1993a, chapter 3], it is possible that a non-essential thread may consume sufficient resources so as to affect the final result. Consider the following valuation functions:

$$\begin{array}{rcl}
\mathcal{E}\llbracket \texttt{par}\ \mathit{var}\ \mathit{exp}\rrbracket\ \rho & = & \mathit{case}\ (\rho\ \mathit{var})\ \mathit{of}\ \begin{array}[t]{lcl} \bot & \rightarrow & \bot \\ \epsilon & \rightarrow & \mathcal{E}\llbracket \mathit{exp}\rrbracket\ \rho \end{array} \\
\mathcal{E}\llbracket \texttt{speculate}\ \mathit{var}\ \mathit{exp}\rrbracket\ \rho & = & \mathcal{E}\llbracket \mathit{exp}\rrbracket\ \rho
\end{array}$$

Based on these definitions, the par combinator can only be used to reduce either essential values, or expressions which are known to terminate. The speculate combinator is less constrained, in that it can be used to evaluate any expression. The cost of this increased expressiveness is that the run-time system must use a *fair* scheduling algorithm, and be capable of garbage collecting unnecessary threads [Mattson Jr., 1993a, section 7.4.1].

5.4.4 Non-determinism

In general, the introduction of non-determinism results in the loss of referential transparency, and hence invalidates a wide range of compilation techniques [Santos, 1995]. There are only two situations where this loss could be justified: when providing access to run-time values, such as the current processor identifier; and allowing threads to interact non-trivially through the use of side effects. Both cases are considered in the following sections, although [Dennis et al., 1995] is recommended as a succinct review of the situation.

Accessing operational parameters

By providing access to certain run-time parameters, including current workloads and the local-processor identifier, it is possible for a program to adapt its behaviour in the hope of improving efficiency [Hudak, 1991, section 5.4]. However, such values are inherently non-deterministic, and therefore complicate the development of a denotational semantics. The remainder of this section looks at a number of different ways of incorporating non-determinism, including the use of *powerdomains*.

The easiest way to handle non-determinism is to ignore it completely, as is done below:

$$\mathcal{E}[\![\texttt{currentPID}\#]\!]\ \rho \quad = \quad 42\#$$

The only time that this approach is defensible is when the resulting values can only affect the operational behaviour of the program. This is often achieved by imposing type restrictions on the language (see sections 5.2.3 and 5.3.2). Mirani and Hudak [1995, section 3] take this idea one step further by wrapping all such values inside an operating-system *monad* [Peyton Jones and Wadler, 1993]. Both of these techniques can be used in conjunction with the other methods described in this section.

A more satisfactory solution is to accurately model the parameter in question. For example, Hudak [1986] has used this technique to develop a semantics for a simple language which includes the exp_1 on exp_2, and $self$ expressions (the latter is similar to the `currentPID#` construct). The formal arguments of all valuation functions are extended to include a processor identifier, which represents the location of the current computation:

$$\begin{array}{lcl} \mathcal{E}[\![exp\ \mathsf{on}\ pid]\!]\ \rho\ current\_pid & = & case\ (\mathcal{E}[\![pid]\!]\ \rho\ current\_pid)\ of \\ & & \quad \bot \qquad\quad \rightarrow\ \bot \\ & & \quad new\_pid\ \rightarrow\ \mathcal{E}[\![exp]\!]\ \rho\ new\_pid \end{array}$$

The $\mathcal{P}rogram[\![]\!]$ rule provides the initial value of the *current_pid* parameter.

Incorporating `randomPID#`-style expressions, or attempting to model data migration, is more problematic. Consider the implicit specification of the location using a function of type PID# → PID# → PID# (at run time, the current processor identifier and a random processor identifier will be supplied as arguments.) This removes the need for both the `currentPID#` and `randomPID#` constructs, as well as simplifying the denotational semantics:

$$\begin{array}{lcl} \mathcal{E}[\![exp_1\ \mathsf{on}\ exp_2]\!]\ \rho & = & let\ location\_function = \mathcal{E}[\![exp_2]\!]\ \rho \\ & & in \begin{cases} \bot, & \text{if } \bot \in \{location\_function\ i\ j \mid \forall i,j \in\ PID\#\} \\ \mathcal{E}[\![exp_1]\!]\ \rho, & \text{otherwise} \end{cases} \end{array}$$

It would obviously be unrealistic to check that each location function meets the above requirement.

If none of the above techniques is applicable, it will be necessary to use a *powerdomain* [Schmidt, 1986, section 12.1, page 275], replacing all occurrences of the **Val** domain with **P**(**Val**), where **P**(D) represents the powerdomain builder. In addition, each rule will have to be updated to handle multiple values, using either Haskell's list comprehensions [Hudak et al., 1992, section 3.10, page 16] or a monad [Wadler, 1992, "Non-deterministic choice", section 2.7] to handle the multiple values, and the order of evaluation re-examined. As a small consolation, the animation of the resulting semantics can be straightforward.

Side effects and thread interaction

The development of denotational semantics for sequential side-effecting language is well understood, and only requires the correct threading of the environment [Schmidt, 1986, chapter 7]. However, the combination of side effects and parallelism [Barth, Nikhil and Arvind, 1991; Jones and Hudak, 1993, section 4.3] generally implies complex descriptions, based on a large number of assumptions and limitations – consider, for example, the semantics presented by Hooman [1991] for an Occam-style language. Unless something can be done to limit the possible interactions, it is recommended that this stage of the design process should simply be missed out. Hudak [1987, section 2.1], for example, only allows destructive array updating if it can be proved (by the compiler) that this will not break the sequential semantics.

5.4.5 Run-time errors and exception handling

Generally, if an informal restriction is violated, the current thread should be terminated (and possibly the entire computation). Assuming that the failure can be detected, the following construct provides the necessary support:

$$\boxed{\mathcal{E}[\![\texttt{error}\ exp]\!]\ \rho \quad = \quad \bot}$$

Unfortunately, using the **Val** domain definition from section 4.7.1, it is not possible to model low-level errors, including division by zero, by returning bottom. Furthermore, due to resource limitations, for example, it is possible that a valid computation will fail to terminate when run on a computer. This kind of error also cannot be easily modelled by the denotational description.

The provision of exception handling mechanisms, as used by Hammond [1991, section 2.4.1], is a more flexible approach to the same problem, in that it can be used to model non-fatal errors without resorting to the use of algebraic data types.

5.4.6 A selection of bottoms

In the previous sections, the bottom element, $\bot$, has been used in a number of different roles:

- to model scheduling dependencies.
- to force evaluation beyond the usual head normal form.
- to represent non-termination, and thus limit the applicability of a spark or location construct.
- to indicate that a fatal error has occurred.

Obviously, the animation of the denotational semantics will only be able to handle the last type of bottom (a fatal error) in a non-trivial manner – all others will result in the non-termination of the implementation.

As a final note, when testing for bottom, extra care must be taken to avoid non-monotonicity [Schmidt, 1986, pages 112–113]. For example, the following valuation function would invalidate the entire semantics:

$$\boxed{\begin{array}{rclrcl} \mathcal{E}[\![\texttt{is\_bottom}\ exp\ exp_{\bot}\ exp_{\not\bot}]\!]\ \rho & = & case\ (\mathcal{E}[\![exp]\!]\ \rho)\ of & \bot & \rightarrow & \mathcal{E}[\![exp_{\bot}]\!]\ \rho \\ & & & \epsilon & \rightarrow & \mathcal{E}[\![exp_{\not\bot}]\!]\ \rho \end{array}}$$

5.5 Summary

In this chapter a number of different syntax-driven guidelines for introducing parallelism into the STG$'$ language have been proposed, covering: the additions of new production rules, primitive functions, and primitive types; and the modification of existing rules, whether they be taken from the abstract syntax, the type-inference rules, or the denotational semantics. Furthermore, the application of language restrictions and denotational semantics to the development of parallel languages has also been discussed.

Chapter 6

Managing parallelism – operational models

6.1 Introduction

This chapter discusses the development of an operational description to augment the denotational semantics of the parallel STG$'$ language (see chapter 5). The STG machine provides the basic recipe, into which the parallel ingredients, including threads, messages, and shared memory, are added. To facilitate testing and debugging, the animation of the model (which is essentially a state-transition system) is also considered. The final description is then used by chapter 8 to provide the foundation upon which the compilation system is built.

Sections 6.2 and 6.3 are concerned with the introduction of parallelism into an operational model – the former develops a general framework to work within, while the latter deals with the issues specific to a parallel STG machine. The implications of the STG$'$ language manipulations described in the previous chapter are then considered in section 6.4. The animation and testing of the resulting state machines are discussed in section 6.5, before the chapter is summarised in section 6.6.

6.2 Parallelism and the STG machine

This section explores the use of state transition systems to model modern multi-processor systems. Section 6.2.1 discusses the gross representation of the processing and communication elements. This model is then refined to explicitly include the notions of time and inter-processor synchronisation in sections 6.2.2 and 6.2.3 respectively. Shared-memory and message-passing abstractions are then examined in greater detail in sections 6.2.4 and 6.2.5.

6.2.1 One abstract machine or many?

When modelling a parallel or concurrent system, the possible interactions between the component processors can either be explicitly or implicitly specified. As an example of the first approach, both Peyton Jones, Gordon and Finne [1996, section 6.2] and Ostheimer [1993, section 3.4, page 39] use the π-calculus [Milner, 1993] as the underlying formalism. The following congruence and structural rule controls the “reactions” (the number of

$$(\textsc{init}) \quad \boxed{\mathit{init} \Longrightarrow (1, \mathit{init}_{\mathrm{P}}\ P_1, \ldots, \mathit{init}_{\mathrm{P}}\ P_n)\ (\mathit{init}_{\mathrm{S}}\ S)}$$

$$(\textsc{step}) \quad \boxed{\begin{array}{l} \mathit{step}\ (i, P_1, \ldots, P_i, \ldots, P_n)\ S \Longrightarrow (i', P_1, \ldots, P_i', \ldots, P_n)\ (\mathit{step}_{\mathrm{S}}\ S') \\ \text{where } i' = 1 + (i \bmod n) \text{ and } (P_i', S') = \mathit{step}_{\mathrm{P}}\ (P_i, S) \end{array}}$$

$$(\textsc{final}) \quad \boxed{\mathit{final}\ (i, P_1, \ldots, P_n)\ S \Longrightarrow \mathit{final}_{\mathrm{P}}\ P_1 \wedge \cdots \wedge \mathit{final}_{\mathrm{P}}\ P_n \wedge \mathit{final}_{\mathrm{S}}\ S}$$

Figure 6.1: A simple processor framework for the parallel STG machine

processors is unbounded):

$$\begin{array}{lrcl} (\textsc{par}) & P \mid Q & \rightarrow & P' \mid Q, \text{ if } P \rightarrow P' \\ (\textsc{comm}) & P \mid Q & \equiv & Q \mid P \\ (\textsc{assoc}) & P_1 \mid (P_2 \mid P_3) & \equiv & (P_1 \mid P_2) \mid P_3 \end{array}$$

On the other hand, the ν-STG machine [Hwang and Rushall, 1992] and the abstract machine for pH [Aditya et al., 1995] are defined in terms of a single processor, with each reaction rule specifying only one half of a processor-processor interaction. In fact, the data-parallel STG machine [Hill, 1994, rules 18, chapter 6, page 123] specifies all forms of parallelism in terms of auxiliary functions and set comprehensions.

While the former approach is undeniably superior from a theoretical standpoint, the latter boasts a greater number of relevant examples and is, arguably, less complex, making it the method of choice. However, to both simplify the animation (see section 6.5) and to clearly identify the primary communication mechanisms, an explicit framework will be assumed throughout this section. The framework shown in figure 6.1 will be used as the start point, and will be subsequently refined as the need arises.

This model comprises three distinct state-transition systems, the abstract states of which are: S, the communication system; P, a single processor; and $(i, P_1, \ldots, P_n)\ S$, an ensemble. Each has its own set of *init*, *final*, and *step* operators for creating the initial state, testing for a final state, and performing one state transition respectively. Note that the processor index, i, records the identity of the processor to be stepped on the next transition. For most applications this round-robin scheduling is overly simplistic, and section 6.2.2 refines the model to allow the use of timing information to control the order of transitions.

To demonstrate the basic principles of the framework, figure 6.2 shows the INIT, STEP, and FINAL reduction rules describing a two processor ping-pong system [Booth et al., 1997]. The first processor, P_0, pings its neighbour, waits for a reply, and then re-starts the cycle. The second processor, P_1, is its dual, and waits to be pinged before ponging P_0. The state diagram of this system is shown in figure 6.3, where the diagonal lines denote communication between the two processors. As specified by the FINAL$_{\mathrm{P}}$ and FINAL$_{\mathrm{S}}$ rules,

$$(\textsc{init}_{\mathrm{P}}) \quad \boxed{\begin{array}{lcl} init_{\mathrm{P}}\ P_0 & \Longrightarrow & Ping \\ init_{\mathrm{P}}\ P_1 & \Longrightarrow & WaitForPing \end{array}}$$

$$(\textsc{init}_{\mathrm{S}}) \quad \boxed{init_{\mathrm{S}}\ S \quad \Longrightarrow \quad Nothing}$$

$$(\textsc{step}_{\mathrm{P}}) \quad \boxed{\begin{array}{c} step_{\mathrm{P}}\ (Ping, S) \Longrightarrow (WaitForPong, HavePinged) \\ step_{\mathrm{P}}\ (Pong, S) \Longrightarrow (WaitForPing, HavePonged) \\ \\ step_{\mathrm{P}}\ (WaitForPong, HavePonged) \Longrightarrow (Ping, Nothing) \\ step_{\mathrm{P}}\ (WaitForPing, HavePinged) \Longrightarrow (Pong, Nothing) \\ \\ step_{\mathrm{P}}\ (P, S) \Longrightarrow (P, S) \end{array}}$$

$$(\textsc{step}_{\mathrm{S}}) \quad \boxed{step_{\mathrm{S}}\ S \Longrightarrow S}$$

$$(\textsc{final}_{\mathrm{P}}) \quad \boxed{final_{\mathrm{P}}\ P \Longrightarrow false}$$

$$(\textsc{final}_{\mathrm{S}}) \quad \boxed{final_{\mathrm{S}}\ S \Longrightarrow false}$$

Figure 6.2: Transition rules for a simple ping-pong system

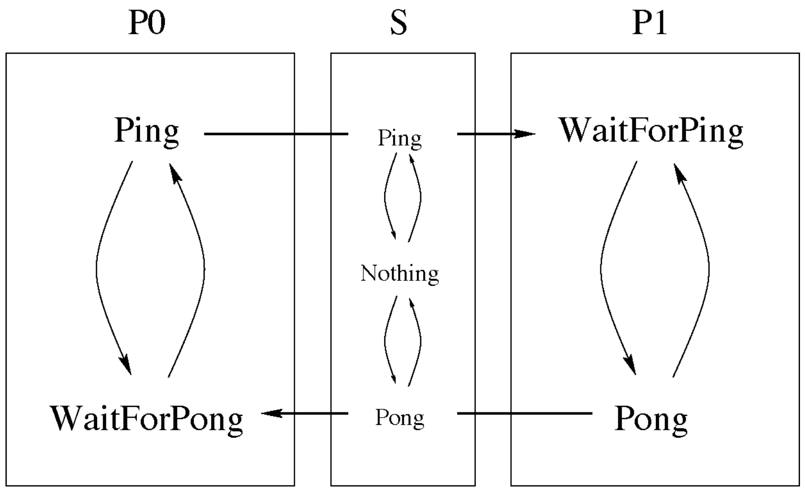


Figure 6.3: The state-transition diagram for the ping-pong system

$$(\textsc{init})\quad \boxed{init \Longrightarrow (init_{\mathrm{P}}\ P_1, \ldots, init_{\mathrm{P}}\ P_n)\ (init_{\mathrm{S}}\ S)}$$

$$(\textsc{step})\quad \boxed{\begin{array}{l} step\ \ (P_1,\ \ldots,\ \ P_i,\ \ \ldots,\ \ P_n) \qquad\qquad S \\ \Longrightarrow\ \ (P_1,\ \ldots,\ \ P_i',\ \ \ldots,\ \ P_n)\ \ (step_{\mathrm{S}}\ S') \\ \text{where } (P_i', S') = step_{\mathrm{P}}\ (P_i, S) \\ \text{such that } \forall\ j \in \{1, \ldots, n\} \bullet\ local\_time\ P_i \leq local\_time\ P_j \end{array}}$$

$$(\textsc{final})\quad \boxed{final\ (P_1, \ldots, P_n)\ S \Longrightarrow final_{\mathrm{P}}\ P_1 \wedge \cdots \wedge final_{\mathrm{P}}\ P_n \wedge final_{\mathrm{S}}\ S}$$

Figure 6.4: Explicitly modelling time in the processor framework

the system never terminates and simply repeats the cycle shown below:

0.	(0,	INIT$_{\mathrm{P}}$ P_0,	INIT$_{\mathrm{P}}$ P_1)	INIT$_{\mathrm{S}}$ S	$\Longrightarrow$
1.	(0,	*Ping*,	*WaitForPing*)	*Nothing*	$\Longrightarrow$
2.	(1,	*WaitForPong*,	*WaitForPing*)	*HavePinged*	$\Longrightarrow$
3.	(0,	*WaitForPong*,	*Pong*)	*Nothing*	$\Longrightarrow$
4.	(1,	*WaitForPong*,	*Pong*)	*Nothing*	$\Longrightarrow$
5.	(0,	*WaitForPong*,	*WaitForPing*)	*HavePonged*	$\Longrightarrow$
6.	(1,	*Ping*,	*WaitForPing*)	*Nothing*	$\Longrightarrow$
7.	(0,	*Ping*,	*WaitForPing*)	*Nothing*	$\Longrightarrow$
			⋮		

Notice that nothing happens to P_0, P_1, and S between steps 3 and 4 – P_0 is waiting for a pong which has not yet been sent. A similar situation occurs between steps 6 and 7. While such wait reductions are acceptable for this small example, they can quickly obscure the other reductions as the number of processors increases. Section 6.2.3 addresses this problem by introducing the concept of busy, waiting, and stopped processors.

6.2.2 Abstractions of time

The operational models developed so far make no reference to time, and hence cannot encode either the expected run time of a rule, or the temporal relationships between events [Sadri, 1987, section 2, page 122]. Other situations which require an explicit model of time include time-stamping messages, specifying a time-out period when waiting for an *Ack* message (section 9.3.2), or implementing a heart-beat algorithm [Andrews, 1991, section 4, pages 63–68]. To incorporate time into the processor framework presented in section 6.2.1, each P is extended to include a local clock, t_{local}, and the STEP rule has to be changed as shown in figure 6.4.

This suggests that a time-aware state-transition system could be used to estimate the run time of a physical system. Hehner [1994, section 12.4, page 195] summarises this line of reasoning, as well as identifying the main problem:

> "To obtain the real execution time, just insert time increments as appropriate. Of course, this requires intimate knowledge of the implementation, both hardware and software; there's no way to avoid it."

Furthermore, the physical implementations can themselves be unpredictable. Hammond, Burn and Howe [1994, figures 1 and 2] demonstrate that a small variation in the size of

$$(\textsc{init}_\mathrm{P})\quad \boxed{\begin{array}{lcl} init_\mathrm{P}\ P_0 & \Longrightarrow & Ping_{t_{local}=0} \\ init_\mathrm{P}\ P_1 & \Longrightarrow & WaitForPing_{t_{local}=0} \end{array}}$$

$$(\textsc{init}_\mathrm{S})\quad \boxed{init_\mathrm{S}\ S \ \Longrightarrow\ Nothing}$$

$$(\textsc{step}_\mathrm{P})\quad \boxed{\begin{array}{c} step_\mathrm{P}\ (Ping_t, S) \Longrightarrow (WaitForPong_{t+10}, NewPing\ t) \\ step_\mathrm{P}\ (Pong_t, S) \Longrightarrow (WaitForPing_{t+10}, NewPong\ t) \\ \\ step_\mathrm{P}\ (WaitForPong_t, HavePonged\ t_{recv}) \text{ such that } t \geq t_{recv} \\ \Longrightarrow (Ping_{t+10}, Nothing) \\ step_\mathrm{P}\ (WaitForPing_t, HavePinged\ t_{recv}) \text{ such that } t \geq t_{recv} \\ \Longrightarrow (Pong_{t+10}, Nothing) \\ \\ step_\mathrm{P}\ (P_t, S) \Longrightarrow (P_{t+1}, S) \end{array}}$$

$$(\textsc{step}_\mathrm{S})\quad \boxed{\begin{array}{lcl} step_\mathrm{S}\ (NewPing\ t) & \Longrightarrow & HavePinged\ (t+100) \\ step_\mathrm{S}\ (NewPong\ t) & \Longrightarrow & HavePonged\ (t+100) \\ step_\mathrm{S}\ S & \Longrightarrow & S \end{array}}$$

$$(\textsc{final}_\mathrm{P})\quad \boxed{final_\mathrm{P}\ P \Longrightarrow false}$$

$$(\textsc{final}_\mathrm{S})\quad \boxed{final_\mathrm{S}\ S \Longrightarrow false}$$

Figure 6.5: Transition rules for a time-aware ping-pong system

the dynamic heap can give rise to a 50% difference in uniprocessor performance (this was attributed to a cache conflict between the argument stack and instruction stream). With regards to parallel systems, Trinder et al. [1996, section 4.1], commenting on the average speedup observed by the GUM system, note that:

> "There is a degree of chaos in the results, since a single change in the placement of a spark at runtime can affect the overall runtime."

In summary, to achieve any degree of accuracy, the ***level of detail*** required [Jain, 1991, section 5.2, pages 66–67] would render the rule set worthless as a design tool. It is therefore assumed that each rule takes either one, ten, one hundred, or one thousand time units to complete. This still allows a certain degree of performance debugging without overburdening the design. It also guarantees that any estimate is treated with caution.

To illustrate the use of the extended framework, figure 6.5 shows the new rules for the ping-pong system presented in the previous section. The processors' local clocks appear as subscripts to the original processor states, and the times associated with each operation are shown in figure 6.6. The state-transition diagram for the new system is similar to that of the original (see figure 6.3). The first cycle of reductions is shown in figure 6.2.2 (disregarding the majority of wait reductions). Notice that the wait reductions account for over ninety percent of the total number of reductions. Not only is this distracting when examining the reduction steps, but can cause serious performance problems when it comes to animating the system. This problem is addressed in the following section.

operation	time
WaitForPing, no ping available	1
WaitForPong, no pong available	1
Ping	10
Pong	10
WaitForPing, ping available	10
WaitForPong, pong available	10
communication delay	100

Figure 6.6: Time costs for the ping-pong system

time	system state			
INIT	(INIT$_{\mathrm{P}}$ P_0,	INIT$_{\mathrm{P}}$ P_1)	INIT$_{\mathrm{S}}$ S	$\Longrightarrow$
0	$(\mathit{Ping}_0,$	$\mathit{WaitForPing}_0)$	*Nothing*	$\Longrightarrow$
0	$(\mathit{WaitForPong}_{10},$	$\mathit{WaitForPing}_0)$	*NewPing* 0	$\Longrightarrow$
0	$(\mathit{WaitForPong}_{10},$	$\mathit{WaitForPing}_0)$	*HavePinged* 100	$\Longrightarrow$
1	$(\mathit{WaitForPong}_{10},$	$\mathit{WaitForPing}_1)$	*HavePinged* 100	$\Longrightarrow$
⋮				
100	$(\mathit{WaitForPong}_{101},$	$\mathit{WaitForPing}_{100})$	*HavePinged* 100	$\Longrightarrow$
101	$(\mathit{WaitForPong}_{101},$	$\mathit{Pong}_{110})$	*Nothing*	$\Longrightarrow$
⋮				
110	$(\mathit{WaitForPong}_{111},$	$\mathit{Pong}_{110})$	*Nothing*	$\Longrightarrow$
111	$(\mathit{WaitForPong}_{111},$	$\mathit{WaitForPing}_{120})$	*NewPong*	$\Longrightarrow$
111	$(\mathit{WaitForPong}_{111},$	$\mathit{WaitForPing}_{120})$	*HavePonged* 210	$\Longrightarrow$
⋮				
210	$(\mathit{WaitForPong}_{210},$	$\mathit{WaitForPing}_{210})$	*HavePonged* 210	$\Longrightarrow$
211	$(\mathit{Ping}_{220},$	$\mathit{WaitForPing}_{210})$	*Nothing*	$\Longrightarrow$
⋮				

Figure 6.7: State transitions for the time-aware ping-pong system

$$\text{(INIT)} \quad \boxed{init \Longrightarrow (init_{\mathrm{P}}\ P_1, \ldots, init_{\mathrm{P}}\ P_n)\ (init_{\mathrm{S}}\ S)}$$

$$\text{(STEP)} \quad \boxed{\begin{array}{l} step\ \ (P_1,\ \ldots,\ \ P_i,\ \ \ldots,\ \ P_n) \qquad S \\ \Longrightarrow\ \ (P_1,\ \ldots,\ \ P_i',\ \ \ldots,\ \ P_n)\ \ (step_{\mathrm{S}}\ S') \\ \text{where}\ \ (P_i', S') \ = \ step_{\mathrm{P}}\ (P_{i,t_{local}=t_{next}}, S) \\ \qquad\quad t_{next} \quad\ = \ next\_time\ (P_i, S) \\ \text{such that } t_{next} < \infty \wedge \forall\ j \in \{1, \ldots, n\} \bullet\ t_{next} \leq next\_time\ (P_j, S) \\ \\ step\ \ (P_1, \ldots, P_n)\ S \Longrightarrow (P_1, \ldots, P_n)\ (step_{\mathrm{S}}\ S) \end{array}}$$

$$\text{(NEXT)} \quad \boxed{next\_time\ (P, S) \ \Longrightarrow \begin{cases} local\_time\ P, & \text{if } is\_active \ \ P \\ comms\_time\ (P, S), & \text{if } is\_waiting\ P \\ \infty, & \text{if } is\_stopped\ P \end{cases}}$$

$$\text{(FINAL)} \quad \boxed{final\ (P_1, \ldots, P_n)\ S \Longrightarrow final_{\mathrm{P}}\ P_1 \wedge \cdots \wedge final_{\mathrm{P}}\ P_n \wedge final_{\mathrm{S}}\ S}$$

Figure 6.8: Incorporating processor states into the processor framework

6.2.3 Inter-processor synchronisation

Despite extending the framework to include an explicit model of time, it is still not yet suitable for modelling a parallel STG machine – the number of wait reductions becomes a significant problem as a system increases in complexity. The first step to eliminating this problem is to note that a a processor can be in one of three states:

state	description
active	the processor is busy performing useful work.
waiting	the processor cannot continue until it either receives data via the communication system or it times out.
stopped	the processor has completed its task and will play no further part in the global computation.

Within the existing framework, the processor's local clock, t_{local}, is used to decide which processor to step next. While this is correct for active processors, it is often too early for waiting processors, and they then have to undergo a number of wait reductions to bring them to a time at which they can interact with the communication system. This situation is avoided in the framework shown in figure 6.8. The *next_time* function returns the earliest time at which a processor can become active based on the current state of the processor and of the communication system. Before a processor is stepped, its local clock is set to this new time t_{next}, i.e. $P_{i,t_{local}=t_{next}}$. Notice also that the communication system may now have to be stepped independently of the processors.

Returning again to the ping-pong example, the system can be upgraded to use the new processor states by defining the functions *is_active*, *is_waiting*, *is_stopped*, and *comms_time*, shown below:

$$\text{(IS\_ACTIVE)} \quad \boxed{\begin{array}{lcl} is\_active\ (WaitForPing, S) & \Longrightarrow & false \\ is\_active\ (WaitForPong, S) & \Longrightarrow & false \\ is\_active\ (P, S) & \Longrightarrow & true \end{array}}$$

$$\text{(IS\_WAITING)} \quad \boxed{is\_waiting\ (P, S) \ \Longrightarrow\ \neg is\_active\ (P, S)}$$

$$\text{(IS\_STOPPED)}\quad \boxed{\mathit{is\_stopped}\ (P, S) \implies \mathit{false}}$$

$$\text{(COMMS\_TIME)}$$

$$\boxed{\begin{array}{lll}
\mathit{comms\_time}\ (\mathit{WaitForPing}_{t_{local}}, \mathit{HavePinged}\ t_{recv}) & \implies & \mathit{max}(t_{local}, t_{recv}) \\
\mathit{comms\_time}\ (\mathit{WaitForPong}_{t_{local}}, \mathit{HavePonged}\ t_{recv}) & \implies & \mathit{max}(t_{local}, t_{recv}) \\
\mathit{comms\_time}\ (P, S) & \implies & \infty
\end{array}}$$

The reduction cycle is as before, but with all of the trivial waiting reductions removed. The ping-pong example can now be easily extended to include time-outs, as shown in figure 6.9. If either a ping or a pong is lost, then one of the processors will time-out and therefore stop. This will then lead to the other processor stopping, as it will also time-out waiting for the reply to the lost message. The final state is reached as soon as both processors have stopped.

The following two sections look at the modelling of shared-memory and message-passing architectures within the processor framework.

6.2.4 Shared memory

On one level, modelling shared-memory architectures is straightforward, requiring only the specification of the hierarchy of components:

P		S		
heap	task pool	heap	task pool	example systems
h_{local}	wp_{local}	none	none	GUM [Trinder et al., 1996]
h_{local}	wp_{local}	h_{global}	wp_{global}	GRIP [Peyton Jones et al., 1987]
none	none	$h_1 \cdots h_n$	$wp_1 \cdots wp_n$	BBN Haskell [Mattson Jr., 1993a]
none	none	h_{global}	wp_{global}	$\langle v, G\rangle$-machine [Augustsson and Johnsson, 1989]

Shared-memory components are mainpulated in exactly the same manner as with the sequential STG machine. The following example illustrates how local heap allocation is preferred in BBN Haskell:

$$\boxed{\begin{array}{ll}
\mathit{Eval}\ (\texttt{let}\ (\mathit{var} = \mathit{vs}\ \pi\ \mathit{xs} \rightarrow \mathit{exp}_{rhs})\ \mathit{exp})\ \rho & \mathit{as}\ \mathit{rs}\ \mathit{us}\ h_1 \cdots h_i \cdots h_n\ \mathit{wps}\ \sigma\ t \\
\implies \mathit{Eval}\ \mathit{exp}\ \rho[\mathit{var} \mapsto a] & \mathit{as}\ \mathit{rs}\ \mathit{us}\ h_1 \cdots h_i' \cdots h_n\ \mathit{wps}\ \sigma\ t + 10 \\
\text{where } i \ = \ \mathit{processor\_id} & \\
\qquad h_i' \ = \ h_i[a \mapsto (\mathit{vs}\ \pi\ \mathit{xs} \rightarrow \mathit{exp}_{rhs})(\rho\ \mathit{vs})] &
\end{array}}$$

In this example, the $\mathit{step}_{\mathrm{P}}$ prefix has been dropped and $P \equiv (\mathit{code}, \mathit{as}, \mathit{rs}, \mathit{us}, \sigma, t_{local})$, and $S \equiv (\mathit{hp}_1, \ldots, \mathit{hp}_n, \mathit{wps})$. Typically, with shared-memory systems, it takes longer to access memory on a remote machine than it does to access local memory. Again, this is demonstrated using the BBN Haskell system:

$$\boxed{\begin{array}{l}
\qquad \mathit{Enter}\ a \quad \mathit{as} \quad \mathit{rs} \quad \mathit{us} \quad h_1 \cdots h_n \quad \mathit{wps} \quad \sigma \quad t \\[1ex]
\text{such that } \exists j \bullet h_j[a \mapsto (\mathit{vs}\ \pi\ \mathit{xs} \rightarrow \mathit{exp})\ \mathit{ws}_f] \text{ and } \mathit{length}(\mathit{as}) \geq \mathit{length}(\mathit{xs}) \\[1ex]
\implies \mathit{Eval}\ \mathit{exp}\ \rho \quad \mathit{as}' \quad \mathit{rs} \quad \mathit{us} \quad h_1 \cdots h_n \quad \mathit{wps} \quad \sigma \quad t + \begin{cases} 1, & \text{such that } i == j \\ 10, & \text{otherwise} \end{cases} \\
\text{where } i = \mathit{processor\_id}, \ldots
\end{array}}$$

However, as noted by Bennett [1993], the second-order effects of the processor's cache are almost as equally important in terms of overall performance. While it would be possible

$$(\textsc{init}_\textsc{p})\quad \boxed{\begin{array}{lll} \mathit{init}_\textsc{p}\ P_0 & \Longrightarrow & \mathit{Ping}_{t_{local}=0} \\ \mathit{init}_\textsc{p}\ P_1 & \Longrightarrow & \mathit{WaitForPing}_{t_{local}=0}\ 1000 \end{array}}$$

$$(\textsc{init}_\textsc{s})\quad \boxed{\mathit{init}_\textsc{s}\ S \Longrightarrow \mathit{Nothing}}$$

$$(\textsc{step}_\textsc{p})\quad \boxed{\begin{array}{l} \mathit{step}_\textsc{p}\ (\mathit{Ping}_t, S) \Longrightarrow (\mathit{WaitForPong}_{t+10}\ (t+1000), \mathit{NewPing}\ t) \\ \mathit{step}_\textsc{p}\ (\mathit{Pong}_t, S) \Longrightarrow (\mathit{WaitForPing}_{t+10}\ (t+1000), \mathit{NewPong}\ t) \\ \\ \mathit{step}_\textsc{p}\ (\mathit{WaitForPong}_t\ t_{time\ out}, \mathit{HavePonged}\ t_{recv}) \text{ such that } t \geq t_{recv} \\ \qquad \Longrightarrow (\mathit{Ping}_{t+10}, \mathit{Nothing}) \\ \mathit{step}_\textsc{p}\ (\mathit{WaitForPong}_t\ t_{time\ out}, S) \Longrightarrow (\mathit{Stopped}, S) \\ \mathit{step}_\textsc{p}\ (\mathit{WaitForPing}_t\ t_{time\ out}, \mathit{HavePinged}\ t_{recv}) \text{ such that } t \geq t_{recv} \\ \qquad \Longrightarrow (\mathit{Pong}_{t+10}, \mathit{Nothing}) \\ \mathit{step}_\textsc{p}\ (\mathit{WaitForPing}_t\ t_{time\ out}, S) \Longrightarrow (\mathit{Stopped}, S) \end{array}}$$

$$(\textsc{step}_\textsc{s})\quad \boxed{\begin{array}{llll} \mathit{step}_\textsc{s}\ (\mathit{NewPing}\ t) & \Longrightarrow & \mathit{HavePinged}\ (t+100), & \text{95\% of the time} \\ \mathit{step}_\textsc{s}\ (\mathit{NewPing}\ t) & \Longrightarrow & \mathit{Nothing}, & \text{otherwise} \\ \mathit{step}_\textsc{s}\ (\mathit{NewPong}\ t) & \Longrightarrow & \mathit{HavePonged}\ (t+100), & \text{95\% of the time} \\ \mathit{step}_\textsc{s}\ (\mathit{NewPong}\ t) & \Longrightarrow & \mathit{Nothing}, & \text{otherwise} \\ \mathit{step}_\textsc{s}\ S & \Longrightarrow & S & \end{array}}$$

$$(\textsc{is\_active})\quad \boxed{\begin{array}{lll} \mathit{is\_active}\ (\mathit{WaitForPing}\ t_{time\ out}, S) & \Longrightarrow & \mathit{false} \\ \mathit{is\_active}\ (\mathit{WaitForPong}\ t_{time\ out}, S) & \Longrightarrow & \mathit{false} \\ \mathit{is\_active}\ (P, S) & \Longrightarrow & \mathit{true} \end{array}}$$

$$(\textsc{is\_waiting})\quad \boxed{\mathit{is\_waiting}\ (P, S) \Longrightarrow \neg \mathit{is\_active}\ (P, S)}$$

$$(\textsc{is\_stopped})\quad \boxed{\begin{array}{lll} \mathit{is\_stopped}\ (\mathit{Stopped}, S) & \Longrightarrow & \mathit{true} \\ \mathit{is\_stopped}\ (P, S) & \Longrightarrow & \mathit{false} \end{array}}$$

$$(\textsc{comms\_time})\quad \boxed{\begin{array}{l} \mathit{comms\_time}\ (\mathit{WaitForPing}_{t_{local}}\ t_{time\ out}, \mathit{HavePinged}\ t_{recv}) \\ \qquad \Longrightarrow \mathit{max}(t_{local}, \mathit{min}(t_{time\ out}, t_{recv})) \\ \mathit{comms\_time}\ (\mathit{WaitForPong}_{t_{local}}\ t_{time\ out}, \mathit{HavePonged}\ t_{recv}) \\ \qquad \Longrightarrow \mathit{max}(t_{local}, \mathit{min}(t_{time\ out}, t_{recv})) \\ \mathit{comms\_time}\ (P, S) \Longrightarrow \infty \end{array}}$$

$$(\textsc{final}_\textsc{p})\quad \boxed{\mathit{final}_\textsc{p}\ P \Longrightarrow \mathit{is\_stopped}\ P}$$

$$(\textsc{final}_\textsc{s})\quad \boxed{\mathit{final}_\textsc{s}\ S \Longrightarrow \mathit{true}}$$

Figure 6.9: Adding time outs to the ping-pong system

to extend the heap model to include such factors, section 6.2.2 strongly warns that the primary aim of the operational model is to concisely specify gross behaviour and not to provide accurate estimates of a system's run-time performance.

Access to global memory often has to be carefully controlled through the use of locks, semaphores, and monitors [Hwang and Briggs, 1985, section 8.1, pages 557–577]. For example, GAML does not lock the global work pool, thereby reducing run-time overheads at the risk of duplicating work [Maranget, 1991, section 4.3]. Unfortunately, as each state transition is atomic, this aspect of an implementation is difficult to specify without fragmenting each rule into a series of closely-coupled steps – again, the resulting complexity would be hard to justify. Roscoe [1997, section 0.1, page 4] summarises the problems with shared-memory systems as follows:

> "The main disadvantage from the point of view of modelling general interacting systems is that the communications between components, which are plainly vitally important, happen *too* implicitly."

6.2.5 Message-passing architectures

Traditional message-passing systems provide support for two main operations: sending messages, and receiving messages. High level operations, such as barriers and reduction trees [Snir et al., 1994, chapter 5, pp. 90–126], are often also provided, but these are almost always built on top of the point-to-point primitives. Typically, there are two types of send operation [Hwang and Briggs, 1985, section 5.1.3, p. 332]:

asynchronous an asynchronous send will complete as soon as the message has either been injected into the communication network, or has been stored by the operating system for later transmission;

synchronous a synchronous send will not complete until the target of the message has acknowledged receipt.

However, a process that commits to receiving a message will wait until either the specified message arrives or the operation times out. To help avoid any potentially long and wasteful delays, a poll function is often used to test if a suitable message has already been received (therfore guaranteeing that a receive operation will complete almost immediately).

Asynchronous message passing

The ping-pong system outlined in the previous sections could be re-written as follows:

$$\boxed{\begin{array}{lcl} P_0 & \equiv & \mathit{repeat}\ (\textsc{send}_{\mathit{asynch}}\ P_1\ \mathit{Ping}; \textsc{recv}\ P_0\ \mathit{Pong}) \\ P_1 & \equiv & \mathit{repeat}\ (\textsc{recv}\ P_0\ \mathit{Ping}; \textsc{send}_{\mathit{asynch}}\ P_0\ \mathit{Pong}) \end{array}}$$

The message-based model for the *Ping* transition, shown below, is unsuprisingly similar to that presented in figure 6.9:

$$(\textsc{ping})\ \boxed{\begin{array}{llllr} & \mathit{Send}\ P_1\ \mathit{Ping} & t_{\mathit{local}} & (\mathit{sends}, & \mathit{recvs}) \\ \Longrightarrow & \mathit{Recv}\ P_1\ \mathit{Pong} & t_{\mathit{local}} + 100 & (\mathit{sends} \mathbin{+\!\!+} \langle (P_1, \mathit{Ping}) \rangle, & \mathit{recvs}) \end{array}}$$

Here, the communication network is represented as a pair of queues: *sends* contains the messages generated on this processor that are to be transmitted over the network; and *recvs* contains the messages that have arrived for the processor upto the current time period. This is a partial view of S, which includes such pairs for each of the processors, and a network model containing all in-transit messages.

Message reception can be defined as follows:

$$\text{(PING)}\quad \boxed{\begin{array}{l} \quad\; \mathit{Receive}\ P_0\ \mathit{Ping}\ \ t_{local} \qquad\qquad (\mathit{sends},\ \ \mathit{recvs}) \\[1ex] \text{such that } (P_0, \mathit{Ping}) \in \mathit{recvs} \\[1ex] \Longrightarrow \quad \mathit{Send}\ P_0\ \mathit{Pong}\ \ t_{local} + 100\ \ (\mathit{sends},\ \ \mathit{recvs}') \\ \text{where } \mathit{recvs}' = \mathit{remove}\ (P_0, \mathit{Ping})\ \mathit{recvs} \end{array}}$$

$$\text{(IS\_ACTIVE)}\quad \boxed{\mathit{is\_active}\ (\mathit{Receive}\ P_i\ \mathit{message}) \Longrightarrow \mathit{false}}$$

$$\text{(COMMS\_TIME)}\quad \boxed{\begin{array}{l} \mathit{comms\_time}\ (\mathit{Receive}_{i,t_{local}}\ P_j\ \mathit{message}, S) \\ \qquad \Longrightarrow \mathit{max}(t_{local}, \mathit{next\_recv}\ P_i\ (P_j, \mathit{message})\ S \end{array}}$$

Note that patterns can be used to specify which messages should be received, with the earliest arrival being returned in the case of multiple matches.

While the previous receive model is superficially correct, the time penalty for receiving a message is paid when the processor commits to receiving it, rather than when the message actually arrived. Despite section 6.2.2's argument against accurate performance modelling, this is a serious flaw. Consider, for example, a centralized transaction server which receives a request every 100 time steps. Using the previous model, and assuming it takes 150 time steps to process and reply to a request, the server will complete a transaction every 250 time steps. However, on a real multiprocessor, the server would be too busy receiving the requests to make any progress towards completing even the first transaction. The following model corrects this problem by immediately copying any new messages across from the communication system into a local queue:

$$\text{(INT\_RECV)}\quad \boxed{\begin{array}{l} \quad\; \mathit{code}\ \ t_{local} \qquad\quad \mathit{messages} \quad (\mathit{sends},\ \ \mathit{recvs}) \\[1ex] \text{such that } \mathit{length}(\mathit{recvs}) > 0 \\[1ex] \Longrightarrow\ \mathit{code}\ \ t_{local} + 100\ \ \mathit{messages}'\ \ (\mathit{sends},\ \ \mathit{recvs}') \\ \text{where } \begin{array}{rcl} \mathit{messages}' & = & \mathit{messages} +\!\!+ \langle \mathit{message} \rangle \\ \mathit{message} & = & \mathit{head}\ \mathit{recvs} \\ \mathit{recvs}' & = & \mathit{tail}\ \mathit{recvs} \end{array} \end{array}}$$

Note that any rule which does not match against a specific *code* mode is analogous to a microprocessor interrupt [Hwang and Briggs, 1985, section 2.5.2, pages 125–126]. Implementing such rules can be problematic, and is discussed in section 8.3.3. Messages are now taken from the local queue rather than directly from the communication system:

$$\text{(RECV)}\quad \boxed{\begin{array}{l} \quad\; \mathit{Receive}\ \mathit{pattern}\ \mathit{cont}\ \ t_{local} \qquad \mathit{messages} \quad S \\[1ex] \text{such that } \mathit{pattern} \in \mathit{messages} \\[1ex] \Longrightarrow \qquad \mathit{cont}\ \mathit{message}\ \ t_{local} + 10\ \ \mathit{messages}'\ \ S \\ \text{where } (\mathit{message}, \mathit{messages}') = \mathit{remove}\ \mathit{pattern}\ \mathit{messages} \end{array}}$$

The *Receive* code component takes two arguments: *pattern* determines which messages are acceptable; and *cont* specifies what to do with the matching message once it has been received. As an example, for the *WaitForPing* transition, the pattern would be $(P_0, Ping)$, and the continuation would be $\lambda message.Send\ (P_0, Pong)$.

Synchronous message passing

Synchronous communication can be modelled using pairs of asynchronous sends and blocking receives, as shown below:

$$\begin{array}{|lcl|}\hline \text{SEND}_{synch}\ P_i\ message & \equiv & \text{SEND}_{asynch}\ P_i\ message; \text{RECV}\ P_i\ Ack \\ \text{RECV}_{synch}\ P_j\ message & \equiv & \text{RECV}\ P_j\ message; \text{SEND}_{asynch}\ P_j\ Ack \\ \hline \end{array}$$

These equivalence relations can be used to re-write the ping-pong systems as follows (with the *Pong* message being replaced by *Ack*):

$$\begin{array}{|lcl|}\hline P_0 & \equiv & repeat\ (\text{SEND}_{synch}\ P_1\ Ping) \\ P_1 & \equiv & repeat\ (\text{RECV}\ P_0\ Ping) \\ \hline \end{array}$$

When dealing with large messages, synchronous sends often involve an initial exchange to allow the receiver to allocate a buffer of sufficient size to hold the message:

$$\begin{array}{|lcl|}\hline \text{SEND}_{synch}\ P_i\ message & \equiv & \text{SEND}_{asynch}\ P_i\ Req\ length(message); \\ & & \text{RECV}\ P_i\ Buffer\ a; \\ & & \text{SEND}_{asynch}\ P_i\ Data\ a\ message; \\ & & \text{RECV}\ P_i\ Ack \\ \text{RECV}_{synch}\ P_j\ message & \equiv & \text{RECV}\ P_j\ Req\ message\_length; \\ & & a = \text{ALLOC}\ message\_length; \\ & & \text{SEND}_{asynch}\ P_j\ Buffera; \\ & & \text{RECV}\ P_j\ Data\ a\ message; \\ & & \text{SEND}_{asynch}\ P_j\ Ack \\ \hline \end{array}$$

6.3 Operational semantics and the STG machine

While a denotational semantics defines a language, an operational semantics can be considered as an abstract implementation of the language. Typically, in the context of parallel functional programming, an operational description will need to address the following issues:

the evaluation mechanism specifies the order of evaluation, the argument-passing convention, the return mechanism, and the closure model. The default sequential STG machine is non-strict, has contiguous argument and continuation-based return stacks, and uses the push-enter closure model [Peyton Jones and Salkild, 1989, section 3].

communication and synchronisation ensures that the myriad computational elements co-operate safely and efficiently. The system can be viewed at three levels: single processors, small groups of inter-working processors, and the system as a whole.

resource management includes the definition of the system components (such as the heap and stack), the sharing mechanism, and any high-level tasks, such as the garbage collector, thread scheduler, or load balancer. The sequential STG machine uses a self-updating model for controlling the sharing of thunks [Peyton Jones and Salkild, 1989, section 3.1.2].

partitioning and naming determines the placement of data and functional groups on processors, and the visibility and scoping of variables. Depending upon the nature of the system, both static and dynamic partitioning may have to be considered. The sequential STG machine uses both global and local environments to control scoping and visibility.

These issues are explored in greater depth in sections 6.3.1 to 6.3.4. Moreover, it is also worth considering what is not dealt with at the operational level, but is deferred to the compilation rules (see chapter 8):

register allocation determines which values should be stored in a processors registers at each point in a program's execution [Muchnick, 1997, section 16.1].

closure layout – while the sequential STG machine uses the push-enter closure model, a number of different implementations are possible. For example, GHC uses reversed information tables to reduce the number of indirections required to enter a closure [Peyton Jones et al., 1993].

low-level implementation of components – the operational semantics uses a high-level model of the components, whereas the implementation may use a more complex representation in order to improve efficiency. For example, the three stacks used by the sequential STG machine are actually implemented using just two stacks [Peyton Jones and Salkild, 1989, section 8.2].

low-level optimisations such as branch optimisations, unreachable-code elimination, and instruction scheduling [Muchnick, 1997].

6.3.1 The evaluation mechanism

The evaluation mechanism used by the sequential STG machine has already been introduced in section 4.8. The remainder of this section, therefore, will concentrate on the ways in which the basic model can be modified. For further details, both the STG report [Peyton Jones and Salkild, 1989] and the implementation taxonomy presented by Douence and Fradet [1995] are highly recommended.

The *code* component

Before investigating the evaluation mechanism in any detail, it is worth reflecting upon the role the *code* component plays in the sequential STG machine. As described in section 4.8.2, the *code* component is the primary driving force behind the evaluation process and serves a role similar to that of a microprocessor's instruction stream. However, unlike its hardware equivalent, the *code* component also splits the computation into distinct phases:

phase	description
Eval	co-ordinates the flow of execution within a closure
Enter	applies a closure to the arguments on the argument stack
Return	invokes the appropriate return convention for a given type

While these are sufficient for a sequential system, it will often be necessary to add new phases to the evaluation mechanism. For example, the GUM distributed-memory system presented in section 9.3 adds the *GetWork* and *WaitWork* phases in order to model the load-balancing algorithm.

i.

$$\begin{array}{lcl} \mathcal{E}[\![\texttt{let\#}\ var = exp_{rhs}\ exp]\!]\ \rho & = & case\ (\mathcal{E}[\![exp_{rhs}]\!]\ \rho)\ of \\ & & \quad \bot \rightarrow \bot \\ & & \quad \epsilon \rightarrow \mathcal{E}[\![exp]\!]\ (\rho \overset{\rightarrow}{\oplus} \{var \mapsto \epsilon\}) \end{array}$$

ii.

$$\begin{array}{llllllll} & Eval\ (\texttt{let\#}\ var = exp_{rhs}\ exp)\ \rho & as & & rs & us & h & \sigma \\ \Longrightarrow & Eval\ exp_{rhs}\ \rho & as & return : & rs & us & h & \sigma \\ \text{where} & return \;=\; Forced_{\texttt{Int\#}}\ var\ exp\ \rho' & & & & & & \\ & exp_{rhs}\ \text{is of type}\ \texttt{Int\#} & & & & & & \\ & dom(\rho') \;=\; \mathcal{FV}[\![exp]\!] & & & & & & \end{array}$$

iii.

$$\begin{array}{llllllll} & Return_{\texttt{Int\#}}\ k & as & (Forced_{\texttt{Int\#}}\ var\ exp\ \rho) : & rs & us & h & \sigma \\ \Longrightarrow & Eval\ exp\ \rho' & as & & rs & us & h & \sigma \\ \text{where} & \rho' \;=\; \rho \overset{\rightarrow}{\oplus} \{var \mapsto k\} & & & & & & \end{array}$$

Figure 6.10: Specifying the order of evaluation of the let# construct: rule i. is the denotational semantics of the expression, while rules ii. and iii. are, respectively, the required *Eval* and *Return* phases of the operational semantics

Order of evaluation

While it is straightforward to specify the order of evaluation using denotational semantics (see section 5.4.1), reflecting such changes in the operational model is more complex. There are two main reasons for this:

1. The *Eval* and *Return* phases are closley coupled, requiring that at least two new rules be added to the system. Figure 6.10 illustrates this by presenting both the denotational and operation descriptions required by the addition of the let# construct (see sections 4.3, 4.3.4, and 4.8.4 for further details).

2. As described in section 4.5.1, the STG machine's aggresive take mechanism [Beemster, 1994] means that the evaluation of a partial application cannot be forced. Therefore, at the operational level, all changes to the order of evaluation can only apply to expressions which are known to return primitive values or algebraic data types. This is an unacceptable restriction to place upon a parallel language which supports both polymorphism and higher-order functions. The modifications required to support the forcing of arbitrary STG′ expressions are shown in figure 6.11.

The resulting polymorphic letstrict construct can be used to alter the default order of evaluation, as demonstrated by the definition of the strict_id function given below:

STG′ code

```
strict_id = [] \r [x] -> letstrict y = x in y;
```

The explicit scheduling annotations of Para-functional Haskell discussed in section 9.4 serve as another example of modifying the order of evaluation of the sequential STG machine.

Passing arguments

The sequential STG machine uses a contiguous stack on which to pass arguments, as specified by rules 1 (function application), 2 (closure entry), and, to a lesser extent, 17'

$$\textit{LETSTRICT-EXP}\quad \frac{\begin{array}{l} TE \overset{\text{bind}}{\vdash} \textit{bind} : (\textit{var},\pi) \\ LVE = \{\textit{var} \mapsto \pi\} \\ TE \overset{\rightarrow}{\oplus} LVE \overset{\text{exp}}{\vdash} \textit{exp} : \pi_{exp}\end{array}}{TE \overset{\text{exp}}{\vdash} \texttt{letstrict}\ \textit{bind}\ \textit{exp} : \pi_{exp}}$$

(4A)

$$\begin{array}{lllllll} & Eval\ (\texttt{letstrict}\ (var = exp_{rhs})\ exp_{body})\ \rho & as & rs & us & h & \sigma\\ \Longrightarrow & Eval\ exp_{rhs}\ \rho & as & r : rs & us & h & \sigma \\ \text{where} & r \quad = \quad Forced\ var\ exp_{body}\ \rho' \\ & dom(\rho') \quad = \quad \mathcal{FV}[\![exp_{body}]\!] \end{array}$$

(8′)

$$\begin{array}{llllll} & Return_{\chi\ \pi_1 \ldots \pi_n}\ c\ ws\quad as\quad (Forced\ var\ exp_{body}\ \rho) : rs & us & h & \sigma\\ \Longrightarrow & Eval\ exp_{body}\ \rho' \quad as & rs & us & h' & \sigma \\ \text{where} & \rho' \quad = \quad \rho \overset{\rightarrow}{\oplus} \{var \mapsto a\} \\ & h' \quad = \quad h[a \mapsto (vs\ \texttt{r}\ \{\} \rightarrow c\ vs, ws)] \\ & vs \text{ is a sequence of arbitrary distinct variables} \\ & length(vs) \quad = \quad length(ws) \end{array}$$

(17′)

$$\begin{array}{llllllll} & Enter\ a & as & ()_{stack} & (as_u, rs_u, a_u) : us & h & \sigma \end{array}$$

such that $h[a \mapsto (vs\ \texttt{r}\ xs \rightarrow exp, ws)]$, and $length(as) < length(xs)$

$$\begin{array}{llllll} \Longrightarrow & Return_{Fun}\ a_u\quad as \mathbin{+\!\!+} as_u\quad rs_u & us & h_u & \sigma \\ \text{where} & xs' \quad = \quad take\ length(as)\ xs \\ & f \text{ is an arbitrary variable} \\ & h_u \quad = \quad h[a_u \mapsto ((f : xs')\ \texttt{r} \rightarrow f\ xs', (a : as))] \end{array}$$

(RET_FUN₁)

$$\begin{array}{lllllll} & Return_{Fun}\ a & as & (Forced\ var\ exp_{body}\ \rho) : rs & us & h & \sigma \\ \Longrightarrow & Eval\ exp_{body}\ \rho' & as & rs & us & h & \sigma \\ \text{where} & \rho' \quad = \quad \rho \overset{\rightarrow}{\oplus} \{var \mapsto a\} \end{array}$$

(RET_FUN₂)

$$\begin{array}{lllllll} & Return_{Fun}\ a & as & (Case_{\tau}\ alts\ \rho) : rs & us & h & \sigma \\ \Longrightarrow & Enter\ a & as & (Case_{\tau}\ alts\ \rho) : rs & us & h & \sigma \end{array}$$

Figure 6.11: Modifying the STG machine to allow the forcing of arbitrary boxed expressions

$$
\begin{array}{llllll}
 & \mathit{Eval}\ (f\ xs)\ \rho & fp & rs & us & h \qquad \sigma \\
\multicolumn{6}{l}{\text{such that } \mathit{val}\ \rho\ \sigma f = \mathit{Addr}\ a} \\
\Longrightarrow & \mathit{Enter}\ a & fp' & rs & us & h[fp' \mapsto (fp, xs)] \quad \sigma
\end{array} \tag{1'}
$$

$$
\begin{array}{llllll}
 & \mathit{Enter}\ a & fp & rs & us & h\begin{bmatrix} fp & \mapsto & (fp', as), \\ a & \mapsto & (vs\ \mathbf{r}\ xs \rightarrow exp, ws) \end{bmatrix} \quad \sigma \\
\multicolumn{6}{l}{\text{such that } \mathit{length}(as) = \mathit{length}(xs)} \\
\Longrightarrow & \mathit{Eval}\ exp\ \rho' & fp' & rs & us & h \qquad \sigma \\
\text{where} & \multicolumn{5}{l}{\rho' \ = \ \rho \overset{\rightarrow}{\oplus} \{vs \mapsto ws\} \overset{\rightarrow}{\oplus} \{xs \mapsto as\}}
\end{array} \tag{2'}
$$

$$
\begin{array}{lllllll}
 & \mathit{Enter}\ a & fp & rs & us & h[a \mapsto (vs\ \mathbf{u} \rightarrow exp, ws)] & \sigma \\
\Longrightarrow & \mathit{Eval}\ e\ \rho & fp' & \langle\rangle_{stack} & (rs, a) : us & h[fp' \mapsto (fp, \langle\rangle_{stack})] & \sigma \\
\text{where} & \multicolumn{6}{l}{\rho \ = \ \{vs \mapsto ws\}}
\end{array} \tag{15'}
$$

$$
\begin{array}{llllll}
 & \mathit{Enter}\ a & fp & \langle\rangle_{stack} & (rs_u, a_u) : us & h \quad \sigma \\
\multicolumn{6}{l}{\text{such that } h\begin{bmatrix} a & \mapsto & (vs\ \mathbf{r}\ xs \rightarrow exp, ws), \\ fp & \mapsto & (fp', as) \end{bmatrix} \text{ and } \mathit{length}(as) < \mathit{length}(xs)} \\
\Longrightarrow & \mathit{Return}_{\mathit{Fun}}\ a_u & fp'' & rs_u & us & h_u \quad \sigma \\
\text{where} & \multicolumn{5}{l}{xs' \ = \ \mathit{take}\ \mathit{length}(as)\ xs} \\
 & \multicolumn{5}{l}{f \text{ is an arbitrary variable}} \\
 & \multicolumn{5}{l}{h_u \ = \ h\begin{bmatrix} a_u & \mapsto & ((f : xs')\ \mathbf{r} \rightarrow f\ xs', (a : as)) \\ fp'' & \mapsto & \mathit{combine\_frames}\ fp\ fp' \end{bmatrix}}
\end{array} \tag{17'}
$$

Figure 6.12: Argument passing using heap-allocated application frames

(updating a partial application). The main alternative to this style, is that of using heap-allocated application frames, as used by the New Jersey SML compiler [Appel and Jim, 1990]. For a discussion of the relative advantages and disadvantages of these two systems see Peyton Jones and Salkild [1989, section 3.2.3].

On first inspection of the sequential STG machine, the argument stack appears to be indespensible – however, figure 6.12 shows the main modifications required to use application frames. Essentially, the argument stack has been replaced with a frame pointer, fp, which is the last entry in a singly-linked list of application frames. Instead of pushing a function's arguments onto a stack, the values are stored in a new heap-allocated application frame. The new frame contains a back pointer to the old frame, thereby allowing access to all of the other unused arguments.

As a further example, Mattson's speculative evaluation system [Mattson Jr., 1993a] studied in section 9.2 uses a separate stack for every independent thread of computation (see rules SCHED$_2$ and BH$_2$).

Returning values

The sequential STG machine uses the return stack to control the order of evaluation – once a sub-computation has finished, the top continuation from the return stack is invoked and passed the result as one of its arguments (see section 4.8.7). The New Jersey SML compiler [Appel, 1992] makes this behaviour explicit by transforming all user-defined functions so that they take the return continuation as an extra argument.

The STG machine allows each different data type to have a custom return mechanism. For example, returning literal values is very different from the multi-way switch used when dealing with algebraic constructors. Indeed, the *ReturnLit* rule serves as a template for a number of more specific rules, including *ReturnInt*, *ReturnChar*, and *ReturnDouble*. In this respect, the STG return mechanism subsumes that of traditional imperative languages such as C and Pascal. As an example, section 6.4.2 presents a return mechanism suitable for handling pipeline representations.

Typically, the *Return* mechanism will initiate another *Eval* phase. However, it may be necessary to abnormally return, either due to a runtime error or, for example, the current thread having finished. As a simple example, consider the rule for handling the `error` primitive in the sequential STG′ language:

$$(\textsc{error})\quad \begin{array}{|lllllll|}\hline & \textit{Eval}\ (\texttt{error}\ \textit{message})\ \rho & as & rs & us & h & \sigma \\ \multicolumn{7}{|l|}{\text{such that } (\textit{message}, a) \in \rho} \\ \Longrightarrow & \textit{Return}_{\textit{Error\#}}\ a & as & rs & us & h & \sigma \\ \hline \end{array}$$

$$(\textsc{error\_ret})\quad \begin{array}{|lllllll|}\hline & \textit{Return}_{\textit{Error\#}}\ a & as & rs & us & h & \sigma \\ \Longrightarrow & \textit{Stop} & \langle\rangle_{\textit{stack}} & \langle\rangle_{\textit{stack}} & \langle\rangle_{\textit{stack}} & h & \sigma \\ \hline \end{array}$$

Notice that these two rules could be combined, such that the *code* component switches immediately from the *Eval* phase to the *Stop* phase. While this would certainly be more concise, the presented rules offer the possibility of providing a comprehensive exception-handling mechanism [Pitman, 1990] to the STG machine. Furthermore, MacLennan's REGULARITY rule of language design could be invoked i.e. *Eval* phases should always end with a transition to a *Return* phase. However, a more substantial example of alternative return mechanisms can be found in section 9.2.2, where the issue of thread termination is discussed (see rules $\textsc{sched}_1$ and END_THREAD).

6.3.2 Communication and synchronisation

Unlike the other mechanisms described so far in this section, communication and synchronisation is not an end in itself – it simply enables the myriad processing elements to cooperate safely and effectively to achieve a shared goal.

Black holes

The closure serves as the primary point of synchronisation for single-processor operations. As an example, on uni-processor systems, a thunk is overwritten with a black hole pending completion of its evaluation. This enables self-referencing code, such as $x = \{x\}\ \mathrm{u} \rightarrow 1{+}x$, to be detected and stopped before the heap is exhausted. The rules for this behaviour are shown below:

$$
(15') \quad
\begin{array}{|llllll|}
\hline
 & \mathit{Enter}\ a & \mathit{as} & \mathit{rs} & \mathit{us} & h[a \mapsto (\mathit{vs}\ \mathbf{u} \to e, \mathit{ws})] \quad \sigma \\
\Longrightarrow & \mathit{Eval}\ e\ \rho & \langle\rangle & \langle\rangle & (a, \mathit{as}, \mathit{rs}) : \mathit{us} & h[a \mapsto \mathit{BlackHole}] \quad \sigma \\
\multicolumn{6}{|l|}{\text{where } \rho = \{v_1 \mapsto w_1, \ldots, v_n \mapsto w_n\} \text{ and } (v_i, w_i) = (\mathit{vs}\ !\ i, \mathit{ws}\ !\ i)} \\
\hline
\end{array}
$$

$$
(\textsc{bh}) \quad
\begin{array}{|lllllll|}
\hline
 & \mathit{Enter}\ a & \mathit{as} & \mathit{rs} & \mathit{us} & h[a \mapsto \mathit{BlackHole}] & \sigma \\
\Longrightarrow & \mathit{Eval}\ \texttt{error} & \mathit{as} & \mathit{rs} & \mathit{us} & h & \sigma \\
\hline
\end{array}
$$

Closures are also used as synchronisation points in threaded systems, and, again, black holes are used. However, entering a black hole now indicates that the current thread cannot proceeed until another thread has finished evaluating the original thunk. The current thread, therefore, blocks and another thread is scheduled (a fuller treatment of this style of synchronisation can be found in section 9.2.2):

$$
(\textsc{bh'}) \quad
\begin{array}{|lllllllll|}
\hline
 & \mathit{Enter}\ a & \mathit{as} & \mathit{rs} & \mathit{us} & t_{id} & \mathit{wp} & h[a \mapsto \mathit{BlackHole}\ \mathit{ts}] & \sigma \\
\Longrightarrow & \mathit{GetThread} & \mathit{as} & \mathit{rs} & \mathit{us} & t_{id} & \mathit{wp} & h\begin{bmatrix} a \mapsto \mathit{BlackHole}\ \mathit{ts}', \\ t_{id} \mapsto \mathit{TSO}\ \mathit{state}_1 \end{bmatrix} & \sigma \\
\text{where} & \mathit{ts}' & = & \multicolumn{6}{l|}{\mathit{enqueue}\ t_{id}\ \mathit{ts}} \\
 & \mathit{state}_1 & = & \multicolumn{6}{l|}{(\mathit{Enter}\ a, \mathit{as}, \mathit{rs}, \mathit{us})} \\
\hline
\end{array}
$$

In shared-memory systems, using this style of synchronisation may require that locks are used to control access to shared closures. Without locks, for example, it would be possible for two processors to simultaneously start evaluating the same thunk, thereby duplicating work and possibly reducing efficiency.

Processor-processor interactions

The primary mechanism for inter-processor communication and synchronisation will depend upon the the target architecture, i.e. messages or shared data structures. Due to the implicit nature of shared-memory systems (see section 6.2.4), messages will be used throughout this section. However, the principles remain the same for both paradigms.

The GUM's mechnism for handling references to remote closures will be used to illustrate the basic techniques (see section 9.3.2 for a more comprehensive presentation). Again, the closure is used as the main synchronisation point, with a *FetchMe* closure being used to represent a remote reference. When the closure is entered, a message is sent to the owner requesting its value. The current thread suspends, pending a reply from the owner:

$$
(\textsc{fm}) \quad
\begin{array}{|llllllllll|}
\hline
 & \mathit{Enter}\ a & \mathit{as} & \mathit{rs} & \mathit{us} & t_{id} & \mathit{wp} & h[a \mapsto \mathit{FetchMe}\ j\ a'] & \sigma & b_i \\
\Longrightarrow & \mathit{GetThread} & \mathit{as} & \mathit{rs} & \mathit{us} & t_{id} & \mathit{wp} & h\begin{bmatrix} a \mapsto \mathit{Wait}\ j\ a'\ \langle t_{id}\rangle \\ t_{id} \mapsto \mathit{state} \end{bmatrix} & \sigma & b'_i \\
\text{where} & b'_i & = & \multicolumn{7}{l|}{\mathit{enqueue}_{\mathit{send}}\ (j, \mathit{Fetch}\ a'\ a)\ b} \\
 & \mathit{state} & = & \multicolumn{7}{l|}{(\mathit{Enter}\ a, \mathit{as}, \mathit{rs}, \mathit{us})} \\
\hline
\end{array}
$$

The *Wait* closure is used to prevent multiple *Fetch* messages being sent, and b_i represents the message buffers for processor i. Note the similarity between the entry routines for the *FetchMe* and threaded *BlackHole* closures.

Upon reception of a *Fetch* message, the remote processor will reply with a *Resume* message, which contains the closure's actual value (it may well be a thunk with references to other closures on the remote machine). The rule for handling a *Resume* message is as follows:

$$\text{(RM)}\quad\begin{array}{|l|}\hline
\quad\mathit{code}\;\; \mathit{as}\;\; \mathit{rs}\;\; \mathit{us}\;\; t_{id}\;\; \mathit{wp}\;\; h[a \mapsto \mathit{Wait}\ j\ a'\ \mathit{ts}]\;\; \sigma\;\; (b_{in},\;\; b_{out})_i \\
\\
\text{such that } (j, \mathit{Resume}\ a\ \mathit{packed\_closures}) \in b_{in} \\
\\
\Longrightarrow\ \mathit{code}\;\; \mathit{as}\;\; \mathit{rs}\;\; \mathit{us}\;\; t_{id}\;\; \mathit{wp}'\;\; h'\qquad\qquad\quad \sigma\;\; (b'_{in},\;\; b'_{out})_i \\
\text{where } \begin{array}{lcl} b'_{in} & = & \mathit{dequeue}\ (j, \mathit{Resume}\ a\ \mathit{packed\_closures})\ b_{in} \\ b'_{out} & = & \mathit{enqueue}\ (j, \mathit{Ack}\ a\ a')\ b_{out} \\ h' & = & \mathit{unpack}\ \mathit{packed\_closures}\ h \\ \mathit{wp}' & = & \mathit{add}_{\mathit{active}}\ \mathit{ts}\ \mathit{wp} \end{array} \\
\hline\end{array}$$

The arrival of the *Resume* message updates the remote closure and releases the blocked threads back into the work pool. An acknowledgement is then sent to the original owner, indicating successful reception of the *Resume* message. This is another example of an interrupt-driven rule, as first described in section 6.2.5.

Notice that all inter-processor communications needs to contain sufficient context that the receiver can react in an appropriate manner. For example, the *FetchMe* closure contains both the processor id of the owner and the address at which the closure is stored on the remote processor. This allows the *Fecth* message to be sent to the correct processor, and that, when received, the owner can identify which closure it needs to pack. Similarly, the *Wait* closure needs to retain a copy of this information to allow it to construct the *Ack* response.

As can be seen from the previous examples, the techniques used to model communication and synchronisation are similar to those already used in the STG machine. However, managing the interactions between remote processors is sufficiently complex that the rule design can quickly become challenging. To help manage this complexity, UML sequence diagrams [Fowler and Scott, 1997] prove useful. Figure 6.13 shows the annotated sequence diagram for the series of *Fetchme* interactions. Each processor appears in its own column (often referred to as a swim lane), and the ordering of events is denoted by horizontal positioning. Messages are represented by dashed lines.

As a final example of inter-processor communication and synchronisation, consider the GUM work-request mechanism shown below:

$$\text{(SCHED}_1\text{)}\quad\begin{array}{|l|}\hline
\quad\mathit{GetThread}\;\; \mathit{as}\;\; \mathit{rs}\;\; \mathit{us}\;\; t_{id}\;\; \mathit{wp}\;\; h\;\; \sigma\;\; (b_{in},\qquad\qquad\quad b_{out})_i \\
\\
\text{such that } \mathit{is\_empty}(\mathit{wp}) \\
\\
\Longrightarrow\ \mathit{WaitWork}\;\; \mathit{as}\;\; \mathit{rs}\;\; \mathit{us}\;\; t_{id}\;\; \mathit{wp}\;\; h\;\; \sigma\;\; (b_{in},\;\; \mathit{request} : b_{out})_i \\
\text{where } \mathit{request} = (j, \mathit{Fish}) \text{ and } j = 1 + (i \bmod n) \\
\hline\end{array}$$

This demonstrates how structures other than closures can be used to trigger interactions. The rule is invoked when the local processor has exhausted its work pool, and therefore needs to ask its neighbours for additional tasks. Section 6.3.3 looks at load-balancing strategies in more detail.

Global communications

While the majority of communication and synchronisation will occur at the inter-processor level [Cypher et al., 1993], at times it will be necessary for some form of global communication. Arguably, the two most common forms of global operations are broadcasts and

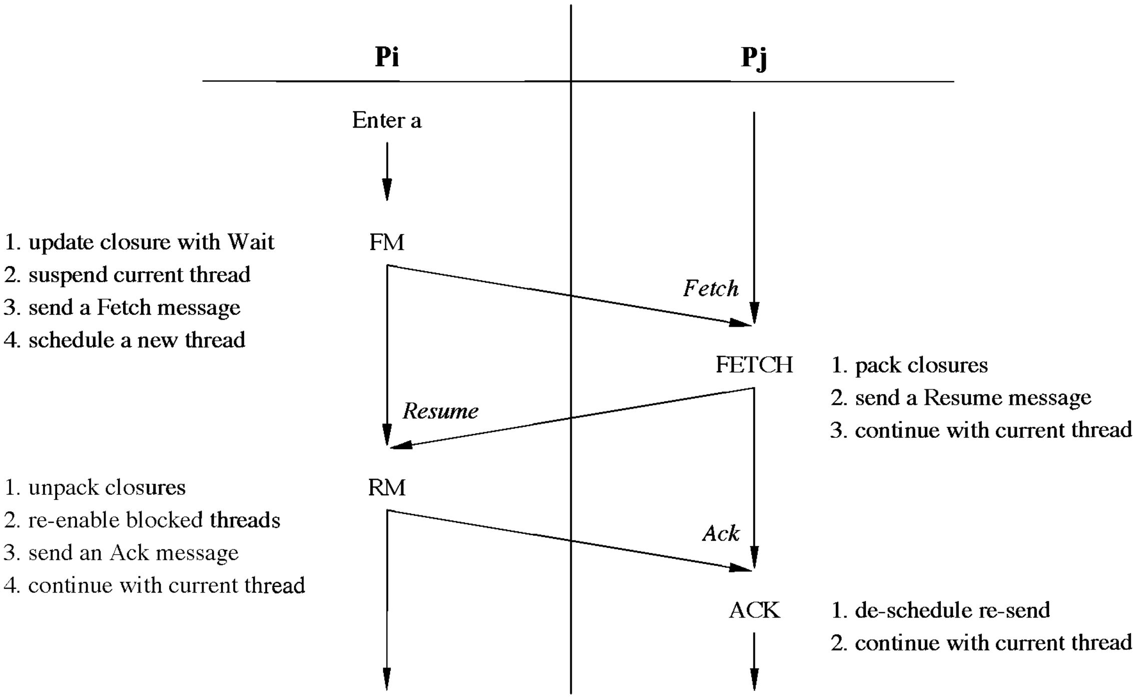


Figure 6.13: A UML sequence diagram for the *FetchMe* entry routine

barriers. To demonstrate the use of these communication primitives, the initialisation and termination phases of a parallel STG machine will be examined.

Surprisingly, the sequential STG machine does not specify a rule for ending the computation. One simple definition would be that the evaluation is complete when all three stacks are empty:

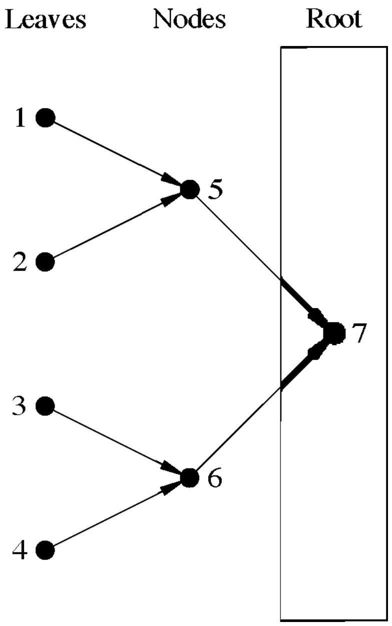
$$(\textsc{stop})\quad \begin{array}{|lllllll|}\hline & \mathit{Return}_\chi\ c\ ws & \langle\rangle_{stack} & \langle\rangle_{stack} & \langle\rangle_{stack} & h & \sigma \\ \Longrightarrow & \mathit{Stop} & \langle\rangle_{stack} & \langle\rangle_{stack} & \langle\rangle_{stack} & h & \sigma \\ \hline \end{array}$$

In a parallel system, however, when the main thread terminates, all processors need to be notified that the evaluation has completed. On a DMMP architecture, a message broadcast would be used:

$$(\textsc{stop}_1)\quad \begin{array}{|llllllllll|}\hline & \mathit{Return}_\chi\ c\ ws & \langle\rangle_{stack} & \langle\rangle_{stack} & \langle \mathit{Finished}\rangle & t_{id} & wp & h & \sigma & b_i \\ \Longrightarrow & \mathit{Stop} & \langle\rangle_{stack} & \langle\rangle_{stack} & \langle\rangle_{stack} & t_{id} & wp & h & \sigma & b_i' \\ \multicolumn{10}{|l|}{\text{where } b_i' = \mathit{broadcast}\ \mathit{Stop}\ b_i} \\ \multicolumn{10}{|l|}{\quad \mathit{broadcast}\ m\ (b_{in}, b_{out}) = (b_{in}, b_{out} +\!\!+ \langle \forall\ j \in \{1,\ldots,n\} \bullet (j,m)\rangle)} \\ \hline \end{array}$$

$$(\textsc{stop}_2)\quad \begin{array}{|llllllllll|}\hline & \mathit{code} & as & rs & us & t_{id} & wp & h & \sigma & (b_{in},\ b_{out})_i \\ \multicolumn{10}{|l|}{\text{such that } (j, \mathit{Stop}) \in b_{in}} \\ \Longrightarrow & \mathit{Stop} & as & rs & us & t_{id} & wp & h & \sigma & (b_{in}',\ b_{out})_i \\ \multicolumn{10}{|l|}{\text{where } b_{in}' \ = \ \mathit{dequeue}\ (j, \mathit{Stop})\ b_{in}} \\ \hline \end{array}$$

During the initialisation phase, each processor loads the code and data required to perform the parallel graph reduction. It is important that the evaluation does not start until all processors are ready. Otherwise, there is the risk of races, whereby an early starter attempts to interact with a laggard, resulting in unpredictable and potentially fatal results. The common solution to this problem is the use of a global barrier. Essentially, a barrier maps each processor onto a logical tree, as shown below.



As soon as a leaf is ready, it sends an *OK* message to its parent, and then awaits the arrival of a *Start* message. A node, however, must wait for its left and right children to become ready (signalled by the reception of their *OK* messages) before it can signal its readiness to its parent. Finally, once the root has received messages from its two children, the *Start* message will then be broadcast, thereby enabling all of the processors to continue. Note also that by inverting the tree, a similar mechanism can be used to implement a more efficient broadcast operation than the version described in the previous section.

Figure 6.14 shows the initialisation rule for a DMMP system using a barrier operation to ensure all processors are ready. The communication roles (*is_root*, *left*, *parent*, etc.) for a 7 processor system are specified as follows (a more robust definition can be found in [Ben-Dyke, 1997]):

i	$is\_leaf(i)$	$is\_node(i)$	$is\_root(i)$	$left(i)$	$right(i)$	$parent(i)$
1	*true*	*false*	*false*			5
2	*true*	*false*	*false*			5
3	*true*	*false*	*false*			6
4	*true*	*false*	*false*			6
5	*false*	*true*	*false*	1	2	
6	*false*	*true*	*false*	3	4	
7	*false*	*false*	*true*	5	6	

Similarly, the low-precedence infix operator $(;)$ denotes the sequence function, which provides a convenient shorthand for creating values of the *code* component:

$$\boxed{(;) \equiv \lambda operation.\lambda continuation.operation\ continuation}$$

$$\boxed{\begin{array}{lcl}
G & = & (P_1,\ldots,P_n)\ \sigma\ b_{init} \\
\text{where} & & \\
P_i & = & (barrier(i); GetThread, \langle\rangle, \langle\rangle, \langle\rangle, h, t_{none}, wp_i) \\
b_{init} & = & (\forall i \in \{1,\cdots,n\} \bullet b_i = (\langle\rangle, \langle\rangle)) \\
wp_i & = & \begin{cases} (\langle\rangle, \langle a_{main}\rangle, & \text{if } is\_root(i) \\ (\langle\rangle, \langle\rangle), & \text{otherwise} \end{cases} \\
h & = & [\forall i \in \{1,\cdots,n\} \bullet a_i \mapsto (vs_i\ \pi_i\ vs_i \rightarrow exp_i, \sigma\ vs_i)] \\
\sigma & = & \{g_1 \mapsto a_1, \cdots, g_n \mapsto a_n\}
\end{array}} \tag{INIT'}$$

$$\boxed{\begin{array}{lcl}
barrier(i) & = & \begin{cases} leaf\_code(i), & \text{if } is\_leaf(i) \\ node\_code(i), & \text{if } is\_node(i) \\ root\_code(i), & \text{if } is\_root(i) \end{cases} \\
leaf\_code(i) & = & send(parent(i), OK); \\
 & & recv(root, Start) \\
node\_code(i) & = & recv(left(i), OK); \\
 & & recv(right(i), OK); \\
 & & send(parent(i), OK); \\
 & & recv(root, Start) \\
root\_code(i) & = & recv(left(i), OK); \\
 & & recv(right(i), OK); \\
 & & broadcast(Start)
\end{array}} \tag{BARRIER}$$

Figure 6.14: Initialisation of a DMMP system using a global barrier operation

Global barriers may also be required during other phases of the evaluation, including, for example, prior to global garbage collection, and termination. Furthermore, the barrier and broadcast templates provide a sound foundation on which to build more complex global operations, such as parallel scans [Lander and Fisher, 1980; Partain, 1991; Springsteel and Stojmenovic, 1989], and gather and scatter [Gropp and Smith, 1993] operations.

6.3.3 Resource management

With regards to the STG machine, resources fall into one of the following three categories:

data is often not considered as a system resource, but its maintenance and distribution is central to the efficient progress of the entire computation. On a uni-processor system, the heap serves as the main data repository, but multi-processor systems offer a spectrum of sharing mechanism. Furthermore, data can also include system information, such as the location of idle processors, or the availability of specialised hardware.

space primarily relates to the processors' memory pool, but can also include on-line storage, such as hard disks. Garbage collection is the main technology used to control a system's usage of space. Typically, most multi-processor implementations will use a two-tiered approach to garbage collection: firstly, each processor will manage its own local memory pool; secondly, when the local collectors fail to reclaim sufficient space, a global collector will be used.

time includes both processor and communication time. Techniques for efficiently managing processor time include scheduling and load balancing. The former is concerned with allocating time on a single processors, while the latter deals with sharing clock cycles over a group of processors. However, neither technology will compensate for inefficient algorithms or poor implementations.

Often it is difficult to completely separate the different concerns of resource management. For example, the following rule demonstrates an optimisation used in the GHC compiler whereby small integer values are pre-allocated in the global heap:

$$(16') \quad \boxed{\begin{array}{llllll} \textit{Return Int } \langle n \rangle & \langle\rangle_{stack} & \langle\rangle_{stack} & (as_u, rs_u, a_u) : us & h & \sigma \\ \multicolumn{6}{l}{\text{such that } 0 \le n \le 100} \\ \Longrightarrow \textit{Return Int } \langle n \rangle & as_u & rs_u & us & h' & \sigma \\ \multicolumn{6}{l}{\text{where } h' = h[a_u \mapsto \textit{Ind } a_{small\_const\_n}]} \end{array}}$$

This can save a large amount of memory as only one closure needs to be allocated for each integer between zero and one hundred. However, this does increase the number of instructions executed during the update phase of integer values. For the GHC compiler, this trade off between the space saved and the increase in time is deemed to be worthwhile. However, in general, such decisions are difficult to make, although the animation can be used to gather supporting empirical data.

The following sections discuss the management of space and time, while the discussion of data management is deferred to section 6.3.4.

Garbage collection

While the original abstract machine does not explicitly provide support for garbage collection [Wilson, 1992], it is essential that the following rules are adhered to[1]

1. *when invoking the garbage collector, the* **root set** *of live closures must be known.* The root set serves as the starting point for the collector, from which all other live closures must be traceable. The root set of the STG machine comprises all of the components, with the exception of the *heap.*

2. *the garbage collector must have access to all of the addresses stored within a closure.* If a closure is known to be live, then all of the closures to which it refers must also be live. To this end, GHC uses specialised garbage-collection entry methods for each type of closure (see section 6.4.3).

3. *during garbage collection it must be possible to differentiate between addresses and literals.* Due to the differences in handling of these two types, a mistake either way could lead to system failure. To avoid tagging, GHC partitions the state's components so that different types rarely appear together, and, when they do, a mask identifies the addresses.

4. *environments must only contain live variables.* This ensures that the garbage collector can remove dead closures as soon as possible. Figure 4.12 illustrates how the free-variable information can be used to safely trim an environment.

[1] Note that most of these restrictions do not apply if a reference-counting system is used. However, due to their inability to reclaim cyclic data structures [Wilson, 1992, section 2.1], it is unlikely that such a system would serve as the main reclamation technology.

$$(\textsc{caf}_1)\quad \boxed{\begin{array}{l} \quad\ \mathit{Enter}\ a \quad\ \ \mathit{as} \quad\ \ \mathit{rs} \quad\ \ \mathit{us} \quad h \quad\ \ \mathit{cafs}\ \ \sigma \\ \\ \text{such that } \mathit{is\_global}\ a \text{ and } h[a \mapsto (\mathrm{u} \mapsto e, \langle\rangle)] \\ \\ \Longrightarrow\ \ \mathit{Eval}\ e\ \{\}_{\mathit{env}}\ \ \langle\rangle_{\mathit{stack}}\ \ \langle\rangle_{\mathit{stack}}\ \ \mathit{us}'\ \ h'\ \ a : \mathit{cafs}\ \ \sigma \\ \text{where} \quad h' \ = \ h[a \mapsto \mathit{FromSpace}\ a', a' \mapsto \mathit{BlackHole}] \\ \qquad\quad \mathit{us}' \ = \ (\mathit{as}, \mathit{rs}, a') : \mathit{us}, \text{and } a' \notin \mathit{dom}(h) \end{array}}$$

$$(\textsc{caf}_2)\quad \boxed{\begin{array}{llllll} & \mathit{Enter}\ a & \mathit{as} & \mathit{rs} & \mathit{us} & h[a \mapsto \mathit{FromSpace}\ a'] \quad \mathit{cafs} \quad \sigma \\ \Longrightarrow & \mathit{Enter}\ a' & \mathit{as} & \mathit{rs} & \mathit{us} & h \qquad\qquad\qquad\qquad\ \ \mathit{cafs} \quad \sigma \end{array}}$$

Figure 6.15: Maintaining a list of CAFs for the garbage collector

A number of different collectors have been used with sequential implementations, including a generational scheme [Sansom and Peyton Jones, 1993] and a hybrid compacting collector [Sansom, 1992]. Despite their differences, a collector can be viewed as a transformation between STG-machine states, and so the basic principles remain the same. Therefore, to illustrate the impact of garbage collection on the STG machine, the following section will focus upon the development of a two-space collector [Wilson, 1992].

Despite dealing with the basics of uni-processor garbage collection, the issue of distributed garbage collection is beyond the scope of this thesis. However, there is no obvious reason why the techniques explored in this chapter could not be used to investigate such algorithms.

A two-space copying collector

As the name suggests, a two-space collector divides the heap into two equal spaces, called *from-space* and *to-space*. During normal operation, closures are allocated in *from-space*, until the available memory is exhausted. The collector is then invoked, copying all live closures from *from-space* into *to-space*. The spaces are then reversed, i.e. *from-space* becomes *to-space* and vice versa, and normal operation resumes.

Before framing the collector in terms of the STG machine, the *root set* needs to be identified. The simplest solution would be to include the entire state, with the exclusion of the heap. However, the global environment would then need to be mutable, as the mapping between top-level variables and their heap addresses could change. This would significantly degrade performance as references to globals would then have to be resolved dynamically. One solution to this problem is to allocate the global closures in a separate block of memory from the heap, and outside the remit of the garbage collector. The various addresses will then remain constant, allowing the compiler to hard-code any references to the variables. However, one problem still remains: constant applicative forms (CAFs, see [Peyton Jones, 1987, section 13.2, page 224]). If evaluated, CAFs will be updated with references into *from space*. According to the first rule presented in the previous section, all such references need to be part of the *root set*. To this end, figure 6.3.3 shows the rules necessary to automatically maintain a list of active CAFs. Essentially, every time a CAF is entered, it allocates a proxy closure inside *from-space* and updates itself with a *FromSpace* indirection. The proxy then becomes the target for the update frame. The heap is now represented by the triple $(\mathit{static}, \mathit{from}, \mathit{to})$, where *static* contains the top-level closures, and from and *to* are *from-space* and *to-space* respectively.

The collector should be invoked whenever heap space becomes scarce, with `let` expressions being the logical triggers:

$$(\textsc{gc}_{init})\quad \boxed{\begin{array}{l} \quad Eval\ e\ \rho \quad as \quad rs \quad us \quad h \quad cafs \quad \sigma \\ \\ \text{such that } is\_let\ e \text{ and } limit\_reached\ h \\ \\ \Longrightarrow\ Eval\ e\ \rho' \quad as' \quad rs' \quad us' \quad h' \quad cafs \quad \sigma \\ \text{where } (\rho', as', rs', us',, h') = two\_space\ (\rho, as, rs, us, cafs, h) \end{array}}$$

The top-level collector simply scavenges the *root set*, and then scavenges the closures which have been copied over into *to-space*:

$$\boxed{\begin{array}{lcll} two\_space\ state & \equiv & Scavenge_{\rho} & state\ \ \$ \\ & & Scavenge_{as} & \quad\ \ \$ \\ & & Scavenge_{rs} & \quad\ \ \$ \\ & & Scavenge_{us} & \quad\ \ \$ \\ & & Scavenge_{cafs} & \quad\ \ \$ \\ & & Scavenge_{h} & \end{array}}$$

The low-precedence infix operator (\$) denotes a function for reversing the normal order of application, and provides a convenient notation for threading state:

$$\boxed{(\$) \quad \equiv \quad \lambda state.\lambda operation.operation\ state}$$

The scavenge routines for the elements of the *root set* simply locates all heap references and copies them into *to-space* by calling the closure's evacuate method (the location of which will be stored in their info table). This method returns the new *to-space* address, and this replaces the original references. As an example, consider the scavenge routine for the local environment, ρ:

$$(\textsc{scav}_{\rho})\quad \boxed{\begin{array}{l} Scavenge_{\rho}\quad (\{v_1 \mapsto w_1, \ldots, v_n \mapsto w_n\},\ \ as,\ \ rs,\ \ us,\ \ cafs,\ \ h_0) \\ \qquad\quad = \quad (\{v_1 \mapsto w'_1, \ldots, v_n \mapsto w'_n\},\ \ as,\ \ rs,\ \ us,\ \ cafs,\ \ h_n) \\ \text{where } (w'_i, h_i) = \begin{cases} (w_i, h_{i-1}), & \text{if } \vdash v_i : \nu \\ Evacuate_a\ (w_i,\ h_{i-1}), & \text{otherwise} \end{cases} \end{array}}$$

Notice that type information is used to differentiate between literal values and addresses (see rule 3 from the previous section) – a real compiler would probably generate static masks from the type information rather than using a dynamic lookup.

The evacuate method is necessarily closure dependent, but some of the more common variants are shown below, starting with a standard *from-space* closure:

$$(\textsc{evac}_{stnd})\quad \boxed{\begin{array}{l} Evacuate_a\quad (a,\quad (static,\quad from[a \mapsto (vs\ \pi\ xs \to e, ws)],\quad to)) \\ \qquad\qquad = \quad (a',\quad (static,\quad from[a \mapsto ToSpace\ a'],\qquad\qquad to')) \\ \text{where } a' \notin dom(to) \text{ and } to' = to[a' \mapsto (vs\ \pi\ xs \to e, ws)] \end{array}}$$

Notice that no attempt is made to scavenge the closure's free variables, as this will take place during $Scavenge_h$. Updating the original closure with a *ToSpace* closure ensures only one copy is every created in *to-space*:

$$(\textsc{evac}_{tospace})\quad \boxed{\begin{array}{l} Evacuate_a\quad (a,\quad (static,\quad from[a \mapsto ToSpace\ a'],\quad to)) \\ \qquad\qquad = \quad (a',\quad (static,\quad from,\qquad\qquad\qquad\qquad to)) \end{array}}$$

Similarly, indirection pointers (see section 6.4.3) do not copy themselves into *to-space*, but simply forward the evacuation:

$$(\text{EVAC}_{ind})\quad \boxed{\begin{array}{llll} \ Evacuate_a & (a, & (static, & from[a \mapsto Ind\ a'], \quad to)) \\ = Evacuate_a & (a', & (static, & from, \quad\quad\quad\quad\quad\ to)) \end{array}}$$

FromSpace closures, created when evaluating CAFs, forward the evacuation to the *from-space* closure, but then update themselves with the new *to-space* address:

$$(\text{EVAC}_{fromspace})\quad \boxed{\begin{array}{l} \begin{array}{lllll} Evacuate_a & (a, & (static[a \mapsto FromSpace\ a'], & from, & to)) \\ \quad\quad = & (a'', & (static', & from', & to')) \end{array} \\ \text{where } (a'', (static', from', to')) = Evacuate_a\ (a', (static, from, to)) \end{array}}$$

Finally, all static non-CAF closures can ignore the evacuation:

$$(\text{EVAC}_{static})\quad \boxed{\begin{array}{lllll} Evacuate_a & (a, & (static[a \mapsto lambda\_form], & from, & to)) \\ \quad\quad = & (a, & (static, & from, & to)) \end{array}}$$

The final stage of the two-space collector requires that the *to-space* closures are scavenged to allow them to update their free variables:

$$(\text{SCAV}_h)\quad \boxed{\begin{array}{l} \begin{array}{lllllllll} Scavenge_h & (\rho, & as, & rs, & us, & cafs, & (static, & from, & to)) \\ \quad\quad = & (\rho, & as, & rs, & us, & & (static', & to', & empty\_heap)) \end{array} \\ \begin{array}{lll} \text{where } (static', from', to') & = & Scavenge_{tospace}\ (a_{start}, static, from, to) \\ \quad\quad a_{start} & = & get\_first\_adress\ to \end{array} \end{array}}$$

The scavenge process starts at the first closure in *to-space* and then iterates through each closure until the end of the heap is reached:

$$(\text{SCAV}_{tospace1})\quad \boxed{\begin{array}{l} Scavenge_{tospace}\ (a, static, from, to) = (a, static, from, to) \\ \text{such that } a \notin dom(to) \end{array}}$$

$$(\text{SCAV}_{tospace2})\quad \boxed{\begin{array}{l} \begin{array}{llll} Scavenge_{tospace} & (a, & static, & from, \quad to) \\ Scavenge_{tospace} = & (a_{next}, & static', & from', \quad to') \end{array} \\ \begin{array}{lll} \text{where } (static', from', to') & = & Scavenge_a\ (a, static, from, to) \\ \quad\quad\quad\quad\quad\quad a_{next} & = & next\_address\ a\ to' \end{array} \end{array}}$$

As with the evacuation methods, the scavenge routines are closure dependent, however, only a few types of closure will ever appear in *to-space*. For the sequential STG machine, only standard closures require a scavenge method:

$$\boxed{\begin{array}{l} \begin{array}{lllll} Scavenge_a & (a, & static_0, & from_0, & to_0[a \mapsto (v_1 \cdots v_n\ \pi\ xs \to e, w_1 \cdots w_n)]) \\ \quad\quad = & & (static_n, & from_n, & to_n[a \mapsto (v_1 \cdots v_n\ \pi\ xs \to e, w'_1 \cdots w'_n)]) \end{array} \\ \text{where } (w'_i, (static_i, from_i, to_i)) \\ = \begin{cases} (w_i, (static_{i-1}, from_{i-1}, to_{i-1})), & \text{if } \vdash v_i : \nu \\ Evacuate_a\ (w_i, (static_{i-1}, from_{,i-1}, to_{i-1})), & \text{otherwise} \end{cases} \end{array}}$$

The pattern is almost exactly the same as for the $Scavenge_\rho$ function, whereby the free variables are evacuated and updated with the new *to-space* addresses.

Scheduling

Typically, there are three levels of scheduling that can be simultaneously active within a parallel functional implementation:

process scheduling tries to ensure that a processor is kept busy by de-scheduling inactive processes and re-scheduling active processes. More sophisticated systems may attempt to ensure fairness (see section 5.4.3) by allowing a process to only run for a fixed *time slice* before re-scheduling.

algorithmic scheduling ensures that the computation proceeds correctly by managing the interactions of the run-time system. Section 6.3.2 deals with the main synchronisation techniques used by this style of scheduling, with the BH' rule demonstrating how the scheduler can be invoked whenever necessary.

user-defined or explicit scheduling attempts to improve the performance of the computation through the use of a priori information. As noted by Burton and Rayward-Smith [1994], without such data it is impossible to develop a fully automatic scheduling strategy that can ensure good performance. Unfortunately, non-strictness interferes with the traditional algorithms for automatically generating explicit schedules [Norman and Thanisch, 1993], and this area has received little attention in the literature (with the exception of algorithmic skeletons). Para-functional Haskell is one of the few functional languages that provides the programmer with explicit scheduling operators, allowing highly complex dependencies to be defined.

As an example of process scheduling, consider the thread-management system presented in section 9.2. As it stands, once a thread is scheduled to run, it will not relinquish control until it either terminates or blocks (on a *BlackHole*). As mentioned above, this is not fair, but the context-switch rule, CS, shown below, provides a solution:

$$
\text{(CS)}\quad
\begin{array}{|l|}
\hline
\begin{array}{llllllll}
\quad\;\; \mathit{code} & \mathit{as} & \mathit{rs} & \mathit{us} & h[a_{tso} \mapsto \mathit{TSO\ state}] & a_{tso} & \mathit{wp} & \sigma \;\; t_{local}
\end{array} \\[1ex]
\text{such that } (t_{local} \bmod (t_{step} * \mathit{steps\_per\_slice})) == 0 \\[1ex]
\begin{array}{llllllll}
\Longrightarrow \mathit{GetThread} & \mathit{as} & \mathit{rs} & \mathit{us} & h[a_{tso} \mapsto \mathit{TSO\ state}'] & a_{tso} & \mathit{wp} & \sigma \;\; t'_{local}
\end{array} \\
\begin{array}{lll}
\text{where } \mathit{state}' & = & (\mathit{code}, \mathit{as}, \mathit{rs}, \mathit{us}) \\
\qquad\;\; t'_{local} & = & t_{local} + t_{step}
\end{array} \\
\hline
\end{array}
$$

This is another example of an interrupt-driven rule, as first described in section 6.2.5. Notice that the guard condition will only capture the intended behaviour if t_{step} is identical for each rule (a down counter would be required to ensure a fixed period if t_{step} varies between rules).

Load balancing

Just as scheduling attempts to maximise the efficiency of a single processor's operation, load balancing aims to maximise the efficiency of a collection of processors. Traditionally, there have been two different approaches to load balancing in parallel functional implementations:

active load balancing is typified by the Alfalfa's diffusion scheduler [Goldberg and Hudak, 1987, section 4.5], which distributes work to neighbouring processors as it is

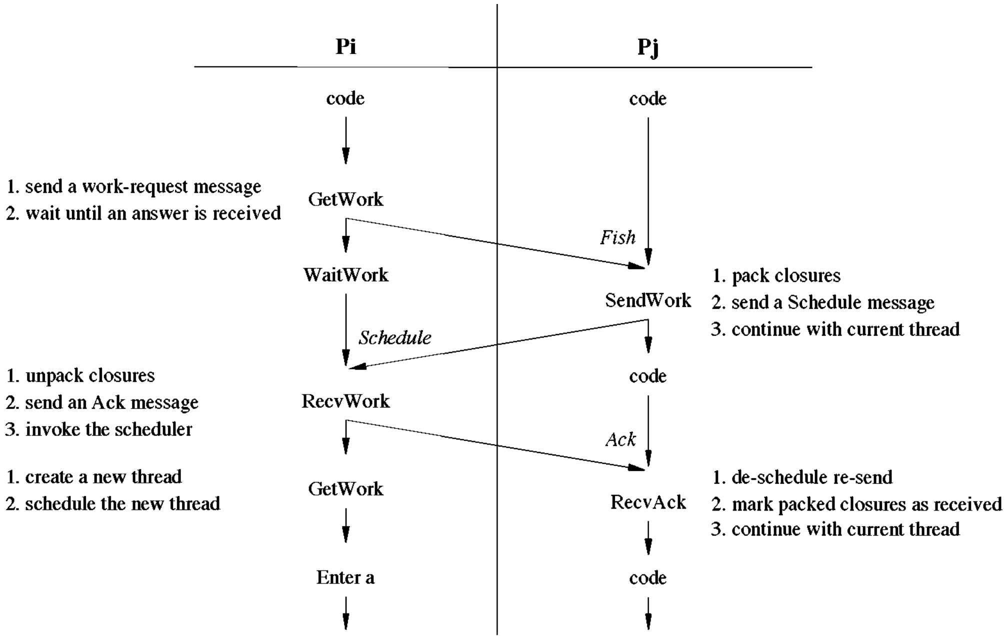

Pi
Pj
code
code
1. send a work-request message
2. wait until an answer is received
GetWork
Fish
WaitWork
1. pack closures
SendWork
2. send a Schedule message
Schedule
3. continue with current thread
1. unpack closures
code
2. send an Ack message
RecvWork
3. invoke the scheduler
Ack
1. create a new thread
1. de-schedule re-send
2. schedule the new thread
GetWork
RecvAck
2. mark packed closures as received
3. continue with current thread
Enter a
code

Figure 6.16: A UML sequence diagram for a successful work-request interaction

generated. As the processors interact, they swap load information, and this is combined with locality maps to determine which processor should receive the new work. The net result is that work "diffuses" through the system. Perhaps surprisingly, no equivalent to the scheduling directives of parafunctional Haskell exists for task placement. Skeletal operators, however, can optimise the load distribution based on knowledge of the precise mechanics of the underlying algorithm.

passive load balancing waits until a processor becomes unemployed before attempting to re-distribute the available work. GUM's fishing mechanism (see section 9.3) involves the out-of-work processor sending a work-request message to one of its neighbours. If the neighbour has sufficient extra work, it returns a suitable portion, otherwise the message is forwarded to another candidate.

While both systems have their merits, a combination of the two is probably necessary for optimal performance.

Whatever approach is finally decided upon, it will undoubtedly build upon the techniques described in section 6.3.2. Once more, UML sequence diagrams can significantly reduce the complexity of designing algorithms involving multiple interactions with remote processors. As an example, consider the two sequence diagrams representing GUM's fishing mechanism shown in figures 6.16 and 6.17. The first figure shows the interactions that need to take place before an unemployed processor receives and starts evaluation of a new task. The second figure shows how fish messages are forwarded if the recipient has no spare work. Furthermore, it also shows how the fish message will be re-spawned after a back-off period once the original message completes a cycle and returns to the unemployed processor.

While the details of the the GUM's passive load-balancing system are contained in section 9.3.2 contains, the following section explores the Alfalfa's diffusion scheduling.

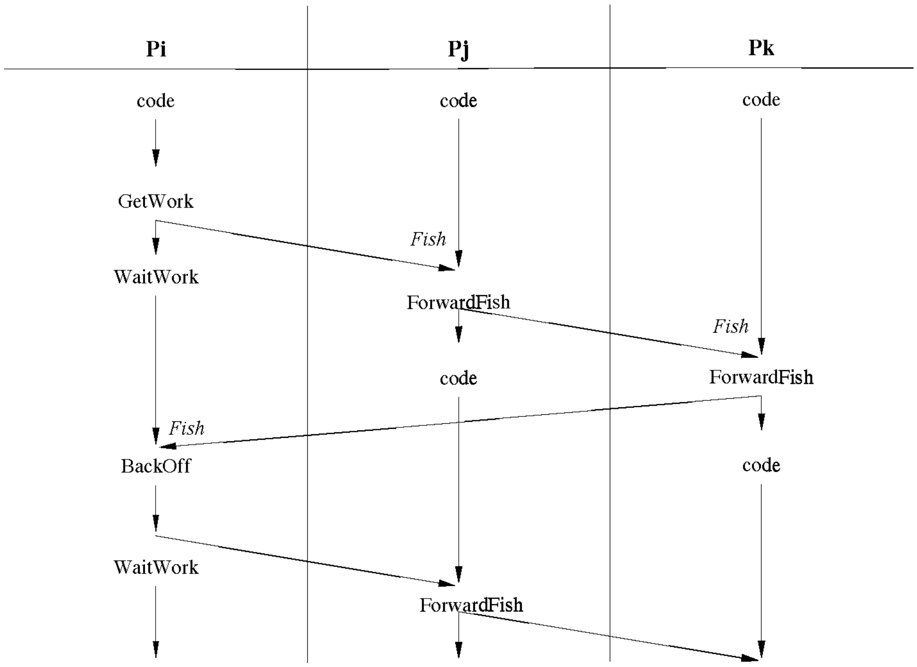


Figure 6.17: A UML sequence diagram for an unsuccessful work-request cycle

Diffusion scheduling

Before moving on to consider the work distribution mechanism, it is worth discussing how each processor is informed of the status of the others. Goldberg and Hudak [1987] use specialised messages to communicate this information, and they develop a number of heuristics to determine the frequency of these transmissions. However, there is no reason why this information could not be piggy-backed onto the regular message traffic. As a simple example, the following rule demonstrates how a GVT-style token-ring algorithm [Ben-Dyke, 1997, section 3.1] could be used to disseminate this information:

$$
\text{(RING)}\quad \boxed{\begin{array}{lllllllllll}
 & \mathit{code} & \mathit{as} & \mathit{rs} & \mathit{us} & t_{id} & \mathit{wp} & \mathit{status} & h & \sigma & (b_{in},\ b_{out})_i \\
\multicolumn{11}{l}{\text{such that } \mathit{probe}\ b_{in}\ (j, \mathit{StatusToken}\ \mathit{new\_status})} \\
\Longrightarrow & \mathit{code} & \mathit{as} & \mathit{rs} & \mathit{us} & t_{id} & \mathit{wp} & \mathit{new\_status} & h & \sigma & (b'_{in},\ b'_{out})_i \\
\text{where} & \multicolumn{10}{l}{b'_{in} = \mathit{dequeue}\ (j, \mathit{StatusToken}\ \mathit{new\_status})\ b_{in}} \\
 & \multicolumn{10}{l}{b'_{out} = \mathit{enqueue}\ (k, \mathit{StatusToken}\ \mathit{new\_status'})\ b_{out}} \\
 & \multicolumn{10}{l}{k = \mathit{neighbour}\ i} \\
 & \multicolumn{10}{l}{\mathit{new\_status'}\ !\ l = \begin{cases} \mathit{size}\ \mathit{wp}, & \text{if } l == i \\ \mathit{new\_status}\ !\ l, & \text{otherwise} \end{cases}}
\end{array}}
$$

Upon reception of the token, the processor updates its own copy of the system's status, modifies the token to reflect its current level of activity, and then passes it on to the next processor in the ring.

The status information enables the local processor to determine the best candidate to receive any new work that is generated. The `letpar` construct is a classic example of a task generator:

$$
\begin{array}{l|l|}
\cline{2-2}
 & \quad \mathit{Eval}\ (\mathtt{letpar}\ v = e_1\ e_2)\ \rho \quad \mathit{as} \quad \mathit{rs} \quad \mathit{us} \quad t_{id} \quad \mathit{wp} \quad \mathit{status} \quad h \quad \sigma \quad b_i \\
 & \text{such that } \mathit{sufficient\_work}\ \mathit{wp} \\
 & \Longrightarrow \quad \mathit{Eval}\ e_2\ (\rho \overrightarrow{\oplus} \{v \mapsto a\}) \quad \mathit{as} \quad \mathit{rs} \quad \mathit{us} \quad t_{id} \quad \mathit{wp} \quad \mathit{status}' \quad h' \quad \sigma \quad b'_i \\
\text{(LETPAR)} & \text{where} \quad (\mathit{task}, h') \ = \ \mathit{pack}\ e_1\ \rho\ h[a \mapsto \mathit{Exported}\ e_1\ \rho] \\
 & \qquad\quad b'_i \qquad\quad = \ \mathit{enqueue}\ (j, \mathit{Schedule}\ \mathit{task})\ b_i \\
 & \qquad\quad j \qquad\quad\ = \ \mathit{select\_target}\ i\ \mathit{status}\ e_1\ \rho \\
 & \qquad\quad \mathit{status}' \quad\ = \ \mathit{inc\_work}\ j\ \mathit{status} \\
\cline{2-2}
\end{array}
$$

The *pack* routine bundles together sufficient context in the hope that the expression e_1 can be evaluated remotely without requiring too much further interaction (see section 6.3.4). The task is then sent to the identified target, but the local processor keeps sufficient data such that the task can be recreated if the message is lost. Upon reception, the remote processor unpacks the task and then sends an acknowledgement to allow the originator to commit the changes to the closures involved. The acknowledgement will also include the remote address of the task's main closure, allowing the *Exported* closure to be replaced with a *FetchMe* closure (see section 6.3.2). Notice also that the status information for the remote processor is increased to avoid it receiving an avalanche of new tasks.

Looking at figure 6.16, it should be clear that there is, in fact, a great deal of similarity between diffusion scheduling and GUM's fishing mechanism. For example, compare the LETPAR rule presented here with GUM's SEND_WRK rule from section 9.3.2. The main difference between the two systems is simply the trigger that initiates the transfer of work.

6.3.4 Partitioning and naming

As mentioned in section 6.3.3, the maintenance and distribution of data is central to the efficient progress of the STG machine. This section covers the following areas of data management:

data partitioning is concerned with striking a balance between the time required to access a particular value, and the amount of time and/or memory dedicated to distributing the data. As with traditional memory management systems [Hwang and Briggs, 1985, section 2.3.1, pages 80–86], the partitioning can either be static (fixed) or dynamic (variable). However, non-strictness again causes problems with abstract analysis, and most modern implementations have to rely on dynamic partitioning.

scoping controls the visibility of variables, and the STG′ language is lexically scoped: identifiers are only accessible within the expression that defines them. The local and global environments, ρ and σ, are used by the STG machine to implement scoping.

locating and accessing remote closures can involve a number of different techniques, depending upon the target architecture. GMSV implementations, for example, have direct access to all closures in the shared heap. However, a number of studies suggest that performance can be improved by moving towards a DMMP design [Hammond and Peyton Jones, 1992; Mattson Jr., 1993b; Islam and Campbell, 1992] where remote values have to be explicitly requested, as with the *FetchMe* closures described previously.

The remainder of this section looks at the first two of these areas.

Static partitioning

The initial partitioning of globally-visible closures is determined by the STG machine's INIT rule. The simplest approach is to allocate all of the *lambda_form* closures to one processor, and use *Ind* or *FetchMe* closures on the others (for GMSV and DMMP architectures respectively). To improve locality at the expense of space efficiency, all functions and constants can be safely allocated on all processors. However, the GUM system goes one step further and even copies top-level thunks [Trinder et al., 1996, section 6.2], risking the duplication of work:

$$\text{(INIT)}\quad \boxed{\begin{array}{lcl}
G & = & (P_1,\ldots,P_n)\ (S_1,\ldots,S_n) \\
\text{where} & & \\
P_i & = & (\mathit{GetThread}, \langle\rangle, \langle\rangle, \langle\rangle, t_{none}, wp_i, h_i, \sigma) \\
S_i & = & (\langle\rangle_{in}, \langle\rangle_{out})_i \\
wp_i & = & (\langle\rangle, \mathit{sparks}_i) \\
\mathit{sparks}_i & = & \begin{cases} \langle a_{main}\rangle, & \text{if } i = 1 \\ \langle\rangle, & \text{otherwise} \end{cases} \\
h_i & = & \begin{cases} h, & \text{if } i = 1 \\ h[a_{main} \mapsto \mathit{FetchMe}\ 1\ a_{main}], & \text{otherwise} \end{cases} \\
h & = & \left[\begin{array}{lll} a_1 & \mapsto & (vs_1\ \pi_1\ vs_1 \rightarrow exp_1, \sigma\ vs_1) \\ \ldots, & & \\ a_n & \mapsto & (vs_n\ \pi_n\ vs_n \rightarrow exp_n, \sigma\ vs_n) \end{array}\right] \\
\sigma & = & \left\{\begin{array}{lll} g_1 & \mapsto & a_1, \\ \ldots, & & \\ g_n & \mapsto & a_n \end{array}\right\}
\end{array}}$$

Ideally, some form of automated mapping strategy [Norman and Thanisch, 1993] should be used. As a first step towards this goal, Dennis [1995, section 5.3, pages 155–156] manually generated *mapping plans* for a Sisal optical-surveillance algorithm. However, in general, traditional static mapping algorithms cannot be readily adapted to work with non-strict systems. Even the large body of work dedicated to strictness analysis has not helped to tame such systems [Bloss and Hudak, 1988; Burn, 1991; Seward, 1992; Beemster, 1994; Peyton Jones and Partain, 1994]. Hence, most modern implementations rely on dynamic partitioning, and only form the crudest of static partitions (as seen previously).

The above discussion ignores the partitioning of the closure-access methods, which typically have to be reproduced on every processor.

Dynamic partitioning

Following on from the previous section, it is unlikely that a static partitioning will prove adequate for the duration of an entire computation. Dynamic partitioning attempts to maintain efficiency by moving data to where it is most needed.

As previously seen, load-balancing systems have a side effect on data placement in that they move clusters of closures between processors as part of their work re-distribution (see, for example, the diffusion scheduler's LETPAR rule from the previous section). Typically, a *pack* function is used to select which closures to include, and collects them together into a structure suitable for transmission. Hammond and Loidl [1996] examined a number of packing schemes, ranging between incremental fetching and bulk fetching. Incremental fetching packs just one closure per message, and invokes the closure's pack method to generate the data:

$$\boxed{\begin{array}{llll}
\multicolumn{4}{l}{\mathit{pack}\ a\ j\ h[a \mapsto (\mathit{vs}\ \pi\ \mathit{xs} \rightarrow \mathit{exp}, \mathit{ws})] = (\mathit{data}, h[a \mapsto \mathit{Fetchme}\ j\ a])} \\
\text{where} & \mathit{data} & = & (a, \mathit{vs}\ \pi\ \mathit{xs} \rightarrow \mathit{exp}, \mathit{mask}, \mathit{ws}) \\
 & \mathit{mask} & = & \mathit{mask}_1 \cdots \mathit{mask}_n \\
 & \mathit{mask}_i & = & \left\{ \begin{array}{ll} 0, & \text{if } \vdash (\mathit{vs}\,!\,i) : \nu \\ 1, & \text{otherwise} \end{array} \right. \\
 & n & = & \mathit{length}\ \mathit{vs}
\end{array}}$$

Note that the *mask* field allows the receiver to differentiate between literal values and addresses (see figure 9.21 for the corresponding *unpack* method). Bulk fetching packs the root closure and as much of its sub-graph as possible using a breadth-first algorithm (see [Trinder, Hammond, Partridge, Peyton Jones and others, 1996] for further details). While these are simple partitioning strategies, they can exhibit good locality of reference, as related values will tend to collect together. However, it is possible for two or more processors to compete for control of a shared thunk, wasting both processor time and communication bandwidth. PAM (the Parallel Abstract Machine [Loogen, Kuchen, Indermark and Damm, 1991]) circumvents this problem by allowing a thread to be migrated once only.

Explicit placement expressions such as parafunctional Haskell's on construct and algorithmic skeletons are the other main drivers of dynamic partitioning. However, as can be seen from the rules in sections 9.4 and 9.5 these simply build upon the techniques presented in this chapter.

Scoping

Free-variable information can be used to determine the exact extent or lifetime of a variable within an expression [Muchnick, 1997, section 3.1, pages 43–44]. Note that this is a different concept to the lifetime of a closure, as references to a closure may be shared and passed outside the confines of a particular expression. However, reference counting does use extent information to garbage collect non-cyclic data [Wilson, 1992, section 2.1]. Consider, for example, the rule for function application, which increases by one the number of references that exist to the function's arguments:

$$(\mathrm{REF}_1)\quad \boxed{\begin{array}{llllllll}
 & \mathit{Eval}\ (f\ x_1 \cdots x_n)\ \rho & & \mathit{as} & \mathit{rs} & \mathit{us} & h_0 & \sigma \\
\multicolumn{3}{l}{\text{such that } \vdash f : \pi} & & & & & \\
\Longrightarrow & \mathit{Enter}\ a & & \mathit{as}' & \mathit{rs} & \mathit{us} & h_n & \sigma \\
\text{where} & a & = & \multicolumn{5}{l}{\mathit{val}\ \rho\ \sigma f} \\
 & \mathit{as}' & = & \multicolumn{5}{l}{\mathit{arg}_1 : \cdots : \mathit{arg}_n : \mathit{as}} \\
 & \mathit{arg}_i & = & \multicolumn{5}{l}{\mathit{val}\ \rho\ \sigma\ x_i} \\
 & h_i & = & \multicolumn{5}{l}{\left\{ \begin{array}{ll} h_{i-1}, & \text{if } \vdash x_i : \nu \\ \mathit{increase\_refs}\ \mathit{arg}_i\ h_{i-1} & \text{otherwise} \end{array} \right.}
\end{array}}$$

$$\boxed{\begin{array}{lcl}
\mathit{increase\_refs}\ a\ h[a \mapsto (\mathit{closure}, \mathit{refs})] & = & h[a \mapsto (\mathit{closure}, \mathit{refs} + 1)] \\
 & & \\
\mathit{decrease\_refs}\ a\ h[a \mapsto (\mathit{closure}, \mathit{refs})] & = & h' \\
\multicolumn{3}{l}{\text{where } h' = \left\{ \begin{array}{ll} h[a \mapsto (\mathit{closure}, \mathit{refs} - 1)], & \text{if } \mathit{refs} > 1 \\ \mathit{add\_free\_cell}\ a\ h[a \mapsto (\mathit{Reclaimed}, 0)], & \text{otherwise} \end{array} \right.}
\end{array}}$$

The references counts are decremented at then end of a boxed variable's extent, as illustrated by the rule handling algebraic returns:

$$
(\textsc{ref}_6)\quad
\boxed{\begin{array}{llllllll}
 & \mathit{Return}_\chi\ \mathit{con}\ \mathit{ws} & \mathit{as} & (\cdots \mathit{con}\ \mathit{vs} \to e \cdots, \rho) : \mathit{rs} & \mathit{us} & h_0 & \sigma \\
\Longrightarrow & \mathit{Eval}\ e\ \rho_{final} & \mathit{as} & \mathit{rs} & \mathit{us} & h_n & \sigma \\
\text{where} & \rho_{final} & = & \rho' \setminus \mathit{dead\_vars} \\
 & \rho' & = & \rho \overset{\rightarrow}{\oplus} \{v_1 \mapsto w_1, \ldots, v_n \mapsto w_n\} \\
 & \mathit{live\_vars} & = & \mathcal{FV}[\![e]\!] \\
 & \mathit{dead\_vars} & = & \mathit{dom}(\rho') \setminus \mathit{live\_vars} \\
 & \mathit{dead\_var}_i & = & \mathit{dead\_vars}\,!\,i \\
 & \multicolumn{6}{l}{h_i = \begin{cases} h_{i-1}, & \text{if } \vdash \mathit{dead\_var}_i : \nu \\ \mathit{decrease\_refs}\ (\rho'\ \mathit{dead\_var}_i)\ h_{i-1} & \text{otherwise} \end{cases}}
\end{array}}
$$

Some expressions will have to both increment and decrement the counts. For example, the `let` expression will increase references during the heap allocation of the closures, and then decreases references to eliminate the dead variables of the body expression. Note, it is important that the reference counts are always incremented before being decremented to avoid incorrect reclamation of a closure. Furthermore, this technique relies on variable renaming to remove all possible ambiguities with respect to shared variable names.

While reference counting is now rarely used as the main garbage collection technology, it can be combined with a copying collector to achieve safe incremental reclamation [Lester, 1989]. Furthermore, GHC uses very similar rules to implement *stack stubbing* to remove potential space leaks. Instead of decrementing reference counts, the stack slots occupied by any dead variables are overwritten or re-used for storing live variables [Peyton Jones, 1992, section 9.4.1, pages 62–63].

Module systems can introduce further complications with regards to scoping, but this is beyond the range of this thesis.

6.4 Modifying the STG machine

In this section a number of guidelines are presented for integrating the changes made to the STG′ language (see section 5.2) into the STG machine. The process involves two interdependent steps: firstly, using the syntax-extension method as an indicator, the rules that need to be added to the state-transition system are identified; secondly, the components needed to support these new rules are developed. To avoid complication, the examples used are all sequential in nature (section 6.2 deals with parallel and architecture-dependent features).

Sections 6.4.1 and 6.4.2 consider the effect of adding a new production rule and a new primitive type (see sections 5.2.1 and 5.2.3 respectively) to the original abstract syntax. Section 6.4.3 then looks at a number of different approaches to implementing the new rules.

6.4.1 New production rules

There are two possible consequences of adding a new production rule to the abstract syntax:

addition of a new state-transition rule when a syntax group is the primary focus of an existing set of transition rules, any extension to the group will be mirrored in the STG machine by the addition of a new rule. Note that the existing rules can serve

syntax group	new rules		existing rules
	mode	templates	
program			initial state
typedecls et al.	see section 6.4.2		initial state, 3, 5, $8'$, 16–17A
bindings	*Eval* `let(rec)`	3	initial state
binding			initial state, 3
simplebind	*Eval* `letstrict` *Eval* `let#`	4A 4B	$8'$ $12'$
lambda_form			initial state, 2, 3, 15–17A
π	*Enter*	2, 15	16–17A
exp	*Eval exp*	3–5, 9, 14	
alts	*Eval* `case` *Return*	4 6, 7, 11, 13	
lalt			4, 11–13
aalt			4, 6, 7
default			4, 6, 7, 11–13
vars	see the *lambda_form* and *aalt* entries		
atoms	*Eval* var_{fun} *Eval cons* *Eval primitive*	1 5 14	
atom			1, 5, 14

Table 6.1: The relationship between the abstract syntax and the STG-machine rules

as example templates – rules 4A, 4B, $8'$, and $12'$ (see figures 4.12 through 4.14) were developed in this manner.

modification of an existing rule if the syntax group is only of minor significance with regards to a rule set, all elements will have to be reviewed. This will result in a list of modifications that must be made in order to incorporate the syntactic extension. If the modifications give rise to complex rules, it is recommended that the whole design be reconsidered (see section 5.2.1 with regards to selecting a suitable alternative).

Either one or both may be applicable, depending upon the syntax group in question – the relationship between the groups and the STG-machine rules is shown in table 6.1. The addition of a new primitive can be treated as if it were an extension of the *exp* syntax group, i.e. a new transition rule must be developed, using rule 14 as a template.

Section 6.4.3 describes the methods that may be employed to support the required additions or modifications. If further extensions are to be made, table 6.1 will have to be updated to take the new and modified rules into account.

6.4.2 New primitive types

Incorporating a new type into the STG machine requires the construction of a specialised $Return_{T_{new}}$ rule. Obviously, if the resulting rule bears little relation to the other of its class (6–$8'$, 11–13, and 16) then the *code* component should be extended. For example, Hill [1994, figure 6.2, page 107] uses the $\overline{Merge_{\texttt{Int\#}}}$ and $\overline{Merge_{\chi'}}$ modes to control the return of literal and algebraic PODs (see section 5.2.3). With regards to the *Eval* rules that will initiate the returns, these will be a product of the integration of the primitive functions and production rules that support the type (see section 6.4.1).

If the type is boxed, or if the corresponding values have to be heap allocated [Peyton Jones et al., 1994, primitive arrays, section 1.4], then a new closure must also be designed (the technical details are explained in the following section).

As an example, consider the *pipeline* type from section 5.2.3. A sequence of addresses could be used to represent a pipeline, with each address pointing to the closure of the function to be performed for that stage. The `emptyPipe` primitive, therefore, simply returns an empty sequence:

$$(\textsc{empty\_pipe})\quad \boxed{\begin{array}{lllllll} & \mathit{Eval}\ \texttt{emptyPipe}\ \rho & \mathit{as} & \mathit{rs} & \mathit{us} & h & \sigma \\ \Longrightarrow & \mathit{Return}_{\underline{\overline{\alpha_1\to\alpha_2}}}\ \langle\rangle & \mathit{as} & \mathit{rs} & \mathit{us} & h & \sigma \end{array}}$$

While these issues would have already been addressed by the denotational semantics, it is now necessary to consider how the pipes will be manipulated. For example, should it be possible to deconstruct a particular pipe through the use of a `case` or equivalent expression? Or is it enough to allow pipes to be specified via chains of `let#` expressions? For the purpose of this example, the latter approach will be used, and the *Return* mechanism can now be specified:

$$(\textsc{ret\_pipe})\quad \boxed{\begin{array}{lllllll} & \mathit{Return}_{\underline{\overline{\alpha_1\to\alpha_2}}}\ \mathit{ps} & \mathit{as} & r : \mathit{rs} & \mathit{us} & h & \sigma \\ \\ \multicolumn{7}{l}{\text{such that } r \equiv \mathit{Forced}_{\underline{\overline{\alpha_1\to\alpha_2}}}\ \mathit{var}\ \mathit{exp}_{\mathit{body}}\ \rho} \\ \\ \Longrightarrow & \mathit{Eval}\ \mathit{exp}_{\mathit{body}}\ \rho' & \mathit{as} & \mathit{rs} & \mathit{us} & h & \sigma \\ \multicolumn{7}{l}{\text{where}\quad \rho' \ = \ \rho \overset{\rightarrow}{\oplus} \{\mathit{var} \mapsto \mathit{ps}\}} \end{array}}$$

Usually the details of the low-level implementation of the sequence should be left to the compilation stage described in chapter 8. However, it is almost certain that the sequence will have to be stored in the heap. This will entail the use of new types of closure, as reflected by the amended rule for `emptyPipe`, and that for `addstagePipe`:

$$(\textsc{empty\_pipe'})\quad \boxed{\begin{array}{lllllll} & \mathit{Eval}\ \texttt{emptyPipe}\ \rho & \mathit{as} & \mathit{rs} & \mathit{us} & h & \sigma \\ \Longrightarrow & \mathit{Return}_{\underline{\overline{\alpha_1\to\alpha_2}}}\ \mathit{ps} & \mathit{as} & \mathit{rs} & \mathit{us} & h' & \sigma \\ \multicolumn{7}{l}{\text{where}\quad h' \ = \ h[\mathit{ps} \mapsto \mathit{EmptyPipe}]} \end{array}}$$

$$(\textsc{extend\_pipe})\quad \boxed{\begin{array}{lllllll} & \mathit{Eval}\ (\texttt{addstagePipe}\ f\ \mathit{ps})\ \rho & \mathit{as} & \mathit{rs} & \mathit{us} & h & \sigma \\ \\ \multicolumn{7}{l}{\text{such that } (f, a) \in \rho} \\ \\ \Longrightarrow & \mathit{Return}_{\underline{\overline{\alpha_1\to\alpha_2}}}\ \mathit{ps}' & \mathit{as} & \mathit{rs} & \mathit{us} & h & \sigma \\ \multicolumn{7}{l}{\text{where}\quad h' \ = \ h[\mathit{ps}' \mapsto \mathit{Pipe}\ a\ \mathit{ps}]} \end{array}}$$

6.4.3 Supporting the new state-transition rules

Having identified the modifications that have to be made to the rule set, the task becomes one of implementing the changes. Hence this section outlines a number of example-driven recipes for providing mechanisms that, in isolation or in combination with others, may prove useful. The recipe book is by no means complete.

$$(\textsc{ind}_1)\quad \boxed{\begin{array}{lllllll} & \mathit{Enter}\ a & \mathit{as} & \mathit{rs} & \mathit{us} & h[a \mapsto \mathit{Ind}\ a'] & \sigma \\ \Longrightarrow & \mathit{Enter}\ a' & \mathit{as} & \mathit{rs} & \mathit{us} & h & \sigma \end{array}}$$

$$(\textsc{ind}_2)\quad \boxed{\begin{array}{lllllll} & \mathit{Return}_{\chi}\ \mathit{con}\ \mathit{ws} & \langle\rangle_{\mathit{stack}} & \langle\rangle_{\mathit{stack}} & (\mathit{as}, \mathit{rs}, a) : \mathit{us} & h[a \mapsto c_{\mathit{old}}] & \sigma \\ \multicolumn{7}{l}{\text{such that } \mathit{size}_{\mathit{closure}}(c_{\mathit{old}}) < \mathit{size}_{\mathit{closure}}(c_{\mathit{new}})} \\ \Rightarrow & \mathit{Return}_{\chi}\ \mathit{con}\ \mathit{ws} & \mathit{as} & \mathit{rs} & \mathit{us} & h' & \sigma \\ \text{where} & h' = h[a \mapsto \mathit{Ind}\ a', a' \mapsto c_{\mathit{new}}] & & & & & \\ & c_{\mathit{new}} = (\mathit{vs}\ \mathtt{r} \rightarrow \mathit{con}\ \mathit{vs}, \mathit{ws}) & & & & & \\ & \multicolumn{6}{l}{\mathit{vs} \text{ is a sequence of arbitrary distinct variables}} \end{array}}$$

Figure 6.18: Indirection pointers and the STG machine

Extending the state

Arguably, the most obvious approach to extending the STG machine is through the addition of a new state component. The high profile afforded the new field is balanced by the potential cost of dedicating machine resources (see chapter 8) to the new part.

The first step is the specification of the component, followed by its integration into the abstract state. Then, all of the existing rules, including the initial and final states, have to be updated. Fortunately, in most cases, this should be trivial. Finally, if the new field contains heap addresses, the component should be added to the garbage collector's root set (specific collectors may have additional obligations).

As an example, the TT rule shown below is a specialised instance of rule 2 (closure entry), which, in addition to the usual entry operations, simply increments the new counter field.

$$(\textsc{tt}_1)\quad \boxed{\begin{array}{llllllll} & \mathit{Enter}\ a & \mathit{as} & \mathit{rs} & \mathit{us} & h[a \mapsto (\mathit{vs}\ \mathtt{r} \rightarrow c\ \mathit{vs}, \mathit{ws})] & \mathit{count} & \sigma \\ \Longrightarrow & \mathit{Eval}\ (c\ \mathit{vs})\ \rho & \mathit{as} & \mathit{rs} & \mathit{us} & h & \mathit{count} + 1 & \sigma \\ \text{where} & \multicolumn{7}{l}{\rho = \{v_1 \mapsto w_1, \ldots, v_n \mapsto w_n\} \text{ and } (v_i, w_i) = (\mathit{vs}\ !\ i, \mathit{ws}\ !\ i)} \end{array}}$$

This is exactly how the AQUA Team [1993, section 9, page 36] implemented GHC's `ticky-ticky` profiling.

New closures

Due to the uniform representation of closures [Peyton Jones, 1992, section 3.1.3], the extension of the *closure* specification will not interfere with other components of the system. The main work lies in the development of new rules to handle all of the applicable entry methods.

For example, the IND$_1$ rule shown in figure 6.18 defines the standard entry method used to access an indirection node, *Ind a* [Peyton Jones, 1987, section 12.4, pages 213–218]. The combination of the new closure and rules provides support for variable-sized closures, a prerequisite of a space-efficient system. The *ToSpace* a_2 closure used by the rules shown in figure 6.20 serves a similar role to an indirection, but is only used during garbage collection [Sansom, 1992, "two-space copying", section 2.1, page 314].

This method is a special case of a more general approach, that of extending an existing component. Relevant examples include the *Forced* continuation used by the `letstrict` and `let#` expressions, and the addition of the new $\overline{Merge}$ mode described in section 6.4.2.

Adding a new computational phase

As with adding a new type of closure, this method is a special case of extending an existing component. A new phase is required whenever a new behaviour cannot be categorised under any of the existing phases. For example, the GUM operational model presented in section 9.3.2 introduces the following phases:

phase	description
GetWork	when a processor runs out of local work, it requests additional work from its neighbouring processors.
WaitWork	having asked for work from a remote processor, the processor simply waits for the arrival of new work

It may also be worth considering the introduction of an artificial phase to highlight a particular behaviour. For example, the update mechanism is spread across the *Return* and *Enter* phases. The following rules show how an *Update* phase can be used to collect together relevant rules:

$$(16') \quad \boxed{\begin{array}{lllllll} & \mathit{Return}\ c\ ws & \langle\rangle_{\mathit{stack}} & \langle\rangle_{\mathit{stack}} & us & h & \sigma \\ \Longrightarrow & \mathit{Update}_{\chi\ \pi_1\ldots\pi_n}\ c\ ws & \langle\rangle_{\mathit{stack}} & \langle\rangle_{\mathit{stack}} & us & h & \sigma \end{array}}$$

$$(\textsc{upd}_1) \quad \boxed{\begin{array}{lllllll} & \mathit{Update}_{\chi\ \pi_1\ldots\pi_n}\ c\ ws & \langle\rangle_{\mathit{stack}} & \langle\rangle_{\mathit{stack}} & (as_u, rs_u, a_u) : us & h & \sigma \\ \Longrightarrow & \mathit{Return}\ c\ ws & as_u & rs_u & us & h_u & \sigma \\ \multicolumn{7}{l}{\text{where } vs \text{ is a sequence of arbitrary distinct variables}} \\ & \multicolumn{6}{l}{\mathit{length}(vs) = \mathit{length}(ws)} \\ & \multicolumn{6}{l}{h_u = h[a_u \mapsto (vs\ \mathtt{n} \rightarrow c\ vs, ws)]} \end{array}}$$

$$(17'\text{a}) \quad \boxed{\begin{array}{lllllll} & \mathit{Enter}\ a & as & rs & us & h & \sigma \\ \multicolumn{7}{l}{\text{such that } h\ a = (vs\ \mathtt{n}\ xs \rightarrow e, ws_f), \text{and } \mathit{length}(as) < \mathit{length}(xs)} \\ \Longrightarrow & \mathit{Update}_{\tau_1 \rightarrow \tau_2}\ a\ vs\ xs\ e\ ws_f & as & rs & us & h & \sigma \end{array}}$$

$$(\textsc{upd}_2) \quad \boxed{\begin{array}{lllllll} & \mathit{Update}_{\tau_1 \rightarrow \tau_2}\ a\ vs\ xs\ e\ ws_f & as & \langle\rangle_{\mathit{stack}} & (as_u, rs_u, a_u) : us & h & \sigma \\ \Longrightarrow & \mathit{Enter}\ a & as' & rs_u & us & h_u & \sigma \\ \multicolumn{7}{l}{\text{where } xs_1 +\!\!+\ xs_2 = xs} \\ & \multicolumn{6}{l}{\mathit{length}(xs_1) = \mathit{length}(as)} \\ & \multicolumn{6}{l}{as' = as +\!\!+\ as_u} \\ & \multicolumn{6}{l}{h_u = h[a_u \mapsto ((vs +\!\!+\ xs_1)\ \mathtt{n}\ xs_2 \rightarrow e, (ws_f +\!\!+\ as))]} \end{array}}$$

As an added advantage, it is now possible for other phases to make use of the update rules without having to duplicate the behaviour. Notice, however, that rules 17'a and $\textsc{upd}_2$ are closely coupled, in that a large amount of context has to be explicitly passed as an argument to the *Update* phase.

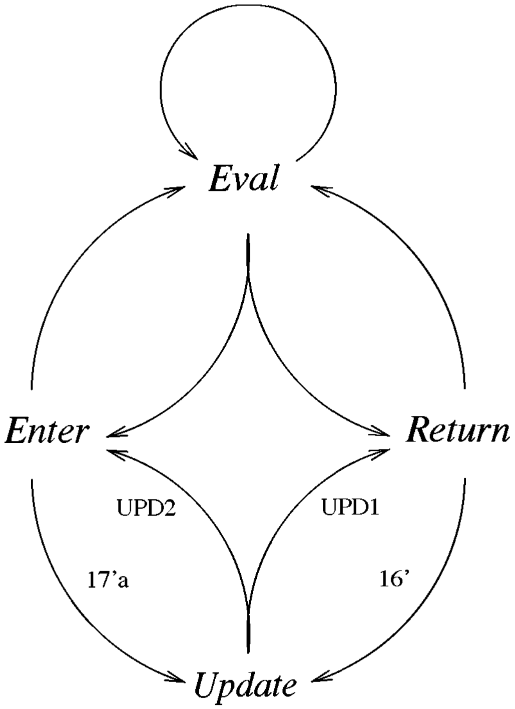


Figure 6.19: The state-transition diagram for the *Update* phase

As the *Update* example demonstrated, the main considerations when adding a new phase are the entry and exit points. The state diagrams introduced in section 4.8.3, clearly show the possible phase interactions and are therefore highly recommended for this stage of the design. Figure 6.19 shows the state-transition diagram for the *Update* example.

Adding a new entry method

When access to a closure's internal representation is required, the only clean solution is to provide a new entry method. However, at one procedure and one word of storage per binding, the associated overhead is high (a number of implementation tricks can reduce both of these costs, see chapter 8). Assuming that the addition cannot be avoided, new rules have to be developed for each type of closure that may be accessed using the new entry method. As an example, figure 6.20 shows the $Entry_{Evac}$ rules for three types of closures: indirections, to-space pointers, and the more usual *lambda_form* variant.

Extending or modifying an existing rule or component

The arguments for and against the modification of a complex system have already been presented in section 5.2.4, and, as before, caution is recommended. To illustrate the power of this approach, the following example does away with the *tagless* aspect of the STG machine.

By evaluating the instruction traces of `case` expressions for a number of modern architectures, Hammond [1992, section 4] notes that a semi-tagging approach to closure representation can achieve a 13% improvement in speed. The rules to effect this change are

$$(\textsc{evac}_1)\quad \begin{array}{|lllllllll|}\hline & \mathit{Enter}_{\mathit{Evac}}\ a & as & rs & us & h_1[a \mapsto \mathit{Ind}\ a'] & h_2 & cafs & \sigma \\ \Longrightarrow & \mathit{Enter}_{\mathit{Evac}}\ a' & as & rs & us & h_1 & h_2 & cafs & \sigma \\ \hline \end{array}$$

$$(\textsc{evac}_2)\quad \begin{array}{|lllllllll|}\hline & \mathit{Enter}_{\mathit{Evac}}\ a_1 & as & rs & us & h_1[a \mapsto \mathit{ToSpace}\ a_2] & h_2 & cafs & \sigma \\ \Longrightarrow & \mathit{Return}_{\mathit{addr}}\ a_2 & as & rs & us & h_1 & h_2 & cafs & \sigma \\ \hline \end{array}$$

$$(\textsc{evac}_3)\quad \begin{array}{|lllllllll|}\hline & \mathit{Enter}_{\mathit{Evac}}\ a_1 & as & rs & us & h_1[a_1 \mapsto c] & h_2 & cafs & \sigma \\ \Longrightarrow & \mathit{Return}_{\mathit{addr}}\ a_2 & as & rs & us & h_1[a_1 \mapsto \mathit{ToSpace}\ a_2] & h_2[a_2 \mapsto c] & cafs & \sigma \\ \hline \end{array}$$

Figure 6.20: Evacuation routines for a two-space compacting collector

$$(\textsc{st}_1)\quad \begin{array}{|llllllll|}\hline & \multicolumn{2}{l}{\mathit{Eval}\ (\texttt{case}\ var\ (\ldots con_i\ xs \to exp_i \ldots))\ \rho} & as & rs & us & h & \sigma \\ & & & & & & & \\ \multicolumn{8}{|l|}{\text{such that } (var, a) \in (\sigma \overset{\rightarrow}{\oplus} \rho),\ \text{ and } h[a \mapsto (vs\ \mathtt{r} \to con_i\ vs, ws)]} \\ & & & & & & & \\ \Longrightarrow & \multicolumn{2}{l}{\mathit{Eval}\ exp_i\ \rho'} & as & rs & us & h & \sigma \\ \text{where} & \rho' & = \rho \overset{\rightarrow}{\oplus} \{x_1 \mapsto w_1, \ldots, x_n \mapsto w_n\} & & & & & \\ & dom(\rho') & = \mathcal{FV}[\![exp_i]\!] & & & & & \\ & (x_j, w_j) & = (xs\,!\,j, ws\,!\,j) & & & & & \\ \hline \end{array}$$

$$(\textsc{st}_2)\quad \begin{array}{|llllllll|}\hline & \multicolumn{2}{l}{\mathit{Eval}\ (\texttt{letstrict}\ var_x = var_y\ exp_{body})\ \rho} & as & rs & us & h & \sigma \\ & & & & & & & \\ \multicolumn{8}{|l|}{\text{such that } (var_y, a) \in (\sigma \overset{\rightarrow}{\oplus} \rho),\ \text{ and } h[a \mapsto (vs\ \mathtt{r}\ ws \to exp, xs)]} \\ & & & & & & & \\ \Longrightarrow & \multicolumn{2}{l}{\mathit{Eval}\ exp_{body}\ \rho'} & as & rs & us & h & \sigma \\ \text{where} & \rho' & = \rho \overset{\rightarrow}{\oplus} \{var_x \mapsto a\} & & & & & \\ & dom(\rho') & = \mathcal{FV}[\![exp_{body}]\!] & & & & & \\ \hline \end{array}$$

Figure 6.21: Semi-tagging `case` and `letstrict` expressions

shown in figure 6.21. With regards to implementation, each closure contains an evaluation-status field which is used to differentiate between thunks (unevaluated expressions), specific constructors, and functions. If the closure is unevaluated, then the original rule (either 4 or 4A) will still apply.

The indirection rule IND$_2$ presented in figure 6.18 could also be considered as a modification of the basic system. However, as it is in keeping with the underlying principles of the STG machine, it is more properly classified as a refinement.

6.5 Animation and testing

While the development of an operational model of a parallel STG machine can be considered an end in itself, the animation of the description provides useful insight into the system dynamics and generally improves confidence in its correctness. The animations

Haskell

```
module SystemSpecification where
import Time
data (Eq p, Show p, Show s)  => SystemSpecification p s = SystemSpecification
 {
  initP :: Int -> p,
  initS :: s,
  stepP :: (p, s) -> (p, s),
  stepS :: s -> s,
  local_time :: p -> Time,
  comms_time :: (p, s) -> Time,
  set_time   :: p -> Time -> p,
  is_active  :: p -> Bool,
  is_waiting :: p -> Bool,
  is_stopped :: p -> Bool,
  finalP :: p -> Bool,
  finalS :: s -> Bool
 }
```

Figure 6.22: The `SystemSpecification` module

described here are primarily built using the techniques described in section 4.8.10, and, as such, the resulting Haskell code is closely related to the operational description.

Section 6.5.1 looks at the animation of the processor framework, while section 6.5.2 provides a concrete example based upon the ping-pong model from section 6.2. Section 6.5.3 then examines how an animation can be used to test and verify a system, before sections 6.5.4 and 6.5.5 look at interactive and batch-mode animations.

6.5.1 The processor framework

The central data structure used during animation is `System Specification`, which is reproduced in figure 6.22. This contains all of the support definitions required to model a system, such as the Haskell implementations of STEP$_P$, IS_ACTIVE, etc. As the specification is polymorphic with respect to `p` and `s`, it can be used for any system, irrespective of the concrete representations used for the processor and communication states.

The `Framework` module provides the main tools for manipulating the system specifications, including the interactive and batch-mode simulations used during the testing phase. These tools typically represent a system's state using the `Ensemble` type:

Haskell

```
type Ensemble p s = ([p], s)
```

All of the tools, however, build upon the `instantiate` function, which derive the three framework operations, INIT, STEP, and FINAL, for a particular system specification. There is a strong correspondence between the implementation and the semi-formal description shown in figure 6.8, as shown by the following fragment:

Haskell

```
  init numProcs    = ([initP n | n <- [1..numProcs]], initS)
  final    (ps, s) = and ([finalP p | p <- ps] ++ [finalS s])
  next_time (p, s) | is_active  p = local_time p
                   | is_waiting p = comms_time (p, s)
                   | is_stopped p = infinity
```

The `initP`, `is_stopped`, etc. functions are extracted from the system specification. The `reduce` function defined below shows how the instantiated operations can be used to obtain all of the states generated during a reduction:

Haskell

```
reduce numProcs specification
  = let (init, step, final) = instantiate specification
        history = iterate step (init numProcs)
    in  takeWhile (not . final) history
```

Note that this style of coding relies on Haskell's non-strict semantics as `history` could well be an infinite list. While it may appear inefficient to generate all reduction states, when combined with a suitable consumer process, the resulting code can be linear in terms of space and time, e.g.:

Haskell

```
putStr $ concat [show e | e <- reduce 2 specification]
```

However, as reported in section 4.8.10, excessive laziness in the system-specification routines can lead to unexpected space leaks, thereby severely damaging performance. To avoid this problem, most of the tools periodically force the evaluation of the entire ensemble.

6.5.2 An example animation: the ping-pong system

Having briefly described the animation of the processor framework, this section provides a concrete example of a `SystemSpecification` for the ping-pong system presented in section 6.2.

The first step is to decide on representations for the processors and communication system. The communication model is a good starting point as it is very simple and can be immediately converted into Haskell code:

Haskell

```
module Communications where
import Time
data Communications = NOTHING
                    | NewPing    Time
                    | HavePinged Time
                    | NewPong    Time
                    | HavePonged Time deriving Show
```

Note that it is suggested that each component be defined in a separate module – this simplifies testing and also improves the chances of re-use between different models.

The processor model is more complex, and needs to record the processor's number, the current time, and a representation of its state. This leads to the following definition:

Haskell

```
module Processor where
import Time
data Processor = Processor {
                              pid   :: Int,
                              time  :: Time,
                              state :: ProcessorState
                            } deriving Show
```

The processor's state is no more complex than that for the communication model:

Haskell

```
data ProcessorState = Ping | WaitForPing |
                      Pong | WaitForPong deriving (Show, Eq)
```

In addition to the type definition, a number of support routines are also required. For example, a `SystemSpecification` requires that the processor model provides an equality operation, `==`. For the ping-pong system, two processors can be considered equivalent if they have the same identifier:

Haskell

```
instance Eq Processor where p1 == p2 = (pid p1) == (pid p2)
```

Other support definitions include get and set methods for the processor's time, and the IS_ACTIVE predicate:

Haskell

```
is_active Processor { state } = state /= WaitForPing && state /= WaitForPong
```

Having developed the communication and processor models, it is now possible to create the `SystemSpecification` shown in figure 6.23. While some of the definitions may look complicated, each operation has been almost directly converted from its operational specification.

6.5.3 Verification and testing

There are three phases associated with the verification and testing of an operational model:

1. *Animation of the model.* The process of converting the operational description into a Haskell program may well reveal problems or faults with the model. The primary aide to the animator is likely to be Haskell's type system. This will ensure that each component is treated in a consistent manner, and that the coupling between different phases is plausible. For example, the type of messages sent and received must match, something that is not necessarily checked in a real implementation. Furthermore, the increased level of detail required by the computer program may well uncover omissions in the system.

2. *Micro-level testing.* Once the generated code compiles correctly, testing can begin in earnest. The main aim of this phase is to check that the animation is faithful to the operational description. This entails testing individual rule transitions, and then moving on to examine sequences of reductions. Fortunately, as the parallel system is built on top of the sequential STG machine, only the new or modified rules need to be considered.

3. *Macro-level testing.* Having established that the various pieces of the animation are correct to a first approximation, the system as a whole must be verified. While the final result of the animation can be validated against the denotational semantics, the gross behaviour of the system is of equal importance. Typical areas of interest may include the total run time, the communication/computation ratio, and patterns of communication. However, it is impossible to anticipate the exact analysis needs for all scenarios.

The last two phases of testing are supported by the animation running in two different modes: interactive, and batch-mode. These are examined in the following sections.

Haskell

```
module SimplePingPong (systemSimplePingPong) where
import SystemSpecification
import Processor
import Communications
import Time

systemSimplePingPong :: SystemSpecification Processor Communications
systemSimplePingPong = SystemSpecification {

  initP = let initP 1 = Processor {pid = 0, time = 0, state = Ping}
              initP 2 = Processor {pid = 1, time = 0, state = WaitForPing}
          in initP,
  initS = NOTHING,

  stepP = let stepP (Processor id     time    Ping,        s)
                  = (Processor id (time + 10) WaitForPong, NewPing time)
              stepP (Processor id     time    Pong,        s)
                  = (Processor id (time + 10) WaitForPing, NewPong time)
              stepP (Processor id     time    WaitForPong, HavePonged t_recv)
                  = (Processor id (time + 10) Ping,        NOTHING)
              stepP (Processor id     time    WaitForPing, HavePinged t_recv)
                  = (Processor id (time + 10) Pong,        NOTHING)
              stepP (p, s) = error "pingpongStepP: no rules matched"
          in stepP,
  stepS = let stepS (NewPing t) = HavePinged (t + 100)
              stepS (NewPong t) = HavePonged (t + 100)
              stepS s = s
          in stepS,

  local_time = processorGetTime,
  comms_time = let ctime (Processor id time WaitForPing, HavePinged t_recv)
                      = max time t_recv
                   ctime (Processor id time WaitForPong, HavePonged t_recv)
                      = max time t_recv
                   ctime _ = infinity
               in ctime,
  set_time   = processorSetTime,

  is_active  = processorIsActive,
  is_waiting = processorIsWaiting,
  is_stopped = processorIsStopped,

  finalP = \p -> False,
  finalS = \s -> False
}
```

Figure 6.23: The `SystemSpecification` for the ping-pong example

command	description
`load` *prog n*	loads the STG′ language program, *prog*, and initialises the system with *n* processors.
`step` *n d*	perform *n* state transitions, displaying a summary every *d* steps. Entering an empty line is equivalent to `step` 1 1.
`unstep` *n*	roll-back *n* state transitions (the system only records the last three states). This command allows a complex or erroneous transition to be re-examined. Furthermore, when used in combination with *set*, it may be possible to repair the system state and continue with the reduction.
`goto` *t*	continue reductions until time *t* is reached. This is primarily used during debugging to jump straight to a known trouble spot.
`show` *c*	display the named component, *c*. Specialised instances of this command can take additional arguments, enabling them, for example, to display specific heap locations.
`set` *c v*	set the value of the named component, *c*, to *v*. This is primarily used to create a scenario for exercising a particular reduction sequence.
`focus` *n*	modifies the behaviour of the `step` command, so that only transitions involving processor P_n are counted. When first started, the interactive animation will have no focus.
`nofocus`	undoes the effect of any previous `focus` commands.

Figure 6.24: The command set supported by the interactive animation framework

6.5.4 Interactive animation

The interactive mode of the animation provides facilities for the animator to examine and adjust the system state, and to apply or undo reduction steps. As an example, consider the use of the interactive environment with the ping-pong example. The animation starts by creating and displaying the INIT state, and then prompts the user to enter a command:

```
  Initial state:
  numProcs=2
  Processorpid=0, time=0, state=Ping
  Processorpid=1, time=0, state=WaitForPing
  NOTHING
interactive>
```

The basic set of commands supported by the interactive animation is shown in figure 6.24. Continuing with the example, the user would force a single reduction step as follows:

```
interactive> step 1 1
  Processorpid=0, time=10, state=WaitForPong
  HavePinged 100
interactive> show P1
  Processorpid=1, time=0, state=WaitForPing
```

Single stepping is useful when closely examining short reduction sequences, or when learning about the system. However, often it is useful to skip ahead a number of reductions, as shown below:

```
interactive> step 3 1
  Processorpid=1, time=110, state=Pong
  NOTHING
  Processorpid=1, time=120, state=WaitForPing
  HavePonged 210
  Processorpid=0, time=220, state=Ping
  NOTHING
```

Notice that only the states that change as a result of a reduction rule are displayed. The following sequence shows how the state can be manipulated to create a new scenario:

```
interactive> unstep 1
  State:
  Processorpid=0, time=10, state=WaitForPong
  Processorpid=1, time=120, state=WaitForPing
  HavePonged 210
interactive> set s (HavePonged 10000)
  HavePonged 10000
```

Now the system cannot proceed until the *Pong* message has been received:

```
interactive> step 1 1
  Processorpid=0, time=10010, state=Ping
  NOTHING
```

6.5.5 Batch-mode animation

Unlike the interactive animation, the batch-mode performs reductions until the final state is reached, incrementally generating a log file. The log file contains an entry for the initial state, the final state (assuming the reduction very terminates), and every intermediate state change. The exact format of the state entries is dependent upon the `Show` instance defined or derived for P and S. For example, the first few entries of the ping-pong system's log file are:

```
Processorpid=0, time=0, state=Ping
Processorpid=1, time=0, state=WaitForPing
NOTHING
Processorpid=0, time=10, state=WaitForPong
HavePinged 100
Processorpid=1, time=110, state=Pong
NOTHING
Processorpid=1, time=120, state=WaitForPing
HavePonged 210
Processorpid=0, time=220, state=Ping
NOTHING
Processorpid=0, time=230, state=WaitForPong
HavePinged 320
```

Typical log files can contain millions of entries, and are therefore of little use in themselves. However, when combined with a general-purpose data-analysis tool, the log files

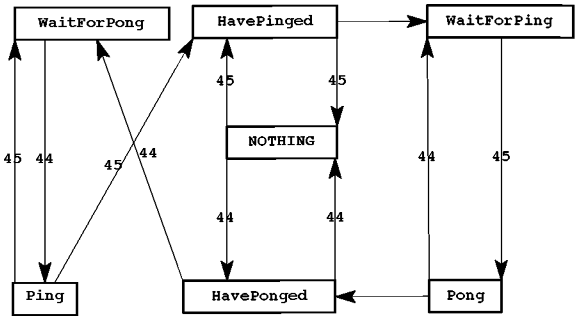


Figure 6.25: The inferred state-transition diagram for the ping-pong system

can potentially be used to extract any behavioural information. To demonstrate this approach, the remainder of this section will describe how the state-transition graph shown in figure 6.25 was derived from the ping-pong system's log file. As well as serving as a useful example, the inferred graph provides an excellent summary of the test coverage of the reduction rules with respect to a particular scenario.

Plumber

Plumber [Haines, Longshaw and Morison, 1997] is a visual programming environment for exploratory data analysis. As can be seen from figure 6.26, the tool comprises two different parts, the canvas and the display table. Computations are constructed by drawing diagrams on the canvas. The diagrams are made up of connected processing elements, where the connecting wires represent the flow of data between them. The display table interactively displays the results either of the whole diagram or of selected processing elements, providing valuable feedback and guidance to the developer. While there are a number of similar tools available both publicly and commercially, Plumber offers a number of advantages:

1. Plumber's open architecture allows it to inter-operate with existing tools, such as databases, spreadsheets, and command-line applications;

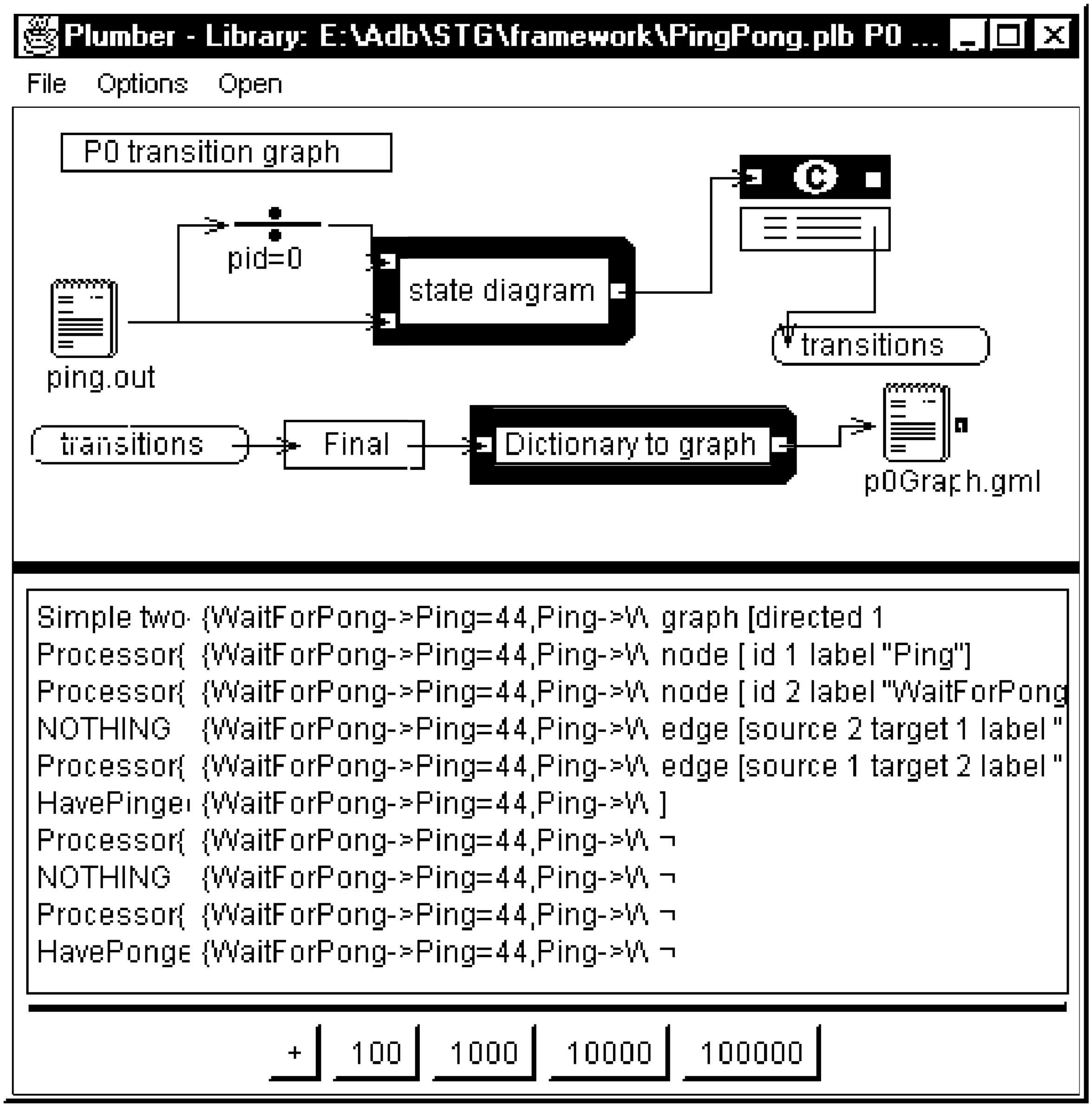
2. Plumber has a rich set of built-in components which can easily be extended and customised to meet the needs of a new application domain;

3. Plumber provides support for structured types, including lists, dictionaries, records, and sets;

4. Plumber is written in Java [Sun Microsystems, 1998], and can therefore run unmodified on all of the popular machine platforms.

GML: a portable graph file format

The Graph Modelling Language, GML [Himsolt, 1996a], is a textual language for describing and annotating graphs. The main body of a GML description contains the node and edge definitions – the example given below defines a simple two-node, one edge graph:



```
Simple two· {WaitForPong->Ping=44,Ping->W\ graph [directed 1
Processor( {WaitForPong->Ping=44,Ping->W\ node [ id 1 label "Ping"]
Processor( {WaitForPong->Ping=44,Ping->W\ node [ id 2 label "WaitForPong
NOTHING    {WaitForPong->Ping=44,Ping->W\ edge [source 2 target 1 label "
Processor( {WaitForPong->Ping=44,Ping->W\ edge [source 1 target 2 label "
HavePinge  {WaitForPong->Ping=44,Ping->W\ ]
Processor( {WaitForPong->Ping=44,Ping->W\ ¬
NOTHING    {WaitForPong->Ping=44,Ping->W\ ¬
Processor( {WaitForPong->Ping=44,Ping->W\ ¬
HavePonge  {WaitForPong->Ping=44,Ping->W\ ¬
```

Figure 6.26: The Plumber diagram for generating state-transition diagrams from logfiles

GML

```
graph [
node [ id 1 label "A" ]
node [ id 2 label "B" ]
edge [ source 1 target 2 label "A->B" ]
]
```

Further tags can be added to both node and edge definitions, including layout information, and node and line styles. A number of graphing tools can import GML files, including the Graphlet editor [Himsolt, 1996b], which provides an automatic layout feature. The graph shown in figure 6.25 was generated by using Graphlet's random layout scheme and then fine tuning the positioning using the manual controls.

Inferring the state-transition diagram

The mechanics of converting the ping-pong log file into a GML description are described below:

1. The log file is split into three streams, each containing the traces for one of the components, P_0, P_1, and S.
2. The current state of the stream's component is then determined. For the processor traces, the state is the label following the `state=` string. The communication state is the first text field. Where appropriate, references to specific times are removed.
3. For each stream, the current state and new state are paired to create a state-transition key. The head of each state stream represents the component's initial state.
4. Each key is then recorded in a counting dictionary, effectively generating a histogram of state transitions.
5. The final dictionary is then converted into a GML description using a Plumber graphing library.
6. The graph definition is written to a file and the Graphlet application invoked on that file. Some manual adjustments may be required to achieve a satisfactory layout of the nodes.

Figure 6.26 shows the Plumber diagram for generating the GML graph for P_0. Note that minor adjustments may be required when processing other types of log files.

6.6 Summary

This chapter has concentrated on the extension of the sequential STG machine into the realm of parallel processing. The first step was to develop a flexible operational system capable of expressing parallel interactions, particularly those common in GMSV and DMMP systems. This then provided the framework within which to carry out a systematic investigation of the impact of parallelism on the STG's evaluation mechanism, communication and synchronisation, resource management, and partitioning and naming. The STG′ language manipulations described in the previous chapter were then considered, and a recipe book developed for integrating them into the STG machine. Finally, a number of techniques for animating and testing the operational models were outlined.

Chapter 7

Simulating the target architecture

7.1 Introduction

This chapter describes the simulator used to test and debug the output of the STG′ compiler (see chapter 8). A RISC-like instruction set, based on the DEC Alpha processor family, serves as the interface between the two systems. The simulator is interpretive and is specified using the state-transition notation presented in chapter 6. While overall performance is relatively poor, the extensible nature of the state-transition model is more important for this particular application.

After an overview of the merits of simulation in section 7.2, the basic uniprocessor model is presented in section 7.3. Using this as a building block, section 7.4 discuses the simulation of multiprocessor architectures. The chapter is then summarised in section 7.5.

7.2 Why simulation?

Traditionally, simulation is used when either analytical modelling or physical measurement is inappropriate. For the purposes of testing and debugging the STG′ compiler, the former can obviously be ruled out, and Bedichek [1995, section 2.1, pages 14–15] attributes the following advantages to simulation over direct measurements: a simulator can easily be augmented with new measurements and debugging features; it can model "ideal" or unavailable components; it is non-intrusive; and simulation runs are often deterministic and therefore repeatable. Taking physical measurements, on the other hand, is typically faster and yields more accurate results. As the correctness of the compiler is the primary concern, these two issues becomes less important, and simulation is the preferred approach.

Multiprocessor simulation is an active area of research, and there are a large number of well-established tools, including PROTEUS [Brewer, Dellarocas and Weihl, 1991], FAST [Boothe, 1994], Shade [Cmelik and Keppel, 1994], and Talisman [Bedichek, 1995]. However, rather than using an existing package, or even a simulation language [Dahl and Nygaard, 1966], it was decided to build a new system using the state-transition approach outlined in chapter 6. The resulting system offers the following advantages: (at the cost of reduced accuracy and performance)

1. the compiler and the simulator use the same internal representation of the target language, thereby simplifying integration and testing. This also allows the representation to be customised (most modern simulators take as their input an executable or a program written either in assembly code or a high-level language, such as C.)

2. modelling a new target architecture is simply a case of modifying a state-transition system, a topic already covered in section 6.4.1. PROTEUS, for example, allows customisation of its four main modules [Brewer et al., 1991, section 3, pages 4–8] but the interface is static and the code serves as both implementation and specification.

3. by adopting the same general approach to animation as used in chapters 6 and 8, the consistency and coherence of the framework is maintained.

7.3 Modelling a RISC uniprocessor

7.3.1 RISC architectures and the DEC Alpha AXP

The recent trend in computer architecture has been towards RISC (Reduced[1] Instruction Set Computer) systems. The salient features of this class of processor include [Kane and Heinrich, 1992, chapter 1, pages 1–22]: one instruction completed per cycle; simple addressing modes and instruction formats; sufficient on-chip memory (registers and cache) to overcome the processor/memory bottleneck; and a reliance on optimising compilers to obtain the best possible performance.

A large number of commercial RISC systems have been developed, including the MIPS [Kane and Heinrich, 1992] and PowerPC [May, Silha, Simpson and Warren, 1994] architectures. Furthermore, modern parallel computers typically use these uniprocessors as basic computational building blocks. For example, Cray's T3D [Koeninger, Furtney and Walker, 1992] uses up to 1,024 DEC Alpha AXP microprocessors, while the CM-5 [Hillis and Tucker, 1993] uses a similar number of SPARC processors [Sun Microsystems, 1988].

For the purposes of this study, the Alpha AXP architecture [Sites, 1992] was selected as the basic model for the uniprocessor simulation. The Alpha is well suited to this role because:

- it is, arguably, the fastest commercial processor currently available.

- by avoiding all non-replicated hidden state, including condition codes [RISC Machines Ltd (ARM), 1994, section 4.2, page 20], suppressed-instruction bits [Hewlett Packard, 1994, section 4, page 4-7], and precise arithmetic exceptions [Kane and Heinrich, 1992, section 9, page 9-2], future designs can take advantage of multiple instruction issue (this also simplifies the design of the state transition model.)

- all operating-system support is handled by privileged software subroutines, called PALcode (see sections 7.3.2 and 7.4).

- shared-memory multiprocessing support is an integral part of the architecture. The $load_{linked}$ and $store_{linked}$ instructions (see section F.3) provide a safe mechanism for updating shared addresses.

7.3.2 The state-transition system

The resulting model is straightforward, if not concise, and uses the abstract state given below:

$$(\mathit{code},\ \mathit{program\ counter},\ \mathit{registers},\ \mathit{memory},\ \mathit{semaphore},\ \mathit{exceptions})$$

[1] Clements [1991] argues that the 'R' of RISC should stand for "Regular" to better reflect the underlying philosophy

2	$Decode$	pc	$registers$	$memory$	$semaphore$	$exceptions$
$\Longrightarrow$	$Execute\ instruction$	pc'	$registers$	$memory$	$semaphore$	$exceptions$
where	$instruction = decode\ memory(pc)$					
	$pc' = pc +_{32} 4$					

6	$Execute\ load\ offset(base)\ target$	pc	$registers$	$memory$	$semaphore$	$exceptions$
$\Longrightarrow$	$PostExec$	pc	$registers'$	$memory$	$semaphore$	$exceptions$
where	$registers' = registers[target \mapsto value]$					
	$value = memory(address)$					
	$address = offset +_{32} registers(base)$					

3	$PostExec$	pc	$registers$	$memory$	$semaphore$	$(pending\ , mask, counter\ , trigger)$
$\Longrightarrow$	$Decode$	pc	$registers$	$memory$	$semaphore$	$(pending', mask, counter', trigger)$
where	$counter' = counter +_{32} 1$					
	$pending' = pending \cup clock\_interrupt$					
	$clock\_interrupt = if\ (trigger == counter)\ then\ \{Clock\}\ else\ \emptyset$					

Figure 7.1: A selection of RISC state-transition rules

The state components are defined in table 7.1, and a number of example instructions and transition rules are shown in table 7.2 and figure 7.1 respectively (appendices F and G contain the full details of the 49 instructions and 30 transition rules). Note that the *exceptions* field contains a counter, which is automatically incremented by the *Decode* mode (rule 3), and this serves as the main performance metric (see section 8.2).

The instruction pipeline

The *code* component loosely models a processor's instruction pipeline [Hwang and Briggs, 1985, chapter 3, page 153], and the transitions will typically proceed as follows: $Decode \Longrightarrow Execute \Longrightarrow PostExec \Longrightarrow Decode \Longrightarrow \cdots$ (rules 2, 5–30, and 3) – see figure 7.1. If an unmasked exception is raised then the sequence will become $Decode \Longrightarrow Exception$ (rule 1). The appropriate PALcode will be invoked to handle the interrupt, and this is responsible for clearing the exception and returning to the *Decode* mode of operation (rule 4).

Accessing operating-system services

The *syscall* instruction provides the interface between a program and the operating system [DEC, 1992, chapter 9]. Rather than modelling these calls down to the instruction level, a separate transition rule specifies the entire operation, as illustrated by the following example:

SET_TRIGGER	$Exception$	pc	$registers$	$memory$	$semaphore$	$(pending\ , mask, counter\ , trigger)$
such that $SysCall \in (pending \setminus mask)$ and $memory(pc -_{32} 1) == syscall$ SET_TRIGGER						
$\Longrightarrow$	$Decode$	pc	$registers$	$memory$	$semaphore$	$(pending', mask, counter', trigger')$
where	$counter' = counter +_{32} t_{os\_call} +_{32} t_{set\_trigger}$					
	$trigger' = registers(1)$					
	$pending' = pending \setminus \{SysCall\}$					

To test this approach, a Unix-style process model [Goodheart and Cox, 1994, chapter 4] has been developed. By adjusting the instruction-count overhead associated with context

	specification	description
code	*Decode*	fetch the instruction referenced by the program counter (which is then incremented), decode it, and then pass it on to be executed
	Execute instruction	evaluate the current instruction
	PostExec	increment the timer counter and generate a timer exception if necessary
	Exception	invoke the exception handler
program counter	*address*	the address of the next instruction to be executed
registers	$registers(i_{reg}) = value$	records the contents of the register file, where $0 \leq i \leq 32$, and $\boldsymbol{registers}(0) \equiv 0$
memory	$memory(address) = value$	a model of the processor's main memory
semaphore	$(address, boolean_{stale?})$	the *address* field records the memory location read by the last linked load, and *stale*? indicates if this location has been updated since the load (see the $load_{link}$ and $store_{link}$ instructions)
exceptions	(set of *exceptions*, set of *exceptions*, $i_{counter}, i_{trigger}$)	the first two fields record which exceptions have been raised and those that should be ignored for the present. The counter is incremented after every instruction, and when it matches the trigger a *Clock* exception is raised
instruction	see appendix F	any one of 49 possible instructions, including memory references, branches, operate instructions, and system instructions
value	*address* | *literal*	either a memory address or a machine literal (only 32-bit integers are supported)
exception	*Clock*	raised when the cycle counter equals the trigger value
	Overflow	raised when an add_{trap} or sub_{trap} instruction causes under- or overflow, but only acted upon when the appropraie $barrier_{trap}$ instruction is executed
	SysCall	raised by a *syscall* instruction
	Unaligned	raised whenever a memory reference is not word aligned (i.e. the two least-significant bits are non-zero)

Table 7.1: State components of the RISC uniprocessor

mnemonic	instruction		description
LD	*load*	$offset_{16}(base_{reg})$ $target_{reg}$	load a word from the address (word-aligned) formed by adding the 16-bit signed offset and the contents of the base register. The value is then stored in the target register
JSR	$jump_{link}$	$offset_{16\leftarrow 2}(base_{reg})$ $link_{reg}$	the 16-bit signed offset is first shifted two places to the left, then added to the base register to form the target address (which must be word aligned). Before the PC is set to this new address, the link register is loaded with the PC's current value, allowing a subroutine to return control back to the caller
ADD	*add*	$value_{reg}$ reg_imm_{8} $target_{reg}$	signed addition of the first two arguments, the result of which is stored in the target register
CALL_PAL	*syscall*	$immediate_{26}$	cause a system-call exception

Table 7.2: A selection of RISC instructions

switches and task creation, the system can also (crudely) simulate user-level threads [Birrell, 1989] and hardware contexts [Weber and Gupta, 1989; Agarwal et al., 1993]. Note that no support for any form of I/O [Goodheart and Cox, 1994, chapter 5] is provided, although adding the necessary interfaces should be straightforward.

7.4 Modelling multiprocessor systems

7.4.1 Basic building blocks

The processor model presented in figure 6.1 is still valid, and provides the underlying structure for the multiprocessor simulator – the RISC uniprocessor model fills the role of P, while the communication system, S, is styled after the intended target architecture.

There are two ways of providing a program with access to the communication system: firstly, the *syscall* interface can be used for large-grained operations, such as message passing (see the SET_TRIGGER example from section 7.3.2); secondly, fine-grained activities, including accessing shared-memory, should be directly incorporated into the state-transition rules – for example, the following rule forms part of a crude model of a local cache:

6'	$Execute\ load\ offset(base)\ target$	pc	$regs$	$cache$	$memory$	smp	exs
such that $(address, value) \in cache$							
$\Longrightarrow$	$PostExec$	pc	$regs'$	$cache$	$memory$	smp	exs
where	$regs' = regs[target \mapsto value]$						
	$address = offset +_{32} regs(base)$						

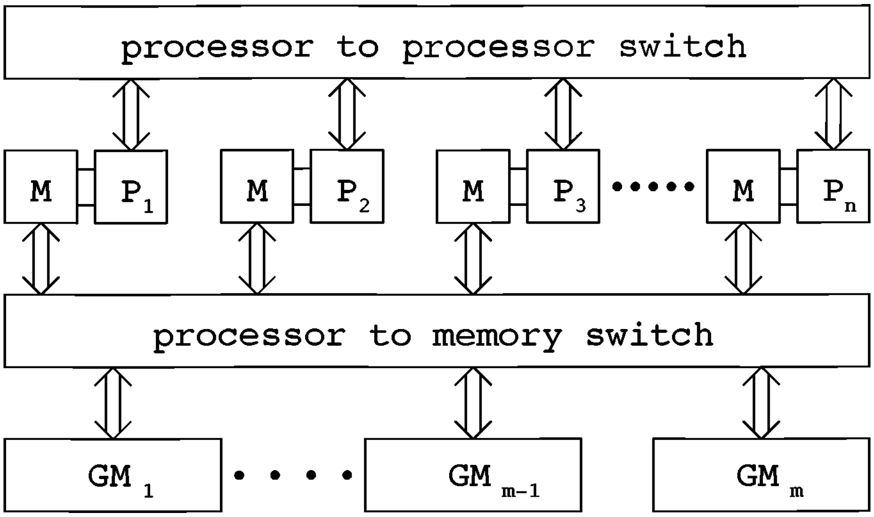


Figure 7.2: The hybrid architecture model, consisting of n interconnected RISC processors (P), each with local memory (M), connected to a pool of m global-memory modules (GM)

7.4.2 Cost models

As each RISC transition rule is equivalent to one instruction step on a real processor, many of the objections raised in section 6.2.2 do not apply, and the instruction count can be used to estimate a program's run time. As for estimating the instruction count of any communication primitives, the LogP model proposed by Culler et al. [1993] is recommended, whereby algorithms are modelled using the parameters L, o, g and P, which are defined as follows:

L - An upper bound on the *latency* involved with communicating a word length message from source to destination.

o - the *overhead* attributed to the transmission or reception of each message. During this time a processor can engage in other activity.

g - the minimum *gap* allowed between consecutive message transmission or reception. The reciprocal gives the per-processor bandwidth.

P - the number of *processors*. (Each local operation is assumed to take unit time.)

Results on a variety of different architectures (including dataflow, shared memory and message passing systems) have shown that the model closely reflects the actual performance of algorithms developed this way. If a more detailed timing model is required, then Talisman's iterative technique [Bedichek, 1995, section 6.1, page 20] should be applied.

7.4.3 A hybrid architecture

To show the viability of the RISC-based framework, the hybrid architecture shown in figure 7.2 has been developed and tested. Shared memory is accessed via the usual *load* and *store* operations, with $load_{linked}$ providing a basic semaphore facility (see sections F.3 and G.5 for a full description of these instructions). The message-passing network is accessed via the system calls shown in table 7.3, which are modelled after the PROTEUS [Brewer, Dellarocas and Weihl, 1991, section 4.4, pages 35–36] application program interface (the `send` primitive is non-blocking).

system call	inputs	outputs	description
`send`	$R19$ return address $R20$ destination $R21$ message buffer $R22$ length of data	$R18$ corrupted $R19$ -1 on failure $R20$ destination $R21$ message buffer $R22$ length of data	send a message
`recv`	$R20$ return address $R21$ buffer length $R22$ message buffer	$R19$ corrupted $R20$ destination $R21$ length of data (-1 on failure) $R22$ message buffer	receive a message
`poll`	$R21$ return address	$R21$ corrupted $R22$ length of data (-1 if no message)	test for arrival

Table 7.3: The hybrid architecture's message-passing interface

The traces generated by the message-passing components of the simulator follow the PICL standard [Geist, Heath, Peyton and Worley, 1990; Worley, 1992], and, with some manual editing, are suitable for use with the ParaGraph visualisation tool [Glendinning, Hockney, Pritchard and others, 1993]. As an example, figure 7.3 shows spacetime diagrams [Heath and Finger, 1993, section 5.2.2, pages 22–23] for three distributed-memory GVT algorithms [Ben-Dyke, 1997].

7.5 Summary

This chapter has presented a state-transition model of a multiprocessor architecture using the Alpha RISC processor as the basic computational engine. This is used to test and debug the output of the STG′ compiler (see chapter 8).

How does this model compare with existing simulation tools? Unfortunately, using performance and accuracy as the main criteria, the system is a failure. The latter could be corrected as "the level of detail is limited only by the time available for simulation development" [Jain, 1991, section 24.1, page 394]. However, as a tool for rapidly testing the output of the STG′ compiler, the flexibility and convenience compensates for these limitations.

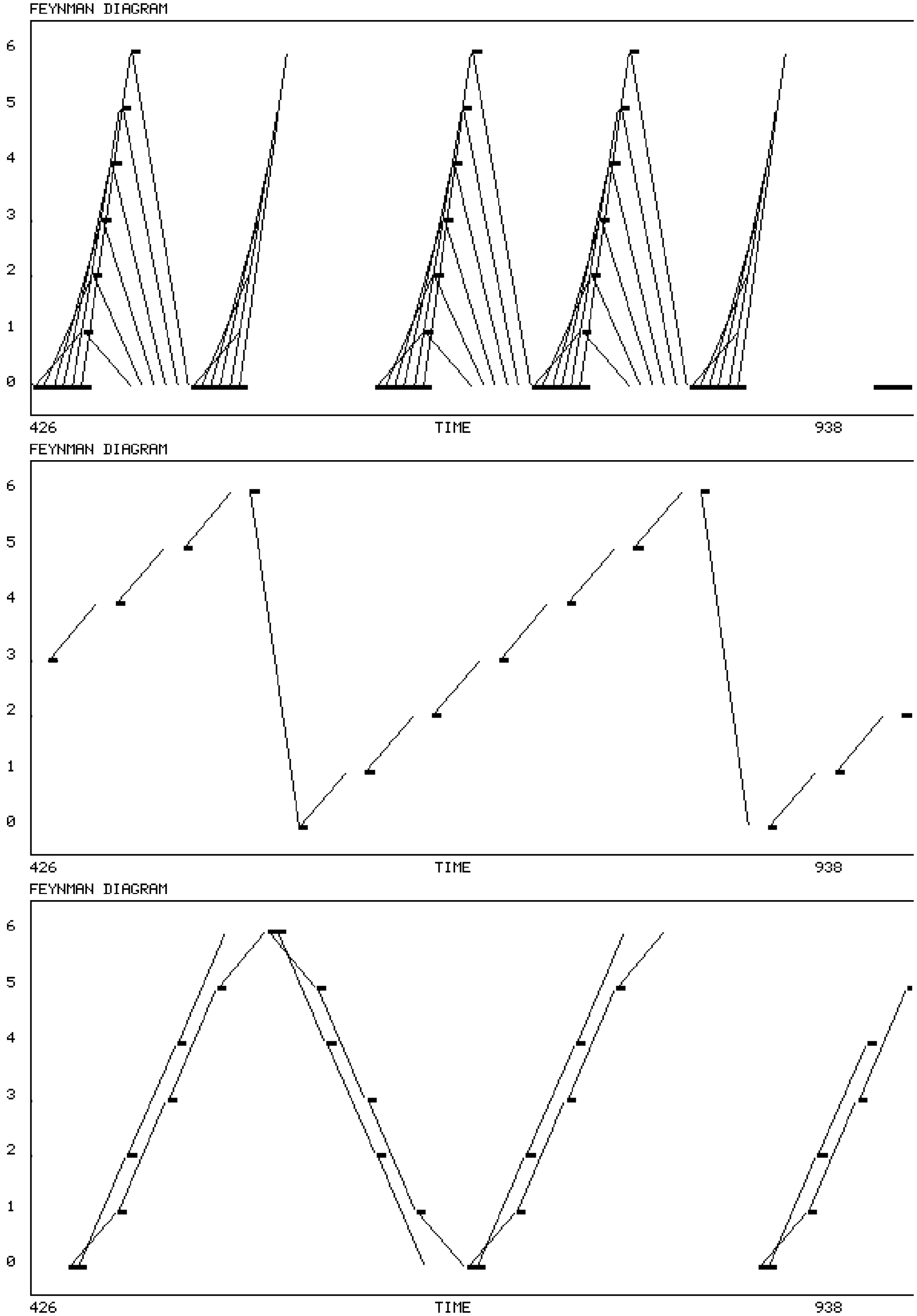


Figure 7.3: Comparing the performance of three GVT algorithms using the ParaGraph visualisation tool

Chapter 8

Compilation rules

8.1 Introduction

This chapter describes how the state-transition model can be used to model a compilation system. Particular emphasis is placed on encoding important optimisations, including register allocation, closure layout, and dead-code elimination. The validity of this approach is demonstrated by developing and prototyping a compilation system for a subset of the sequential STG$'$ language.

Section 8.2 motivates the selection of a RISC assembly language as the target for the compilation system, while the proposed state-transition system is described in section 8.3 (all of the rules are collected in appendix H.) Section 8.4 then considers the development of the run-time support for the generated code, before the chapter is concluded in section 8.6.

8.2 Targeting a RISC assembly language

The two most common target languages for compilers of functional programming languages are C [Kernighan and Ritchie, 1978] and assembly language. Bartlett [1989] cites the following advantages to using C:

portable most modern computers provide a C compiler.

high level many of the technical aspects of efficient code generation will be handled by the C compiler (and others' improvements to this technology will be passed on to the new system)

easy to interface with C typically, if a system provides an inter-language interface it will be modelled on C's calling convention. For example, Glasgow Haskell provides the `ccall` and `casm` primitives [AQUA Team, 1993, section 3.2.3, pages 33–34]), and the Unix operating system uses a C-style interface [Leffler, McKusick, Karels and Quarterman, 1989, chapter 1, page 3]

Peyton Jones et al. [1993, section 6.2] also point out the following, unexpected, benefit:

debugging source-level debuggers, such as `gdb`, simplify the testing process.

The only drawback to the high-level approach is the potential loss of performance, but, where the two language models are similar, this cost is small. For example, depending upon the C compiler used, Bartlett [1989, section 4, pages 20–22] observed either a 5% slowdown or 8% speedup over a traditional Scheme compiler. On the other hand, GHC requires a jump statement to efficiently implement the *Entry* mechanism of the STG machine, and going via C would impose a "considerable" overhead [Peyton Jones et al., 1993, section 6.1]. To overcome this problem, GHC uses non-standard features of the GNU C compiler to explicitly manage the register mapping, thereby circumventing the standard calling convention. The complexity of the resulting implementation [AQUA Team, 1994] is such that the decision to target the C language can be questioned.

Therefore, the reasons for the adoption of a RISC-like assembly language as the compiler's target can be stated as follows:

expressiveness assembler traditional provides finer control over the layout of the data and code components of a program. This allows a number of optimisations to be expressed, including register allocation, and reversed info tables.

simplicity a RISC instruction set is regular, thereby simplifying register-allocation and instruction-scheduling algorithms.

portability as a direct result of simplicity, converting to a CISC-like language should be straightforward.

accuracy both Appel [1992, section 15.1, page 182] and Santos [1995, section 2.3, page 13] use the total number of assembler instructions executed as a primary metric for performance evaluation.[1]

Note that, amongst others, Standard ML of New Jersey [Appel, 1992, section 14.3, pages 169–174], rationalised Tim [Chitnis et al., 1995, section 3.2, page 96] and WYBERT [Langendoen, 1993, figure 5.5, page 100] all use a RISC-like target language.

From a modelling perspective, however, there is one potential problem with the assembly-language approach: developing the run-time support systems can be tedious, error prone, and time consuming. This issue is addressed in section 8.4.

8.3 Prototyping a modern optimising compiler using a state-transition system

The work described in this section is based on the third part of the original STG report [Peyton Jones and Salkild, 1989, sections 6–11, and appendix A], and also draws on the techniques described in chapter nine of the dragon book [Aho, Sethi and Ullman, 1986, pages 513–584].

The prototype system uses the following state to structure the compilation:

$$\boxed{\left(\begin{array}{llllll} & \mathit{expression} & \mathit{continuation} & \mathit{pending} & \mathit{code} & \mathit{global} \\ \mathit{code}, & \mathit{stack}, & \mathit{stack}, & \mathit{bindings}, & \mathit{blocks}, & \mathit{environment} \end{array}\right)}$$

The individual components are specified in table 8.1, while section 8.3.1 describes the *code* component in greater depth. The associated rules, collected in appendix H, are only a subset of what would be required for a complete compilation system, with the most notable omissions being the rules dealing with constructors and higher-order functions. The development of the missing rules should not be difficult.

[1] The simulator described in chapter 7 originally used a C-like language (see section A.4.3 for further details), but it proved difficult to cost the different statements and expressions.

	specification	description
code	control flow and expression compilation – see section 8.3.1	
expression stack	stack of *instructions*	results of sub-compilations are returned on this stack
continuation stack	stack of *code*	the return stack whereby control reverts to the initiator of a sub-compilation
pending bindings	set of *bind*	global and local definitions awaiting compilation
code blocks	$\{label \mapsto instructions\}_{env}$	accumulates the output of the compilation system
global environment	σ *var* = *label*	records the static addresses of all top-level closures
instructions	sequence of *instruction*	a basic block [Aho, Sethi and Ullman, 1986, section 9.4]
instruction	a RISC instruction – see appendix F	
label	an operand – see section 8.3.2	

Table 8.1: The state components of the compiler framework

8.3.1 The code component

The system has two modes of operation, compiler control and expression compilation, and both are detailed in table 8.2. The former codes manipulate bindings and provide flow control, while the rules associated with the latter codes are closely related to those of their STG-machine counterparts,[2] as demonstrated by the following rule for compiling literal expressions:

9	*CEval* (*k*)	ρ	*code*	*returns*	*exps*	*conts*	*pending*	*blocks*	σ
$\Longrightarrow$	*CReturnInt k*	ρ	*code*	*returns*	*exps*	*conts*	*pending*	*blocks*	σ

In some cases, however, it has been necessary to introduce an extra stage, as illustrated by the splitting of the STG-machine *Enter* code into the *CEnter* and *CJoinEnter* codes of the compilation system.

[2] Indeed, the compilation rules have been numbered in accordance with their STG-machine equivalents – see appendix H.

Compiler control

Continue	sets the next command to be either: 1. the top item on the continuation stack. 2. if the continuation stack is empty, then select a binding from the set of those pending compilation, and set the next command to *CompileBind*. 3. if the set of pending bindings is empty (and the continuation stack is finished) then the next command is set to *Finish*.
CompileBind	initiates the compilation of a STG binding
ReturnExpression	returns the instructions needed to evaluate an expression sequence and allows fine-tuning at this level (including common peephole optimisations).
SealEntry	complements *CompileBind* in that it allows any pre- or post-amble to be added to the main body of a function. This could include, for example, stack, heap and argument checks or (simple) interface manipulations.
ReturnBind	similar to *ReturnExpression*, except the instructions encode a whole binding, either top-level or bound by a let(rec) expression. This command is a good point at which to generate info tables, static heap entries, and specialised garbage-collection routines.
Finish	indicates that the compilation process has completed successfully.

Expression compilation

CEval	generates the RISC instructions needed to evaluate a given expression sequence.
CEnter	determines the calling mechanism for a non-literal variable
CJoinEnter	glues together code either side of a non-local application.
CReturnCon	determines what return mechanism to use for the given type and is the dual of the *CJoinReturns* instruction. Together they realise the behaviour of the *ReturnCon* code of the operational semantics.
CReturnLit	has the same effect as *ReturnCon* except it deals with literal values.
CJoinReturns	combines all of the specified alternatives of a case expression into one return method. The spectrum of possible methods is delimited by vector and in-line returns.

Table 8.2: The code component of the compilation state-transition system

8.3.2 Operands and register allocation

The compilation system uses the following operands:

operand	STG-machine C-code equivalent	description
$register_n$	`var`	contents of the nth general-purpose register
$stack_n^{A\|B}$	`SpA[n], SpB[n]`	the value stored in the nth slot of the A (boxed) or B (unboxed) stack
$heap_{offset}$	`Hp[offset]`	the address created by adding $offset$ to the heap pointer
$memory_{offset}^{operand}$	`operand[offset]`	contents of the memory location specified by adding *offset* to *operand*
$label_{name}$	`&name`	a named label pointing to a static address, which may reference an entry routine, an information table, a jump table etc.
literal	`1, ..., UINT_MAX`	a constant integer value

A modern RISC processor will typically provide either 32 or 64 general-purpose registers, although a number of these are reserved by the compilation system for holding important values, as shown below:

register	1–23	24	25	26	27	28	29	30	31
use	general purpose	`Ret`	`Np`	`StkA`	`StkABase`	`StkB`	`StkBBase`	`HLimit`	`Hp`

The general purpose registers, in combination with the node pointer, `Np`, and stack pointers, simulate the local environment of the STG machine.

Furthermore, depending upon the entry and return conventions other registers may have special meanings (see rules 1–2c and 11a–13′). For example, upon entry to a closure (see the following section), register 24 will hold the return address (rule 2c), and the node pointer, `Np`, will point to the base of the closure (rule 1).

While calling conventions rigidly define the location of certain values upon entry and exit of a basic block, the strategy for making the best use of the general-purpose registers within the block itself is known as register allocation [Aho, Sethi and Ullman, 1986, section 9.7]. As demonstrated by Fraser and Hanson [1992], even simple allocation schemes can be effective. Despite the maturity of such algorithms for imperative languages, their functional counterparts have received little attention, with notable exceptions including the work of Boquist [1995] and Appel [1992, chapter 11].

8.3.3 Counters, timers and interrupts

As discussed in section 9.3.2, it is often useful to interrupt the current thread of control, perform some task, and then continue as before. Unfortunately, this can significantly complicate the run-time code [Axford, 1989, section 1.2], and, therefore, the compilation system itself. The Gambit compiler [Feeley and Miller, 1990] neatly avoids these problems by inserting tests after every basic block – the tests, and any handlers they may invoke, can then assume that the system is in a stable state. To ensure that the tests are performed in a timely manner, it may be necessary for the compilation system to split large blocks into a number of smaller ones.

8.3.4 The fac function

Figure 8.1 shows the output of the compilation system for the specialised `fac` function shown below:

STG′ code

```
fac = [] \r [n] -> case n of
                   {
                    0# -> 1#;
                    _  -> let# n_less_one = minusInt# [n, 1#] in
                          let# fac_n_less_one = fac n_less_one
                          in timesInt# [n, fac_n_less_one]
                   };
```

$label_{entry\_fac}$: // standard entry point
$compare_{x=y}\ register_{stkB}, register_{stkBLimit}, register_{tmp_1};$ // check the number of arguments
$branch_{bit0\_set}\ register_{tmp_1}, label_{PAP\_update}$ // insufficient arguments
$load\ +4(register_{stkB}), register_{tmp_2};$ // load the argument **n**
$subtract\ register_{tmp_2}, +1, register_{tmp_2};$ // test if **n** is one
$branch_{x=0}\ register_{tmp_2}, label_{fac\_1};$ // if so, select the first alternative
$branch\ label_{fac\_2};$ // otherwise, select the default

$label_{fac\_1}$: // code to handle first alternative
$add\ 0, +1, register_{return\_Int\#};$ // set the result to one
$add\ register_{stkB}, +4, register_{stkB};$ // trim the stack
$jump\ register_{return};$ // invoke the return continuation

$label_{fac\_2}$:
$subtract\ register_{stkB}, +8, register_{stkB};$ // allocate stack space
$compare_{x<y}\ register_{stkB}, register_{stkA}, register_{tmp_3};$ // check for stack overflow
$branch_{bit0\_set}\ register_{tmp_3}, label_{stack\_overflow};$ // overflow error handler
$load\ +12(register_{stkB}), register_{tmp_4};$ // load the argument **n**
$store\ register_{return}, +12(register_{stkB});$ // save the original continuation
$store\ register_{tmp_4}, +8(register_{stkB});$ // save the original argument
$subtract\ register_{tmp_4}, +1, register_{tmp_4};$ // calculate **(n-1)**
$store\ register_{tmp_4}, +4(register_{stkB});$ // push the new argument
$load\ label_{fac\_2\_1}, register_{return};$ // set the new continuation
$branch\ label_{entry\_fac}$ // call **fac** on **(n-1)**

$label_{fac\_2\_1}$: // continuation code
$load\ +4(register_{stkB}), register_{tmp\_a};$ // restore the value of n
$mult\ register_{tmp\_a}, register_{return\_Int\#}, register_{tmp\_a};$ // multiply n by `fac (n-1)`
$move\ register_{tmp\_a}, register_{return\_Int\#};$ // set the result
$load\ +8(register_{stkB}), register_{return};$ // restore the originial continuation
$add\ register_{stkB}, +8, register_{stk\_B};$ // trim the stack
$jump\ register_{return};$ // invoke the return continuation

Figure 8.1: Unoptimised RISC code produced by the compiler for the `fac` function

Appendix I contains a number of other examples, including the `nofib` programs, `fib`, and `primes`, as well as common prelude functions such as `map` and `quotRem`. In addition, it also include the code required to update partial applications and algebraic constructors.

8.4 Run-time support

Run-time support covers both the traditional operating-system libraries (including message passing and thread management) as well as the more specialised capabilities, such as distributed garbage collection [Lester, 1989; Trinder, Hammond, Partridge, Peyton Jones and others, 1996, section 2.3.3] and load balancing (see section 9.3.2). The compilation rules access both types of libraries using application-program interfaces (API) similar to those described in section 7.4.3, but extended to include the static label of the appropriate code block.

While generating a specific API will be straightforward, the implementation in RISC assembler is likely to be tedious, time consuming, and error prone. While it may be tempting to provide the functionality via the simulator's *syscall* interface – thereby enabling the use of Haskell – this should only be used for the operations described in section 7.3.2. Apart from "feeling" wrong, abusing the *syscall* mechanism could affect the accuracy and correctness of the simulation as each such operation is atomic.

The simple solution to the above-mentioned problem is to use an existing compiler to generate the assembly code from, for example, a C implementation of the function. The `lcc` re-targetable C compiler [Fraser and Hanson, 1991] is an obvious candidate as it supports cross compilation to MIPS assembler [Kane and Heinrich, 1992]. Moreover, `lcc` only performs simple peephole optimisations, thereby maintaining the correspondence between the source and output codes. Some editing of the resulting code will be required, but there is a considerable net saving in both time and effort, and increased confidence in the correctness of the generated assembler code.

8.5 Benchmarking the nofib routines

Tables 8.5 and 8.5 contain the RISC instruction counts when running unoptimised and optimised versions of the `fib`, `primes`, and `queens` benchmarks. The total instruction count is broken down into the following categories for each benchmark:

computation includes the numerical and logical operators, add, multiply, exclusive or, shift left, etc. These instructions are primarily used when performing argument checks, trimming the stack, and allocating memory. The computation performed as a result of primitive STG′ operations is typically less than 10%.

memory includes both loads and stores. Loads are used to retrieve stack parameters, and access data from the heap (typically info tables and free variables). Stores are used to push data onto the stack and to initialise or update closures. Within the benchmarks the ratio between loads and stores is approximately 50% (with the exception of the optimised `queens`, where loads account for 60% of the total).

calls include both branches (*BR* and *JMP*) and subroutine calls (*BSR* and *JSR*). Branches are used to call known entry points and to return to the correct vector entry. Calls are used to enter closures for single constructor data-types, such as *Integer* and *Boolean*. The ratio between branches and calls is typically between 70% and 80%, although for the optimised `fib` the ratio drops to 60%.

benchmark	total	computation	memory	calls	immediates	conditionals
fib 5	2 676	31·6%	41·3%	13·6%	6·8%	6·7%
fib 10	32 565	31·7%	41·4%	13·6%	6·5%	6·8%
fib 15	363 927	31·7%	41·5%	13·6%	6·5%	6·8%
queens 5	176 314	28·5%	46·4%	12·4%	6·2%	6·5%
queens 6	849 691	28·4%	46·6%	12·5%	6·0%	6·5%
primes 50	299 406	27·7%	47·0%	11·9%	7·0%	6·4%
primes 100	1 077 037	27·6%	47·1%	12·0%	6·9%	6·4%

Table 8.3: RISC-instruction counts for the unoptimised benchmarks

benchmark	total	computation	memory	calls	immediates	conditionals
fib 5	230	40·4%	27·8%	19·6%	5·2%	7·0%
fib 10	2 174	45·3%	25·3%	20·7%	0·6%	8·2%
fib 15	23 726	45·8%	25·0%	20·8%	0·1%	8·3%
queens 5	42 239	25·1%	47·8%	13·4%	8·2%	5·5%
queens 6	167 527	24·9%	48·1%	14·0%	7·6%	5·4%
primes 50	190 890	27·9%	45·6%	11·4%	6·9%	8·2%
primes 100	673 132	27·7%	45·9%	11·4%	6·7%	8·3%

Table 8.4: RISC-instruction counts for the optimised benchmarks

immediates represents the loading of numeric constants via the load-address instructions (*LA* and *LAH*). These instructions are typically used to load the address of the info tables when initialising heap-allocated closures. As they tend to appear in pairs, halving the number of immediate instructions provides a good estimate of the total number of heap allocations.

conditionals includes both the conditional move (*CMOVE*) and the conditional jump (*CBR*). These are used when testing for stack and heap overflows, and for implementing simple case expressions.

Table 8.5 compares the total number of RISC instructions for each benchmark to the total number of reduction steps performed by the STG machine. The bracketed numbers denote the ratio between these two counts, and, with the exception of the optimised fib results, the instruction-level simulation performs two to four times the number of steps of the STG machine. Furthermore, the amount of memory required to simulate the RISC machine is upto twenty times greater than that for the STG machine. The net effect is that the RISC simulator runs considerably slower than the STG machine, and can therfore only be used to evaluate smaller problems.

8.6 Summary

This chapter has described a state-transition model of a modern optimising compiler, which is closely related to the STG-machine (see chapter 6). To demonstrate the viability of the resulting rules, a prototype compiler has been developed. The results from benchmarking

	STG		RISC	
benchmark	unoptimised	optimised	unoptimised	optimised
fib 5	771	211	2 676 (3.5)	230 (1.1)
fib 10	9 357	2 479	32 565 (3.5)	2 174 (0.9)
fib 15	104 545	306 475	363 927 (3.5)	23 726 (0.1)
queens 5	38 630	16 863	176 314 (4.6)	42 239 (2.5)
queens 6	188 174	75 102	849 691 (4.5)	167 527 (2.2)
primes 50	96 374	79 032	299 406 (3.1)	190 890 (2.4)
primes 100	348 835	286 485	1 077 037 (3.1)	673 132 (2.4)

Table 8.5: Comparing STG machine reductions and RISC instructions

the compiled versions of the nofib programs (fib, queens, and primes) show that the instruction-level simulation requires between two and four times as many cycles as the STG machine. This reduces the problem size that can be examined at this level of detail.

Chapter 9

Prototyping parallel functional intermediate languages

9.1 Introduction

In this chapter the use of the prototyping framework is illustrated by four case studies. Each of the studies are based upon existing well-known systems, and, between them, include examples of the main programming abstractions used in modern parallel functional programming (see section 2.4) and cover both GMSV and DMMP architectures (see section 2.2.1). The first (section 9.2) is based upon shared-memory Haskell [Mattson Jr., 1993a], and considers the introduction of parallel threads into the STG′ language. This provides a simple overview of the methodology, and serves as a foundation upon which the other case studies build. The second (section 9.3) moves on to consider GUM Haskell [Trinder et al., 1996], essentially a DMMP implementation of the previous study. While the static semantics are very similar to those of the first case study, the operational model is far more complex, and demonstrates how message passing can be modelled by a state-transition system. The third (section 9.4) investigates the data placement primitives of para-functional Haskell [Hudak, 1991]. These prove interesting both in terms of the denotational and operational models. Skeletal parallelism [Cole, 1989] is the subject of the final case study (section 9.5), dealing with farms, pipes and divide-and-conquer skeletons [Darlington et al., 1993].

9.2 Mattson's speculative evaluation technique

Under the *evaluate-and-die* model [Peyton Jones, 1989, page 178], a thread is an independent process which computes the value of one expression and then terminates. This approach to thread management has been adopted by most modern systems, including GUM [Trinder et al., 1996, section 2.2], the JUMP* machine [Chakravarty, 1994, section 2.3.2], and the ν-STG machine [Hwang and Rushall, 1992, sections 6–8].

Traditionally, only expressions essential to the main computation are candidates for threads. By sparking non-essential expressions, speculative systems increases the number of available threads, thereby decreasing the chance that any processor is idle. However, there is a chance that the time and space expended on the computation will be wasted, and complications arise when a speculative task is detected to be either necessary or irrelevant.

The system presented in this section is primarily based on Mattson's speculative graph

STG′ code

```
spec_map = [] \r [f xss] -> case xss of
 { Nil       -> Nil [] ;
   Cons x xs -> letspec 90% { xs' = [f xs] \u [] -> spec_map f xs ; }
                in let      {  x' = [f  x] \u [] -> f x ; }
                in Cons [x', xs'] ;
 };
```

Figure 9.1: A speculative STG′ version of the map function

reducer [Mattson Jr., 1993a, section 4.3, pages 69–80] and the GRIP multiprocessor [Peyton Jones et al., 1987; Mattson Jr., 1993b, sections 2–3].

9.2.1 The static semantics

Speculative parallelism is introduced into the STG′language by extending the *exp* production rule (see section 5.2.1) as follows:

$$exp \quad \longrightarrow \quad \texttt{letspec}\ literal\ simple\_bind\ exp \mid \cdots \qquad \text{speculative evaluation}$$

The *literal* value should be between 0–100, and estimates the percentage probability that the bound expression will be required as part of the main computation. Note, that this relates to the traditional `letpar` as follows:

$$\texttt{letpar}\ simple\_bind\ exp \ \equiv\ \texttt{letspec}\ 100\ simple\_bind\ exp$$

As an example, figure 9.1 shows a speculative variant of the map function, which estimates that the first element of the tail will be required 90% of the time. When a speculative thread evaluates a `letspec` expression, the effective probability of the new thread is the product of the probabilities of the current thread and the specified probability – any thread with a probability less than 10% is ignored. Therefore, the speculative map will evaluate a list up to a maximum depth of 21 elements (if the probability were changed to 50%, then a maximum of 3 elements would be evaluated). The free variables of the new expression are determined by the following equation:

$$\begin{aligned} &\mathcal{FV}_{exp}[\![\texttt{letspec}\ literal\ var = exp_{rhs}\ exp_{body}]\!]\ g \\ &\quad = \mathcal{FV}_{exp}[\![exp_{rhs}]\!]\ g \cup (\mathcal{FV}_{exp_{body}}[\![exp]\!]\ g \setminus \{var\}) \end{aligned}$$

The denotational semantics of the new expression is shown below:

$$\begin{aligned} \mathcal{E}[\![\texttt{letspec}\ literal\ var = exp_{rhs}\ exp_{body}]\!]\ \rho \ = \ & let\ \epsilon = \mathcal{E}[\![exp_{rhs}]\!]\ \rho \\ & in\ if \quad (literal \geq 100) \bigwedge\ (\epsilon = \perp) \\ & \quad then\ \perp \\ & \quad else\ \mathcal{E}[\![exp_{body}]\!]\ (\rho \overset{\rightarrow}{\oplus} \{var \mapsto \epsilon\}) \end{aligned}$$

Notice that a test for bottom is only made if the thread is guaranteed to be required. Otherwise, a non-terminating speculative expression can only affect the result of the entire program if it turns out to be required, or if evaluation of the expression causes the system to run out of resources. The denotational semantics captures the first of these conditions, but cannot express the second.

$$
\mathit{LETSPEC\text{-}EXP}\qquad \frac{\begin{array}{l} \overset{\text{literal}}{\vdash}\ \mathit{literal} : \mathit{Int\#} \\ TE \overset{\text{simplebind}}{\vdash}\ \mathit{simplebind} : (\mathit{var}, \chi\ \pi_1 \ldots \pi_v) \\ LVE = \{\mathit{var} \mapsto \chi\ \pi_1 \ldots \pi_v\} \\ TE \overset{\rightarrow}{\oplus} LVE \overset{\text{exp}}{\vdash} \mathit{exp} : \tau_{exp} \end{array}}{TE \overset{\text{exp}}{\vdash} \texttt{letspec}\ \mathit{literal}\ \mathit{simplebind}\ \mathit{exp} : \tau_{exp}}
$$

Figure 9.2: The Hindley–Milner type rule for the `letspec` expression

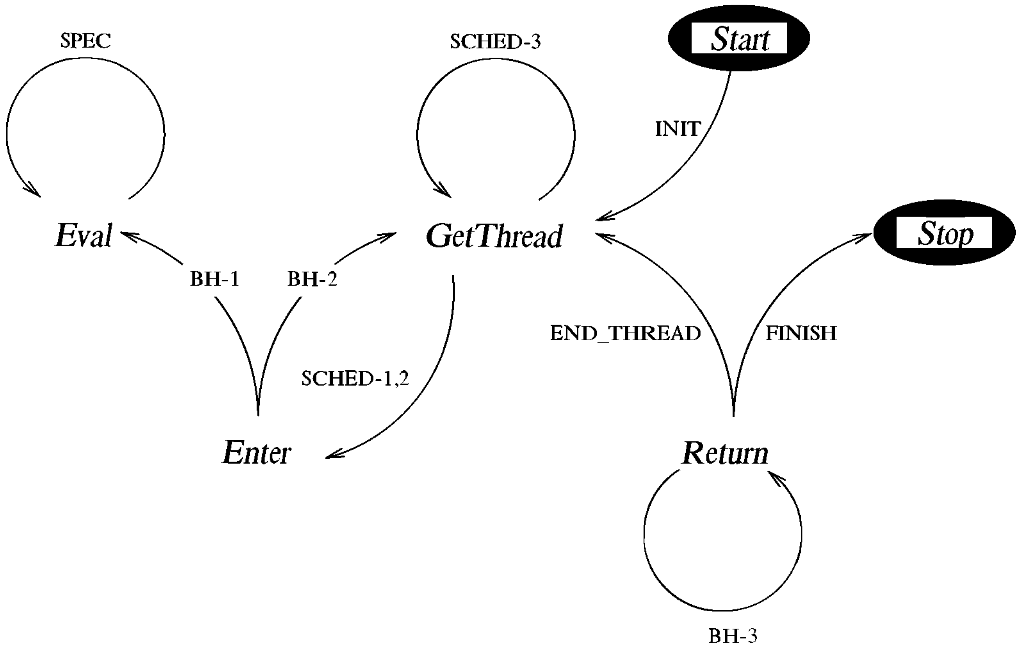


Figure 9.3: The relationship between the GMSV rules and the *code* component

The type rule, shown in figure 9.2, asserts that the probability is an unboxed integer, and that the bound expression must be a data constructor. The reasoning behind the latter restriction is described in section 4.5.1. Section 5.3.1 discusses ways in which the range restriction on the percentage probability could be enforced.

9.2.2 The operational model

The abstract state is defined in table 9.1, and the relationship between the *code* field and the new rules is illustrated in figure 9.3 (see also figure 4.11). An overview of the rules can be found in table 9.2. With the exception of rules BH_1 and BH_2, all of the rules are additions to the original STG machine – the two black-hole rules replace rule 15 and 16 respectively (which handle the entry and updating of thunks). The following sections look at these rules in greater detail.

	specification	description
G	$(P_1, \ldots, P_n)$ wp h σ	a collection of processors, P_i, all sharing a global work pool, wp, memory, h, and environment, σ.
P	$(code, \ldots, t_{id}, prob)$	the standard STG abstract state extended to include the *id* of the the currently active thread and its probability. Extensions have also been made to the *code*, *closure*, and *continuation* components.
t_{id}	a	a thread's identifier is the address of its heap-allocated state object, TSO.
wp	$(threads, sparks)$	the tasks currently available to the system.
$threads$	queue of $(t_{id}, prob)$	a collection of threads ordered by the threads' probabilites.
$sparks$	sequence of $(a, prob)$	pointers to closures whose values may be required as part of the main computation, ordered by probability.
code	*GetThread*	schedule the next thread to be run.
closure	TSO $prob$ $(code, as, rs, us)$	represents the state of a thread, which comprises its probability, an instruction sequence, and the three standard stacks.
	BlackHole t_{id} *threads*	records the *id* of thread which created the black hole, and any threads which are awaiting the final value of the closure.
	Active | *Stopped*	these values will only ever be stored at address a_{status}, and are used to indicate the current state of the computation.
continuation	*EndThread*	terminate the current thread.
	Finished	terminate the entire computation.
prob	0–100	likelihood that a thread will be required as part of the main computation. The operational model uses the probability as an indication of a threads importance or priority.

Table 9.1: State components of a thread-management system

category	rule	description
evaluation	SPEC	evaluates the `letspec` expression, creating new sparks for use by the scheduler.
synchronisation	BH_1	black holes thunks upon entry
	BH_2	suspends the current thread upon entry to a black hole
	BH_3	updates a black hole, releasing all suspended threads.
resource management	SCHED_1	converts a spark to a thread.
	SCHED_2	schedules an existing thread.
	SCHED_3	busy-wait for new work.
initialisation/ termination	INIT	static partitioning of the STG machine state.
	FINISH_1	signal the end of the computation.
	FINISH_2	detect the end of the computation.

Table 9.2: Overview of the STG rules for Mattson's speculative evaluation engine

Thread creation

Thread creation, often referred to as *sparking*, is a two-stage process as shown in figure 9.4. The first step is to identify the necessary and speculative expressions, as demonstrated by the SPEC rule:

$$
(\textsc{spec})\quad
\begin{array}{|llllllllll|}
\hline
\mathit{Eval}\ (\texttt{letspec}\ \mathit{prob}\ v = e_1\ e_2)\ \rho & \mathit{as} & \mathit{rs} & \mathit{us} & t_{id} & p & (\mathit{tp}, \mathit{spk}) & h & \sigma & \\
\text{such that } (\mathit{prob}' \geq 10) & & & & & & & & & \\
\Longrightarrow\ \mathit{Eval}\ e_2\ (\rho \overset{\rightarrow}{\oplus} \{v \mapsto a\}) & \mathit{as} & \mathit{rs} & \mathit{us} & t_{id} & p & (\mathit{tp}, \mathit{spk}') & h' & \sigma & \\
\text{where } \ \mathit{prob}' \ = \ p * \mathit{prob}/100 & & & & & & & & & \\
\qquad\quad h' \ = \ h[a \mapsto \mathit{create\_closure}\ e_1\ \rho] & & & & & & & & & \\
\qquad\quad \mathit{spk}' \ = \ \mathit{insert}_{\mathit{spark}}\ (a, \mathit{prob}')\ \mathit{spk} & & & & & & & & & \\
\hline
\end{array}
$$

This operation is very cheap as it only involves a heap allocation and the addition of the closure's address and probability to the spark pool, *spk*. The $\mathit{insert}_{\mathit{spark}}$ function maintains the correct ordering of the pool, thereby ensuring the spark with the highest probability appears at the head of the queue. On a single processor system, this rule is equivalent to a normal `let` expression, as the spark and thread pool will never be used. The associated closure may be evaluated as part of the normal computation, but this will happen within the main thread.

The second part of the *sparking* process involves the closures stored in the spark pool being converted into threads. This occurs when the current thread either blocks or terminates (see the BH_2 and END_THREAD rules):

$$
(\textsc{sched}_1)\quad
\begin{array}{|lllllllll|}
\hline
\mathit{GetThread}\ \ \langle\rangle & \langle\rangle & \langle\rangle & t_{id} & p & (\mathit{wp}, \mathit{spk}) & h & \sigma & \\
\text{such that } (\mathit{empty}\ \mathit{tp}) \bigvee\ (\mathit{max\_prob}\ \mathit{wp} < p') & & & & & & & & \\
\Longrightarrow\ \mathit{Enter}\ a \quad \langle\rangle & \langle \mathit{EndThread} \rangle & \langle\rangle & t_{new\,id} & p' & (\mathit{wp}, \mathit{spk}') & h' & \sigma & \\
\text{where } \ (a, p') : \mathit{spk}' = \mathit{spk} & & & & & & & & \\
\qquad\quad h' = h[t_{new\,id} \mapsto \mathit{TSO}\ p'\ \mathit{init\_tso\_state}] & & & & & & & & \\
\hline
\end{array}
$$

Observe that a new thread is only created when either the work pool is empty, i.e. all existing threads have either blocked or finished, or if a higher-priority spark is available. The *TSO* closure is used to preserve the thread's local state when suspending the thread.

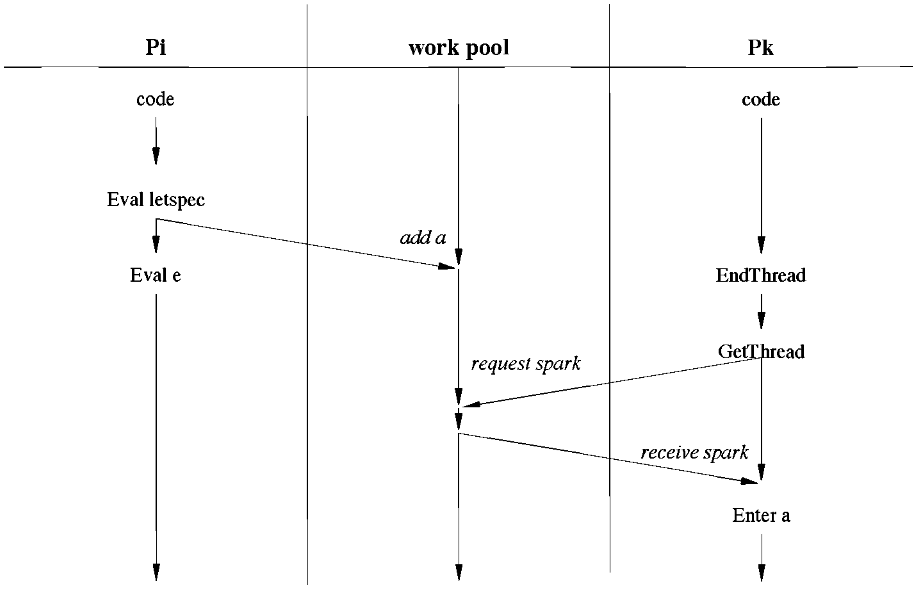


Figure 9.4: Sparking a new thread

Black holes and thread synchronisation

To prevent duplication of work, whenever a thread enters a potentially shared thunk it updates the closure with a *BlackHole*:

$$
(\text{BH}_1)\quad
\boxed{\begin{array}{lllllllll}
 & Enter\ a & as & rs & us & t_{id} & p & wp & h[a \mapsto (vs\ \mathbf{u} \rightarrow e, ws)] \quad \sigma \\
\Longrightarrow & Eval\ e\ \rho & \langle\rangle & \langle\rangle & (a, as, rs) : us & t_{id} & p & wp & h[a \mapsto BlackHole\ t_{id}\ \langle\rangle] \quad \sigma \\
\multicolumn{9}{l}{\text{where } \rho = \{v_1 \mapsto w_1, \ldots, v_n \mapsto w_n\} \text{ and } (v_i, w_i) = (vs\,!\,i, ws\,!\,i)}
\end{array}}
$$

Whenever another thread enters the black hole, its local state is saved, the thread is added to the closure's list of blocked threads, the importance of the thread evaluating the closure is increased, and a new thread is scheduled:

$$
(\text{BH}_2)\quad
\boxed{\begin{array}{llllllllll}
 & Enter\ a & as & rs & us & t_{id_1} & p_1 & wp & h & \left[\begin{array}{ll} a & \mapsto BlackHole\ t_{id_2}\ ts, \\ t_{id_1} & \mapsto TSO\ p_1\ state_1 \\ t_{id_2} & \mapsto TSO\ p_2\ state_2 \end{array}\right] \quad \sigma \\
\Longrightarrow & GetThread & as & rs & us & t_{id_1} & p_1 & wp & h & \left[\begin{array}{ll} a & \mapsto BlackHole\ t_{id_2}\ ts', \\ t_{id_1} & \mapsto TSO\ p_1\ state_1' \\ t_{id_2} & \mapsto TSO\ p_2'\ state_2 \end{array}\right] \quad \sigma \\
\multicolumn{10}{l}{\begin{array}{lcl} \text{where } ts' & = & enqueue\ (t_{id_1}, p_1)\ ts \\ state_1' & = & (Enter\ a, as, rs, us) \\ p_2' & = & max(p_1, p_2) \end{array}}
\end{array}}
$$

Notice that the importance of any threads that t_{id_2} may have sparked, or any threads upon which t_{id_2} may be waiting, are not increased – Mattson [1993a, section 3.2.4] calls this the *low-impact* model of speculative evaluation. Furthermore, there is no mechanism for reverting t_{id_2}'s priority once the closure has been evaluated (the required changes would be significant and add little to the presentation.)

When the black hole is updated all of the blocked threads are added to the work pool:

$$\text{(BH3)}\quad\boxed{\begin{array}{llllllll}
\multicolumn{2}{l}{\quad Return_{\chi}\; c\; ws \quad \langle\rangle \quad \langle\rangle \quad \begin{pmatrix} a_u, \\ as_u, \\ rs_u \end{pmatrix} : us} & t_{id} & p & (tp, spk) & h & \sigma \\
\multicolumn{7}{l}{\text{such that } h[a_u \mapsto BlackHole\; t_{id}\; ts]} \\
\multicolumn{2}{l}{\Longrightarrow\; Return_{\chi}\; c\; ws \quad as_u \quad rs_u \qquad\qquad us} & t_{id} & p & (tp', spk) & h' & \sigma \\
\multicolumn{7}{l}{\text{where } tp' \;=\; q\_append\; ts\; tp} \\
\multicolumn{7}{l}{\qquad\quad h' \;=\; h[a_u \mapsto (vs\; \texttt{r} \rightarrow c\; vs, ws)]} \\
\multicolumn{7}{l}{\qquad\quad length\; vs = length\; ws} \\
\multicolumn{7}{l}{\qquad\quad vs \text{ is a sequence of arbitrary distinct variables}}
\end{array}}$$

Terminating a thread

Once a thread has evaluated and updated its target closure, the *EndThread* continuation, pushed by the SCHED1 rule, will be invoked:

$$\text{(END\_THREAD)}\quad\boxed{\begin{array}{llllllllll}
 & Return_{\chi}\; c\; ws & \langle\rangle & \langle EndThread\rangle & \langle\rangle & t_{id} & p & wp & h & \sigma \\
\Longrightarrow & GetThread & \langle\rangle & \langle\rangle & \langle\rangle & t_{id} & p & wp & h & \sigma
\end{array}}$$

The memory occupied by the thread's *TSO* closure will eventually be reclaimed by the garbage collector, so explicit de-allocation is not necessary.

Scheduling

Whenever a thread terminates, blocks on a black hole, or a timer interrupt occurs (see section 6.2.2), a new thread is selected from the current work pool (see also SCHED1):

$$\text{(SCHED2)}\quad\boxed{\begin{array}{llllllll}
\quad GetThread & \langle\rangle & \langle\rangle & \langle\rangle & t_{id} & p & (tp, spk) & h \quad \sigma \\
\multicolumn{8}{l}{\text{such that } (\neg empty\; tp) \bigwedge (max\_prob\; spk \leq p')} \\
\Longrightarrow\; code & as & rs & us & t_{new\,id} & p' & (tp', spk) & h' \quad \sigma \\
\multicolumn{8}{l}{\text{where } \;(t_{new\,id}, p', tp') = dequeue\; tp} \\
\multicolumn{8}{l}{\qquad\quad h' = h[t_{new\,id} \mapsto TSO\; p'\; (code, as, rs, us)]}
\end{array}}$$

The *dequeue* function determines the style of scheduling amongst equal-priority threads, whether it be FIFO/LIFO (first/last in, first out) [Hammond and Peyton Jones, 1992, section 5.2], or round robin [Trinder et al., 1996, section 2.2]. Parrott [1993] outlines a system which combines risk aversion and stochastic learning which significantly outperforms a random schedule for most workloads. If no work is available, the processor busy waits:

$$\text{(SCHED3)}\quad\boxed{\begin{array}{llllllllll}
 & GetThread & \langle\rangle & \langle\rangle & \langle\rangle & t_{id} & p & wp & h & \sigma \\
\multicolumn{10}{l}{\text{such that } empty\; tp} \\
\Longrightarrow & GetThread & \langle\rangle & \langle\rangle & \langle\rangle & t_{id} & p & wp & h & \sigma
\end{array}}$$

Initialisation and termination

The initial state of an an n-processor system is defined as follows:

$$
\text{(INIT)}\quad
\boxed{
\begin{array}{lcl}
G & = & (P_1, \ldots, P_n)\ wp\ h\ \sigma \\
\multicolumn{3}{l}{\text{where}} \\
P_i & = & (\mathit{GetThread}, \langle\rangle, \langle\rangle, \langle\rangle, t_{none}, 0) \\
wp & = & (tp, \langle\rangle) \text{ where } tp = \langle(t_{main}, 100)\rangle \\
h & = & \left[\begin{array}{lcl}
t_{main} & \mapsto & \mathit{TSO}\ 100\ (\mathit{Enter}\ a_{main}, \langle\rangle, \langle \mathit{Finished}\rangle, \langle\rangle), \\
a_{status} & \mapsto & \mathit{Active} \\
a_1 & \mapsto & (vs_1\ \pi_1\ vs_1 \rightarrow exp_1, \sigma\ vs_1) \\
\ldots, & & \\
a_n & \mapsto & (vs_n\ \pi_n\ vs_n \rightarrow exp_n, \sigma\ vs_n)
\end{array}\right] \\
\sigma & = & \left\{\begin{array}{lcl}
g_1 & \mapsto & a_1, \\
\ldots, & & \\
g_n & \mapsto & a_n
\end{array}\right\}
\end{array}
}
$$

Note that all of the processors will be in competition to steal the main thread of computation. This is acceptable in a GMSV system as all messaging is implicit and guaranteed, so there is no risk of losing important data due to one processor starting before another has finished its initialisation.

The computation is finished whenever the main thread terminates:

$$
\text{(FINISH}_1\text{)}\quad
\boxed{
\begin{array}{lllllllll}
 & \mathit{Return}_{\chi}\ c\ ws & \langle\rangle & \langle \mathit{Finish}\rangle & \langle\rangle & t_{main} & 100 & wp & h & \sigma \\
\Longrightarrow & \mathit{Stop} & \langle\rangle & \langle\rangle & \langle\rangle & t_{main} & 100 & wp & h' & \sigma \\
\multicolumn{10}{l}{\text{where } h' = h[a_{status} \mapsto \mathit{Stopped}]}
\end{array}
}
$$

By signalling that the computation has ended (via the flag stored at a_{status}), the other processors can finish what they're doing and exit cleanly (using, for example, the broadcast tree outlined in section 6.3.2). As heap allocations are a frequent occurrence, and are also comparatively expensive operations, they provide a convenient place to place to check the current status:

$$
\text{(FINISH}_2\text{)}\quad
\boxed{
\begin{array}{llllllllll}
 & \mathit{Eval}\ (\texttt{let}\ \mathit{bindings}\ \mathit{exp})\ \rho & as & rs & us & t_{id} & p & wp & h & \sigma \\
\multicolumn{10}{l}{\text{such that } h[a_{status} \mapsto \mathit{Stopped}]} \\
\Longrightarrow & \mathit{Stop} & as & rs & us & t_{id} & p & wp & h & \sigma
\end{array}
}
$$

9.2.3 Compilation rules

The following sections outline the changes that need to be made to the compilation rules presented in chapter 8 to support the new operational rules.

The register map

The register map is shown in figure 9.3 and is superficially similar to that used in a purely sequential context (see section 8.3.2). The status register, `Sts`, caches the address a_{status} as it is used in every `let` expression. The work pool is accessed infrequently, and so there is no need to waste a register caching its static address. Note that replacing both `Hp` and `HpLimit` with just `HpVar` is an optimisation that is only possible at the assembly-language level.

register	1–22	23	24	25	26	27	28	29	30	31
use	general purpose	Ret	Sts	Np	StkA	StkA Base	StkB	StkB Base	Tp	HpVar

Ret	stores the address of the return handler for the current evaluation (which may be a generic update-handler, when evaluating a polymorphic thunk).
Sts	stores the address a_{status}, which is used during heap allocation to determine if the computation has finished.
Np	points to the closure which is currently being evaluated, and is used to access an expression's free variables.
StkA	points to the next available slot on the A stack. This is used in conjunction with StkB to detect stack overflow.
StkABase	points to the lower limit of the A stack, and is used to detect stack underflow.
StkB	points to the next available slot on the B stack.
StkBBase	points to the upper limit of the B stack, and is used to detect stack underflow.
Tp	points to the current thread's *TSO* closure, and, can be used, indirectly, to access the thread's priority.
HpVar	replaces both Hp and HpLimit by pointing to the address where the actual heap pointer is stored. This extra indirection is necessary as any processor can extend the heap at any time, so caching the last value seen by the local processor is not safe. The heap limit is stored in the address directly after that heap pointer, and can therefore also be accessed via the HpVar register.

Table 9.3: The register map for compiling speculative expressions

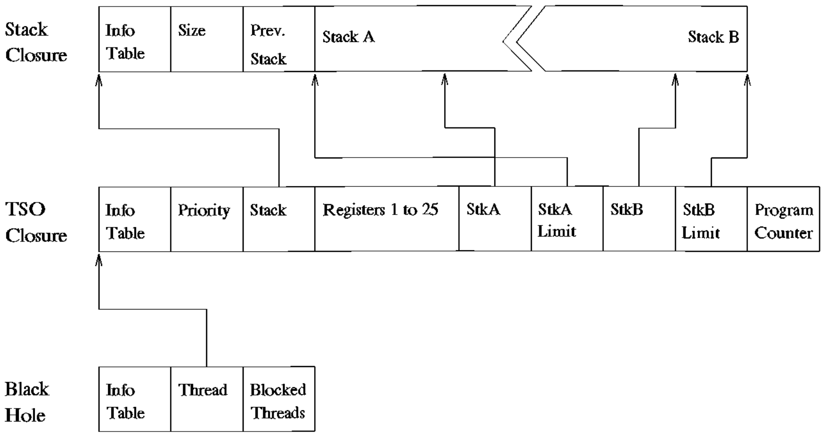


Figure 9.5: Closure layouts for a speculative GMSV system

Closure layout

Figure 9.5 shows the layout of the *TSO* and *BlackHole* closures, plus a heap-allocated stack object required to support dynamic thread creation. Note that there is no need to store the `Tp` or `HpVar` registers in the *TSO* closure, as the information they contain are trivial to compute. Furthermore, the standard garbage collection mechanisms can be used to reclaim both *TSO* closures and the associated stack space.

Communication and synchronisation

The main consideration with regards to synchronisation is access to the shared resources, namely the work pool and the global heap. All access to the former will have to be mutually exclusive[Axford, 1989, chapter 3], while only updates to the latter will need to be controlled. To illustrate the basic mechanisms, figure 9.6 shows the instruction sequence used to implement the heap allocation. As mentioned in section 6.2.4, specifying this level of detail in the operational rules would severely limit their usefulness.

New compilation rules

One new rule needs to be introduced to handle the `letspec` expression, and this is a modification of the `let` rule (see rule 3 in appendix H). The main difference between the two is that after creating the closure, the `letspec` rule generates code to add the closure's address to the spark pool:

$$\begin{array}{llllllllll}
\boxed{\text{LETSPEC}} & \mathit{CEval}\ (\texttt{letspec}\ \mathit{prob}\ v = e_1\ e_2) & \rho & \mathit{code} & rs & es & \mathit{conts} & \mathit{pending} & b & \sigma \\
\Longrightarrow & \mathit{CEval}\ e_2 & \rho' & \mathit{code}' & rs & es & \mathit{conts} & \mathit{pending}' & b & \sigma
\end{array}$$

$$\begin{array}{lll}
\text{where} \quad \rho' & = & \rho_{moves} \setminus vars_{dead} \\
\mathit{code}' & = & \mathit{code} +\!\!+\ \mathit{moves} +\!\!+\ \mathit{add\_spark} \\
\mathit{pending}' & = & \{(v, e_1)\} \cup \mathit{pending} \\
(\mathit{moves}, \rho_{moves}) & = & \mathit{allocate\_closure}\ v\ e_1\ \rho\ \sigma \\
vars_{dead} & = & \mathcal{FV}[\![e_1]\!] \setminus \mathcal{FV}[\![e_2]\!]
\end{array}$$

Note that code for checking the status flag has already been added to the heap-allocation routine, so there is no need to update the `let` compilation routines to implement the

system call	inputs		outputs	
alloc	$R21$	bytes required	$R21$	address of the allocated memory
	$R22$	return address	$R20$	corrupted
			$R19$	corrupted

```
$label_{alloc}$ :                                   // standard entry point
        $load\ (reg_{Sts}), reg_{19}$               // load the current status
        $branch_{x\leq 0}\ reg_{19}, label_{exit}$   // terminate the computation if necessary
        $load\ +4(reg_{HpVar}), reg_{19}$           // load the heap limit
        $load_{linked}\ (reg_{HpVar}), reg_{20}$    // (link) load the current heap pointer
        $subtract\ reg_{19}, reg_{20}, reg_{19}$    // has there been a heap overflow
        $branch_{x\leq 0}\ reg_{19}, label_{GC}$     // if so, invoke the garbage collector
        $add\ reg_{21}, reg_{20}, reg_{19}$         // otherwise, increase the heap pointer
        $store_{linked}\ reg_{19}, (reg_{HpVar})$   // attempt to update the heap pointer
        $branch_{x==0}\ reg_{19}, label_{alloc}$    // retry if the allocation failed
        $move\ reg_{19}, reg_{21}$                  // otherwise, set the result parameter
        $jump\ reg_{22}$                            // return to the caller
```

Figure 9.6: Heap allocation in a GMSV system

FINISH$_2$ STG-machine rule. Figure 9.7 shows a typical code fragment generated by the LETSPEC compilation rule for the `fib` benchmark (see figure 9.8).

Garbage collection

There are two issues related to garbage collection that need to be considered in a GMSV system. Firstly, the root set of the collector (see section 6.3.3) should be extended to include the thread pools, and scavenge and evacuation routines must be specified for the new *TSO* and *BlackHole* closures. Secondly, and more difficultly, a strategy for coordinating the collection phase needs to be adopted. A simple, yet inefficient, approach is for each processor to enter a barrier (see section 6.3.2 or [Almasi and Gottlieb, 1993]) once it detects that the heap has been exhausted. As heap allocation is inevitable, all processor must eventually enter the barrier. When all of the processors have entered the barrier, one processor garbage collects the entire memory as for a uniprocessor machine, and afterwards computation continues as before (see section 6.3.3 for further details). More complex schemes [North and Reppy, 1987; Lester, 1989] are beyond the scope of this thesis.

9.2.4 Performance

The STG′-equivalents of the parallel (conservative) benchmark programs, `fib` and **queens**, are shown in figures 9.8 and 9.10 respectively. Both are derived from the optimised sequential version presented in appendix B (see sections B.2 and B.4 for further details).

The `fib` benchmark

The `fib` program is often described as *embarrassingly parallel* as it produces a large number of tasks related via a simple tree structure. As such, it is often used to assess the

RISC code

```
Linfo_table_fib.wrk:

        ...                                     // header for fib.wrk
        load_address +8(R0), R21;               // set the heap-space required
        branch_link Lalloc, R22;                // allocate the space
        load_high Linfo_table_fib_n_less_2(R0), R20;
        load_address +0(R20), R20;              // load the spark's info table
        store R20, -8(R21);                     // store the info table
        store R1, -4(R21);                      // store the FV n'_less_2
        branch_link Lspark, R22;                // add the spark to the pool
        ...                                     // footer of fib.wrk

Linfo_table_fib_n_less_2:

        dw Lupdate_Int;                         // update routine
        dw Linfo_table_fib_n_less_2;            // fast entry
        dw Linfo_table_fib_n_less_2;            // stnd entry

        load_linked +0(RNp), R1;                // black hole the closure
        load +4(RNp), R2;                       // recover n'_less_2
        load_high Linfo_table_BH(R0), R1;       // load the black hole's info-
        load_address +0(R1), R1;                //   table (high+low bits)
        store_linked R1, +0(RNp);               // lock the closure
        branch_x<>0 R1, Lfib_n_less_2;          // continue if successful
        branch_link Lspin, R22;                 // random delay
        load (RNp), R1;                         // get the info table
        jump R1;                                // re-enter the closure

Lfib_n_less_2:

        subtract RStkB, +16, RStkB;             // decrease the B stack frame
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler
        store RStkABase, +16(RStkB);            // the A stack pointer
        store RStkBBase, +12(RStkB);            // the B stack pointer
        store RNp,        +8(RStkB);            // the node pointer
        store RRet,       +4(RStkB);            // the current return vector
        move RStkA, RStkABase;                  // clear the A stack
        move RStkB, RStkBBase;                  // clear the B stack
        load_high LUpdateInt(R0), RRet;         // select the integer update-
        load_address +0(RRet), RRet;            //    routine
        move R2, R1;                            // load R1 with n'_less_2
        branch Lfib.wrk +12;                    // fast-call fib.wrk
```

Figure 9.7: Example code generated by the LETSPEC compilation rule

STG′ code

```
value = [] \r [] -> Int [15#];
main  = [] \u [] -> fib value ;

fib = [] \r [n] -> case  n  of { Int n' -> fib.wrk n'; };
fib.wrk = [] \r [n'] -> case leInt# [n', 1#] of
 { True  -> Int [1#];
   False -> let# n'_less_1 = minusInt# [n', 1#] in
            letpar fib_n_less1 =  fib.wrk n'_less_1 in
            let# n'_less_2 = minusInt# [n', 2#] in
            letpar fib_n_less2 = fib.wrk n'_less_2 in
            case fib_n_less1 of { Int fib_n'_less_1 ->
            case fib_n_less2 of { Int fib_n'_less_2 ->
            let# sum_2_fibs' = plusInt# [fib_n'_less_1, fib_n'_less_2] in
            let# result = plusInt# [sum_2_fibs', 1#]
            in Int [result];         }; };
 };
```

Figure 9.8: The parallel STG′ `fib` -O benchmark

performance of an implementation under near optimal conditions. The tasks are very fine grained, typically involving just two additions. If necessary, the grain size of the computation can be controlled by restricting the depth of the tree. The relative speedups for the `fib` program running under the GMSV STG machine are shown in figure 9.9. The curves show that the system achieves near linear speedups for the larger problem sizes. For the smaller problem sizes, the amount of available work is sufficiently small that the speedup reaches a plateau after a fixed number of processors [Gustafson, 1988].

These results are consistent with those observed by Mattson Jr. [1993a, section 5.2, pages 93–94] for larger problems sizes. The problem sizes examined here had to be restricted to fifteen and under in order for the simulations to complete on a standard desktop machine (a 266Mhz PII PC with 65M of memory, running Windows NT 4.0, and using GHC 4.03). Each run completed in less than a minute. There is no reason why larger problem sizes could not be attempted on a more powerful machine.

The relative speedups for the unoptimised version of the `fib` program were very similar to those for the optimised version, and even tended to be slightly better due to the increased grain size of the computations. However, there can be no justification for using sub-optimal algorithms when evaluating parallel performance.

The queens benchmark

The **queens** benchmark is more demanding than the `fib` program described in the previous section. Firstly, far fewer tasks are generated: `fib` 15 creates approximately 2000 tasks, while **queens** 6 generates just over 150. Secondly, the dependencies between threads is more complex, with the output of one task typically depending upon the outputs of a number of other tasks. Finally, the grain size is variable and difficult to control. Unsurprisingly, this program is often used to demonstrate an implementation's performance under more challenging conditions. The relative speedups for the **queens** program running under the GMSV STG machine are shown in figure 9.11. The curves start almost linearly, but then quickly reach a plateau due to the limited number of tasks available.

Looking at the results taken by Mattson Jr. [1993a, section 5.2.2, pages 90–93], the initial parts of the curve are similar. However, with Mattson's implementation the speedup drops off with increasing processors after the maximum speedup has been achieved. The

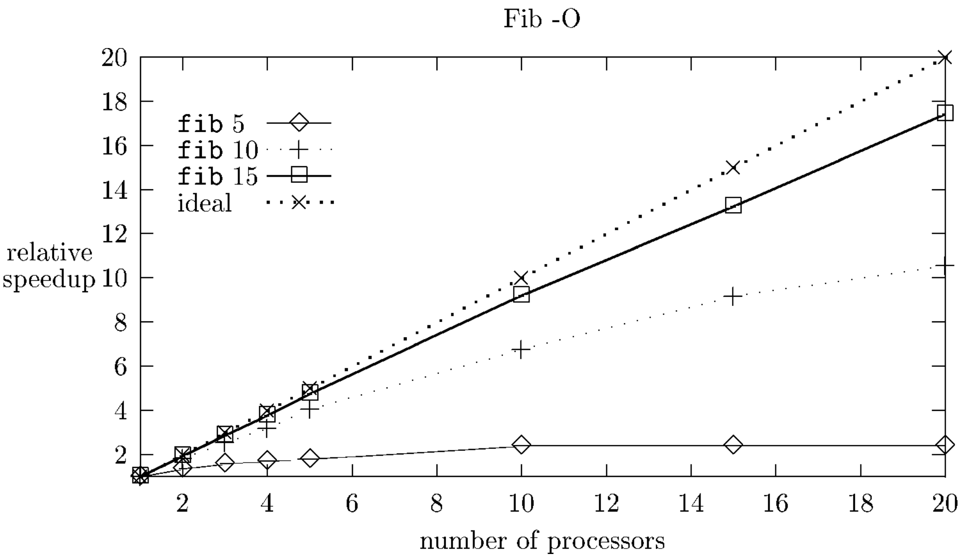


Figure 9.9: Relative speedups for the conservative `fib` -O benchmark

STG′ code

```
main =  [] \u [] ->  nsoln.wrk int 5#;

gen.wrk = [] \r [nq n] ->
  case  n  of {
   0# -> nil_nil ;
   _  -> let# dec_n' = minusInt# [n,  1#] in
         letstrict bs = gen.wrk nq dec_n' in
         let { qs = [nq] \u [] -> const.Int.enumFromTo one nq; } in
         gen_comprehension nq qs bs };

gen_comprehension = [] \r [nq one_to_nq dss] -> case dss of
 { Nil       -> Nil [];
   Cons d ds -> letpar a = gen_comprehension nq one_to_nq ds
                in  g a d nq one_to_nq;
 };
```

Figure 9.10: The core of the parallel STG′ **queens** -O benchmark

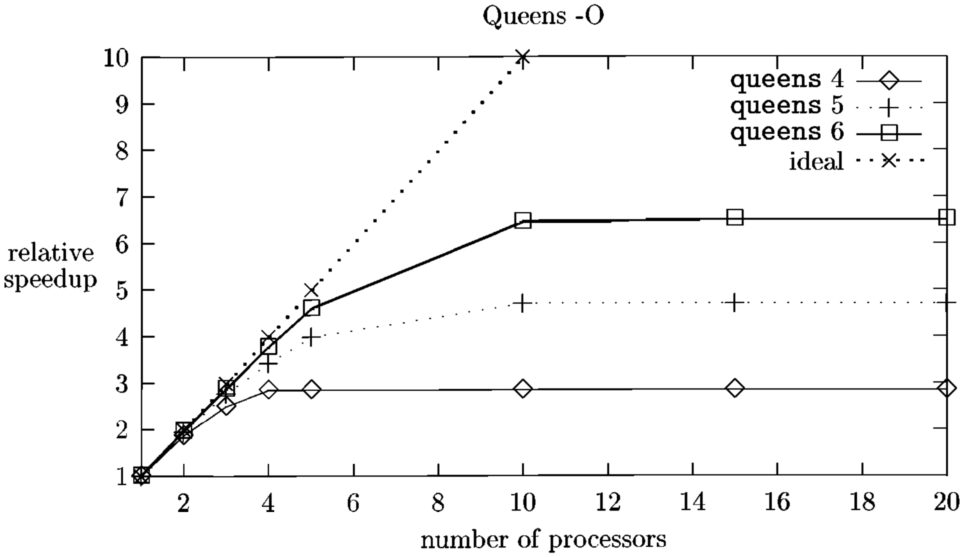


Figure 9.11: Relative speedups for the conservative **queens** -O benchmark

STG results, however, remain perfectly flat. The reason for this discrepancy is due to the STG simulation not modelling resource contention (caused by the locking mechanisms described in section 6.2.4). The RISC simulation, on the other hand, does model the first-order effects of locking, and figure 9.12 compares the STG and RISC speedups for **queens** 3.

9.2.5 Extensions

Unlike the sequential STG machine, the speculative rule set does not detect erroneously cyclic definitions of the form:

STG′ code

```
x = [] \u [] -> const.Int.+ x one;
```

One solution would be for the BH$_1$ to record the id of the thread responsible for evaluating a black-holed closure. The BH$_2$ rule could then check if the newly blocked thread is directly or indirectly responsible for the evaluation upon which it is waiting. However, the simplest solution would be to debug the algorithm on a sequential implementation which can easily detect the presence of such cycles.

The rule set also uses a very crude priority-upgrade mechanism (as did Mattson's implementation), whereby a blocked thread can boost the priority of a speculative thread currently evaluating the associated thunk (see rule BH$_2$). Currently, this increase in status is permanent. Mattson Jr. [1993a, section 3.2.4, pages 54–56] proposes a number of alternatives, including tracking the stack depth at which the speculative task entered the thunk. While the STG animation would provide an excellent environment for testing these strategies, none of Mattson's benchmark programs suffered due to the false upgrading of speculative threads.

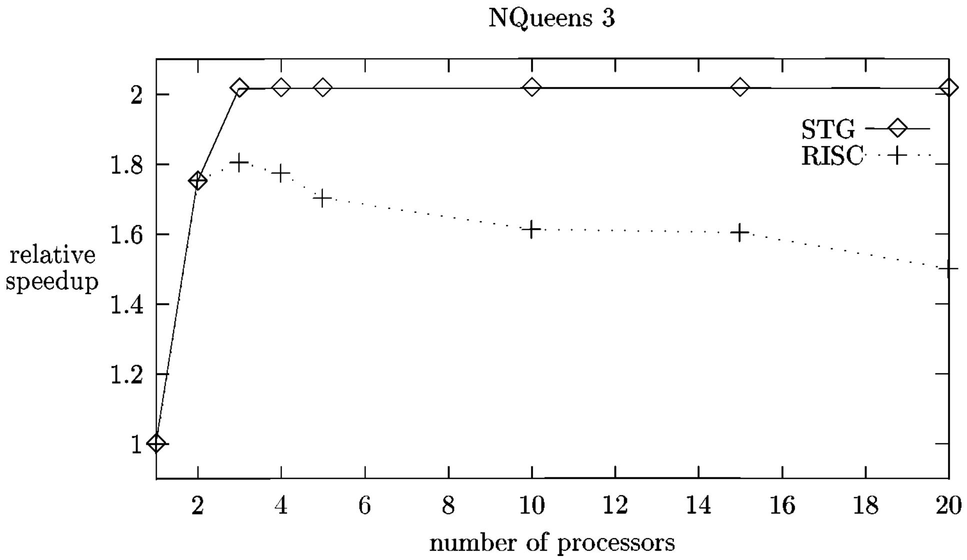


Figure 9.12: Comparing the STG′ and RISC animations for the **queens** -O benchmark

9.2.6 Assessment

Overall, despite being the first case study, the development of the static semantics, STG-machine rules and corresponding animation were straightforward extensions of their sequential counterparts. The compilation and RISC animations were more problematic, but this was mainly due to the large number of support routines that needed to be developed and tested (subsequent studies simply made use of this groundwork). While each new phase of the development process introduced additional details and complexity, the tools and descriptions developed during the previous phase provided a strong foundation upon which to build. The animations helped to test the correctness of the semi-formal specifications, and also provided valuable insight into the system dynamics. Indeed, the operational specification and STG animation were developed iteratively (as was the case with the sequential compilation rules and the RISC compiler described in chapter 8).

As previously mentioned, the development of the RISC animation was probably the most time consuming phase of the development. Fortunately, the STG animation was sufficiently accurate to allow different strategies to be compared and tested, such that only the successful candidates needed to proceed to the final (expensive) phase. However, the RISC animation does not model cache effects, and so can only be used as a rough guide.

The performance results obtained from both the STG and RISC simulations broadly agree with those observed by Mattson Jr. [1993a], although both simulators are only capable of handling significantly smaller problem sizes. The primary limitation of the STG simulations is that they ignore resource contention, and therefore do not exhibit the classic degradation of performance with increasing numbers of surplus processors.

abstract syntax	$exp \longrightarrow \texttt{letpar}\ simple\_bind\ exp \mid \cdots$ parallel evaluation
free variables	$\mathcal{FV}_{exp}\llbracket\texttt{letpar}\ var = exp_{rhs}\ exp_{body}\rrbracket\ g$ $= \mathcal{FV}_{exp}\llbracket exp_{rhs}\rrbracket\ g \cup (\mathcal{FV}_{exp_{body}}\llbracket exp\rrbracket\ g \setminus \{var\})$
denotational semantics	$\mathcal{E}\llbracket\texttt{letpar}\ var = exp_{rhs}\ exp_{body}\rrbracket\ \rho$ $= \ let\ \epsilon = \mathcal{E}\llbracket exp_{rhs}\rrbracket\ \rho\ in\ if\ \ (\epsilon = \perp)$ $then\ \perp$ $else\ \ \mathcal{E}\llbracket exp_{body}\rrbracket\ (\rho \overset{\rightarrow}{\oplus} \{var \mapsto \epsilon\})$
type inference	$LETPAR\text{-}EXP\ \dfrac{\begin{array}{l} TE \overset{\text{simplebind}}{\vdash} simplebind : (var, \chi\ \pi_1 \ldots \pi_v) \\ LVE = \{var \mapsto \chi\ \pi_1 \ldots \pi_v\} \\ TE \overset{\rightarrow}{\oplus} LVE \overset{\text{exp}}{\vdash} exp : \tau_{exp} \end{array}}{TE \overset{\text{exp}}{\vdash} \texttt{letpar}\ simplebind\ exp : \tau_{exp}}$

Figure 9.13: The static semantics for the GUM case study

9.3 GUM: Graph reduction for a Unified Machine

GUM [Trinder et al., 1996] (Graph reduction for a Unified Machine) is a DMMP implementation of Haskell, using the classic par operator to identify parallel threads. GUM is built on top of the PVM communication system [Beguelin et al., 1993]) and is therefore portable to a range of architectures, including both GMSV and DMMP machines. Notably, absolute speedups over the best sequential compilers have been observed for both high-performance shared-memory machines and clusters of workstations operating over Ethernet.

9.3.1 The static semantics

The static semantics are very similar to those presented in the previous case study, and the necessary details can be found in figure 9.13.

9.3.2 The operational model

This section presents a state-transition model of the asynchronous message-passing features of GUM. The abstract states for the processor, P, and communication system, S, are shown in table 9.4, and the relationship between the *code* field and the new rules is illustrated in figure 9.14. An overview of the rules can be found in table 9.5.

Sending and receiving messages

Unlike the previous study, all communication has to be explicit declared in a DMMP system. The model used here is based closely on that presented in section 6.2.5: all sends are asynchronous, and all receives are blocking. This section presents three of the STG rules – SEND, RECV , and BCAST. These provide convenient abstractions for use by the other rules, hiding the details of the actual network interface. Furthermore, by centralising access to the network, it is possible to modify or enhance the communication system without changing the other rules. For example, it would be straightforward to re-implement the GUM rule set using a shared-memory implementation of the messaging

	specification	description
G	$(P_1, \ldots, P_n)\ S$	a collection of processors, P_i, which have to communicate via the message-passing system, S.
P	$(code, \ldots, h, t_{id}, wp, \sigma)$	the standard STG abstract state extended to include support for a local work pool, wp (see table 9.7 for the wp-related definitions).
S	$(buffers_1 \cdots buffers_n, network)$	the message-passing system, which comprises the processor-network interfaces and a model of the communication hardware.
$buffers$	$(buffer_{in}, buffer_{out})$	the input and output message buffers for a single processor
$buffer$	queue of $(i, message)$	i is either the source or destination of the message
	$probe\ buffer_{in}\ message$	search for an entry that matches the $message$ pattern
code	*Send message code*	sends the specified message and then invokes the continuation code. Table 9.6 details the messages used by the GUM system.
	Receive message code	indicates the arrival of a message, which interrupted the execution of the specified code.

Table 9.4: State components of a message-passing system

category	rule	description
evaluation	PAR	evaluates the `letpar` expression, creating new sparks for use by the scheduler.
communications	SEND	send a message to a remote processor.
	BCAST	broadcast a message to all other remote processors.
	RECV	receive a message from a remote processor.
synchronisation	BH_1	black holes thunks upon entry
	BH_2	suspends the current thread upon entry to a black hole
	BH_3	update a black hole.
	UNBLOCK	re-activate suspended threads and blocked *Fetch* messages.
scheduling	SCHED_1	converts a spark to a thread.
	SCHED_2	schedules an existing thread.
	SCHED_3	busy-wait for new work.
load balancing	FISH_1	request work from a neighbouring procesor.
	FISH_2	forward a work-request message to another processor.
	FISH_3	receive a work-request message which originated from the local processor.
	SEND_WORK	send surplus work to a remote processor.
	RECV_WORK	receive work from a remote processor.
	RECV_ACK	receive acknowledgment of the safe arrival of a work packet.
partitioning	FETCH_1	request the value of a remote closure.
	FETCH_2	return the value of a local closure to a remote processor.
	FETCH_3	receive the value of a closure from a remote processor.
	FETCH_4	suspends a *Fetch* message when requesting the value of either a *BlackHole* or *FetchMe* closure.
initialisation/ termination	INIT	static partitioning of the STG machine state.
	FINISH_1	signal the end of the computation.
	FINISH_2	detect the end of the computation.

Table 9.5: Overview of the GUM STG rules

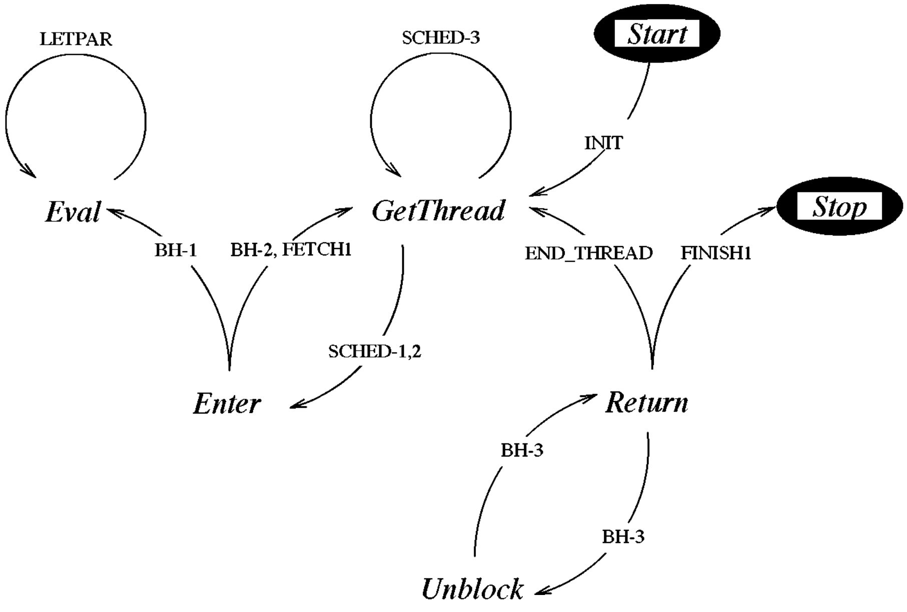


Figure 9.14: The relationship between the GUM rules and the *code* component

rules. Another possibility would be to piggy-back status information onto all outgoing messages, as discussed in section 6.3.3.

As shown in table 9.4, the network-interface comprises two buffers per processor, $(b_{in}, b_{out})_i$. The first contains all messages that have been delivered to processor i but have not yet been received (messages are added by the network and removed by the processor). The second contains all outgoing messages from processor i that have not yet been injected into the network (messages are added by the processor and removed by the network). The SEND rule, therefore, manipulates b_{out}, and the *code* continuation indicates what actions should be taken after the message has been sent:

$$
(\textsc{send}) \quad \boxed{\begin{array}{llllllllll}
 & \textit{Send message code} & as & rs & us & h & t_{id} & wp & \sigma & (b_{in},\ b_{out})_i \\
\Longrightarrow & \textit{code} & as & rs & us & h & t_{id} & wp & \sigma & (b_{in},\ b'_{out})_i \\
\multicolumn{10}{l}{\text{where } b'_{out} = \textit{enqueue message } b_{out}}
\end{array}}
$$

The BCAST rule is similar, but sends a copy of the specified message to all of the other processors:

$$
(\textsc{bcast}) \quad \boxed{\begin{array}{llllllllll}
 & \textit{Broadcast body code} & as & rs & us & h & t_{id} & wp & \sigma & (b_{in},\ b_{out})_i \\
\Longrightarrow & \textit{code} & as & rs & us & h & t_{id} & wp & \sigma & (b_{in},\ b'_{out})_i \\
\multicolumn{10}{l}{\text{where } b'_{out} \quad = \quad \textit{enqueue messages } b_{out}} \\
\multicolumn{10}{l}{\qquad\quad \textit{messages} \quad = \quad \langle \forall\, j \in \{1, \ldots, n\} \wedge j \neq i \bullet (i, j, \textit{body}) \rangle}
\end{array}}
$$

As stated previously, all receives are blocking. Rather than committing to a potentially infinite delay, the GUM architecture continually polls the network to determine if a

	specification	description
message	$(i_{source}, j_{destination}, body)$	in addition to its content, a message also records both its sender and receiver.
body	$\mathit{Fish\ age\ originator}$	request work from another processor. The *age* field denotes the number of times the message can be forwarded before aborting the request and returning it to the *originator*.
	$\mathit{Schedule}\ a_{local}\ (\mathit{closure}, \mathit{mask})$	send work to another processor. The *mask* differentiates between addresses and literals contained within the free variables of the *closure*.
	$\mathit{Ack}\ a_{remote}\ a_{local}$	acknowledge receipt of work.
	$\mathit{Fetch}\ a_{remote}\ a_{local}$	request the value of a closure stored on a remote processor.
	$\mathit{Resume}\ a_{remote}\ (\mathit{closure}, \mathit{mask})$	return a value requested by a *Fetch* message.
	Exit	shutdown the system when either evaluation is complete or an error has occured.

Table 9.6: Messages used by GUM

message has arrived. If this is the case, then it is safe to invoke the receive operation:

(RECV)

$$\begin{array}{l} \quad \mathit{code} \qquad\qquad\qquad \mathit{as}\ \ \mathit{rs}\ \ \mathit{us}\ \ h\ \ t_{id}\ \ \mathit{wp}\ \ \sigma\ \ (b_{in},\ b_{out})_i \\ \text{such that } \mathit{probe}\ b_{in}\ \mathit{wild\_card} \\ \Longrightarrow\ \mathit{Receive\ message\ code}\ \ \mathit{as}\ \ \mathit{rs}\ \ \mathit{us}\ \ h\ \ t_{id}\ \ \mathit{wp}\ \ \sigma\ \ (b'_{in},\ b_{out})_i \\ \text{where } (\mathit{message}, b'_{in}) = \mathit{dequeue}\ b_{in} \end{array}$$

In effect, this rule will be triggered as soon as a message arrives, thereby overriding the normal sequence of transitions (this is analogous to a microprocessor interrupt handler – see section 6.2.3). GUM then invokes a specialised message handler, based on the type of the message received (table 9.6 lists the various message types). The handler is passed the *code* continuation to allow it to resume the interrupted task (if appropriate). The GUM message handlers (and generators) are as follows:

category	message	SEND/BCAST rules	RECV rules
load balancing	*Fish*	FISH$_1$, FISH$_2$	SEND_WORK$_1$, FISH$_2$, FISH$_3$
	Schedule	SEND_WORK$_1$	RECV_WORK
	Ack	RECV_WORK	RECV_ACK
remote references	*Fetch*	FETCH$_1$	FETCH$_2$
	Resume	FETCH$_2$, UNBLOCK	FETCH$_3$, FETCH$_4$
termination	*Exit*	FINISH$_1$	FINISH$_2$

The three communication rules are widely used by the rest of the GUM rule set, as

$$\text{(LETPAR)}\quad \boxed{\begin{array}{llllllllll} & \mathit{Eval}\ (\mathtt{letpar}\ v = e_1\ e_2)\ \rho & as & rs & us & h & t_{id} & wp & \sigma & b_i \\ \Longrightarrow & \mathit{Eval}\ e_2\ (\rho \overset{\rightarrow}{\oplus} \{v \mapsto a\}) & as & rs & us & h' & t_{id} & wp' & \sigma & b_i \\ \multicolumn{10}{l}{\text{where } h' \ = \ h[a \mapsto \mathit{create\_closure}\ e_1\ \rho]} \\ \multicolumn{10}{l}{\qquad\ \ wp' \ = \ \mathit{insert}_{spark}\ a\ wp} \end{array}}$$

$$\text{(SCHED}_1\text{)}\quad \boxed{\begin{array}{llllllll} & \mathit{GetThread} & \langle\rangle & \langle\rangle & \langle\rangle & h & t_{id} & (\langle\rangle_{threads}, \mathit{sparks}, f)\quad \sigma \quad b_i \\ \Longrightarrow & \mathit{Enter}\ a & \langle\rangle & rs & \langle\rangle & h & t_{new\_id} & (\langle\rangle_{threads}, \mathit{sparks}', f)\quad \sigma \quad b_i \\ \multicolumn{8}{l}{\text{where } rs \qquad\qquad = \ \langle \mathit{EndThread} \rangle} \\ \multicolumn{8}{l}{\qquad\quad (a, \mathit{sparks}') \ = \ \mathit{dequeue}\ \mathit{sparks}} \\ \multicolumn{8}{l}{\qquad\quad h' \qquad\qquad = \ h[t_{new\_id} \mapsto \mathit{TSO}\ \mathit{init\_tso\_state}]} \end{array}}$$

$$\text{(SCHED}_2\text{)}\quad \boxed{\begin{array}{llllllll} & \mathit{GetThread} & \langle\rangle & \langle\rangle & \langle\rangle & h & t_{id} & (\mathit{threads}, \mathit{sparks}, f)\quad \sigma \quad b_i \\ \multicolumn{8}{l}{\text{such that } \neg \mathit{is\_empty}\ \mathit{threads}} \\ \Longrightarrow & \mathit{code} & as & rs & us & h & t_{next} & (\mathit{threads}', \mathit{sparks}, f)\quad \sigma \quad b_i \\ \multicolumn{8}{l}{\text{where } (t_{next}, \mathit{threads}') \qquad\quad = \ \mathit{dequeue}\ \mathit{threads}} \\ \multicolumn{8}{l}{\qquad\quad (\mathit{TSO}\ (\mathit{code}, as, rs, us)) \ = \ h\ t'_{id}} \end{array}}$$

$$\text{(SCHED}_3\text{)}\quad \boxed{\begin{array}{lllllllll} & \mathit{GetThread} & as & rs & us & h & t_{id} & wp & \sigma\ \ b_i \\ \multicolumn{9}{l}{\text{such that } \mathit{is\_empty}\ wp \text{ and } \mathit{is\_fishing}\ wp} \\ \Longrightarrow & \mathit{GetThread} & as & rs & us & h & t_{id} & wp & \sigma\ \ b_i \end{array}}$$

Figure 9.15: GUM thread-management rules: scheduling

shown by the following table:

action	associated rules
SEND	FISH$_1$, FETCH$_1$, UNBLOCK
RECV	FETCH$_4$, FINISH$_2$
SEND & RECV	FISH$_3$, SEND_WORK$_1$, RECV_WORK, RECV_ACK, FETCH$_2$
BCAST	FINISH$_1$

The rules that both send and receive messages are akin to Culler's active messages [Culler, Goldstein, Schauser and von Eicken, 1992].

Scheduling

As with the previous study, GUM uses the *evaluate-and-die* thread model, and therefore the work-pool definitions are very similar (see table 9.7). Note, however, that each processor has a local pool, as opposed to the centralised structure used by the speculative system. Due to their similarity with the rules described previously, the GUM scheduling rules are presented together in figure 9.15. Note that the SCHED$_3$ busy-wait can only be broken by the the RECV_WORK rule. One alternative to this busy-wait would be to perform a local garbage collection.

Synchronisation

Again, as with scheduling, the synchronisation rules are very similar to those used in the previous case study (see figure 9.16). The main difference occurs with the BH$_3$ rule, which is responsible for updating a shared closure. In addition to releasing any blocked

	specification	description
$code$	$GetThread$	schedule the next thread to be run from the work pool.
wp	$(threads, sparks, fishing)$	the tasks currently available to the system.
$threads$	queue of t_{id}	an unordered collection of threads.
$sparks$	sequence of a	pointers to closures whose values will be required as part the main computation, and which may be evaluated in parallel with the main thread.
$fishing$	$true \mid false$	indicates whether the system has issued a work request to a neighbour and is awaiting a reply.
t_{id}	a	a thread's identifier is the address of its heap-allocated state object, TSO.
$closure$	$BlackHole\ blocked$	used to replace thunks once evaluation begins, ensure there is no duplication of work. Records the ids of any threads and fetches which are awaiting the final value of the closure.
	$TSO\ (code, as, rs, us)$	represents the state of a thread, which comprises its instruction sequence, and the three standard stacks.
$continuation$	$EndThread$	terminate the current thread.
	$Finished$	terminate the entire computation.

Table 9.7: State components of GUM's work pool

(BH$_1$)

$$\boxed{\begin{array}{llllllll}
\quad \textit{Enter}\ a & as & rs & us & h[a \mapsto (vs\ \mathbf{u} \rightarrow e, ws)] & t_{id} & wp\ \ \sigma & b_i \\
\Longrightarrow\ \textit{Eval}\ e\ \rho & \langle\rangle & \langle\rangle & us' & h[a \mapsto \textit{BlackHole}\ bk_{empty}] & t_{id} & wp\ \ \sigma & b_i \\
\text{where}\ \ \rho & = & \multicolumn{6}{l}{\{v_1 \mapsto w_1, \ldots, v_n \mapsto w_n\}} \\
\qquad us' & = & \multicolumn{6}{l}{(a, as, rs) : us} \\
\qquad (v_i, w_i) & = & \multicolumn{6}{l}{(vs\ !\ i, ws\ !\ i)} \\
\qquad bk_{empty} & = & \multicolumn{6}{l}{(\langle\rangle_{threads}, \langle\rangle_{fetches})_{blocked}}
\end{array}}$$

(BH$_2$)

$$\boxed{\begin{array}{llllllll}
\quad \textit{Enter}\ a & as & rs & us & h\,[a \mapsto \textit{BlackHole}\ bk] & t_{id} & wp\ \ \sigma & b_i \\
\Longrightarrow\ \textit{GetThread} & as & rs & us & h'[a \mapsto \textit{BlackHole}\ bk'] & t_{id} & wp\ \ \sigma & b_i \\
\multicolumn{8}{l}{\text{where}\ \ h' \ = \ h'[t_{id} \mapsto TSO\ (\textit{Entera}, as, rs, us)]} \\
\multicolumn{8}{l}{\qquad\quad bk' \ = \ \textit{enqueue}_{thread}\ t_{id}\ bk}
\end{array}}$$

(BH$_3$)

$$\boxed{\begin{array}{llllllll}
\quad \textit{Return}_{\chi}\ c\ ws\ \ \langle\rangle & \langle\rangle & \begin{pmatrix} a_u, \\ as_u, \\ rs_u \end{pmatrix} : us & h & t_{id} & wp & \sigma & b_i \\
\multicolumn{8}{l}{\text{such that}\ h[a_u \mapsto \textit{BlackHole blocked}]} \\
\Longrightarrow\ \textit{unblock} & as_u\ \ rs_u & us & h' & t_{id} & wp & \sigma & b_i \\
\multicolumn{8}{l}{\text{where}\ \textit{unblock} \ = \ \textit{Unblock blocked}\ (\textit{Return}_{\chi}\ c\ ws)} \\
\multicolumn{8}{l}{\qquad h' \ = \ h[a_u \mapsto (vs\ \mathbf{r} \rightarrow c\ vs, ws)]} \\
\multicolumn{8}{l}{\qquad \textit{length}\ vs \ = \ \textit{length}\ ws} \\
\multicolumn{8}{l}{\qquad vs\ \text{is a sequence of arbitrary distinct variables}}
\end{array}}$$

Figure 9.16: GUM thread-management rules: synchronisation

threads, it must also reply to any blocked *Fetch* request. This is handled by the *Unblock* mechanism, which is described in greater detail in the the remote-referencing section.

Load balancing – an overview

As discussed in section 6.3.3, GUM uses a passive load-balancing strategy: when a processor runs out of work, it sends a *Fish* message to one of its neighbours requesting additional work. If the receiver has any spare work then it packages it up and returns it. Figure 9.17 shows the necessary interactions for an unemployed processor to receive work. If the receiver had no spare work, then the *Fish* would be forwarded on, until either a suitable processor is found or the message becomes stale. Stale messages are returned to their originator, as shown in figure 9.18. In summary, upon arrival of a *Fish* message, there are four possible outcomes:

1. there is sufficient local work, with at least one spare spark available, which is therefore packed and returned to the source of the fish message. (rule SEND_WORK)

2. there is no work, but the fish message is not stale, in which case it is forwarded to another processor. (rule FISH$_2$)

3. the out-of-work processor receives its own fish message, (and assuming no local work has become unblocked) then the fish is regenerated after a suitable timeout period. (rule FISH$_3$)

4. there is no work and the fish has become stale, i.e. has visited too many processors, in which case a stale-fish message is returned to the source of the fish message. (rule

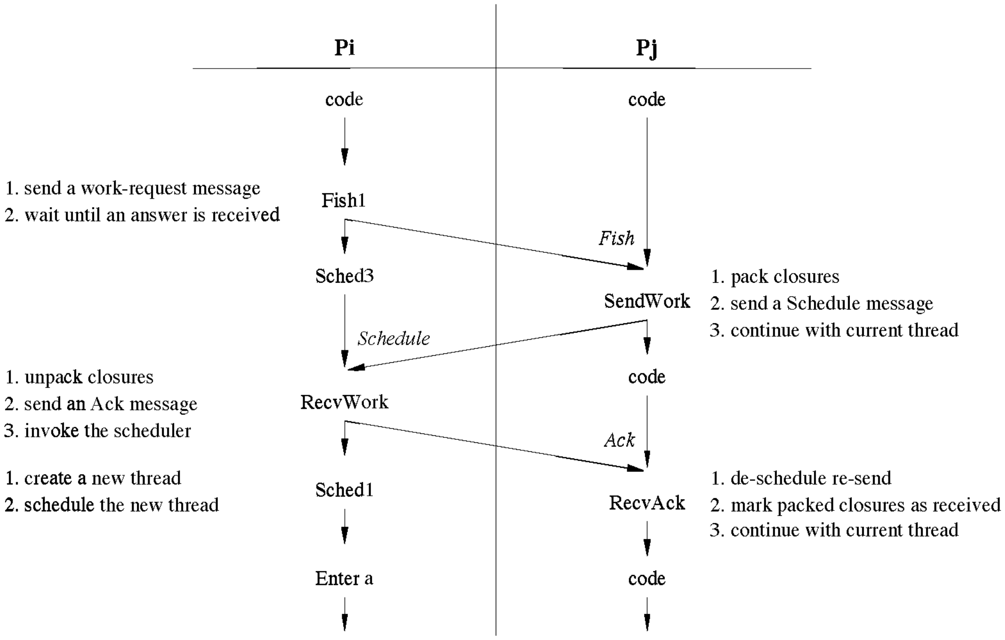

Pi
Pj
code
code
1. send a work-request message
2. wait until an answer is received
Fish1
Fish
Sched3
1. pack closures
2. send a Schedule message
3. continue with current thread
SendWork
Schedule
1. unpack closures
2. send an Ack message
3. invoke the scheduler
code
RecvWork
Ack
1. create a new thread
2. schedule the new thread
Sched1
1. de-schedule re-send
2. mark packed closures as received
3. continue with current thread
RecvAck
Enter a
code

Figure 9.17: GUM load balancing: a successful work-request cycle

FISH$_2$)

The FISH$_1$, SEND_WORK, RECV_WORK, and RECV_ACK rules form the backbone of the load-balancing mechanism and are discussed in the subsequent sections. The remaining rules, FISH$_2$ and FISH$_3$, are shown in figure 9.23.

Load balancing – asking for work

The load-balancing mechanism is activated whenever a processor becomes idle. This situation typically arises because all local threads have either been fully evaluated or are currently blocked awaiting the arrival of a remote reference. Also, at the start of the computation, only the main processor will have any work (see the INIT rule). The initial phases of the evaluation will therefore entail a large number of *Fish* messages. The rule for generating *Fish* messages is shown in figure 9.19. In the real GUM implementation, the *Fish* message is sent to a random processor, rather than to its right-hand side neighbour (the HDG machine employs a neighbour-first strategy [Kingdon, Lester and Burn, 1991, section 3.2, page 293].)

Load balancing – receiving work

Having sent the *Fish* message, the processor will remain idle until either a *Schedule*, *Resume*, or *Exit* message is received. The RECV_WORK rule is the handler for *Schedule* messages, and is shown in figure 9.20. Upon arrival of a *Schedule* message, the sequence of events is as follows: the message is unpacked (see figure 9.21) and the closure contained therein is stored at heap address a_{local}; next, an acknowledgement is sent to the donor processor; and, finally, the closure's standard-entry method is invoked. When the evaluation returns, the *EndThread* continuation will place the system into the *GetThread* mode, thereby re-starting the work-request cycle.

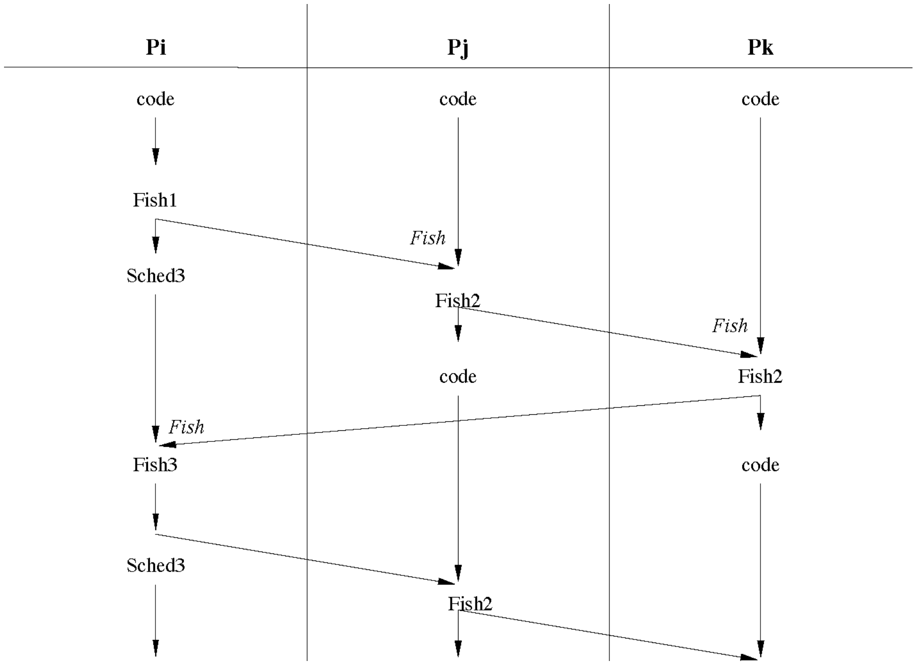


Figure 9.18: GUM load balancing: an unsuccessful work-request cycle

$$
\begin{array}{llll}
 & \multicolumn{3}{l}{GetThread \qquad\qquad\qquad as\ \ rs\ \ us\ \ h\ \ t_{id}\ \ wp\ \ \sigma\ \ b_i} \\
\\
\multicolumn{4}{l}{\text{such that } is\_empty\ wp \text{ and } \neg is\_fishing\ wp} \\
\\
\Longrightarrow & \multicolumn{3}{l}{Send\ request\ GetThread\ \ as\ \ rs\ \ us\ \ h\ \ t_{id}\ \ wp'\ \ \sigma\ \ b_i} \\
\text{where} & request & = & (i, neighbour, Fish) \\
 & neighbour & = & 1 + (i \bmod n) \\
 & wp' & = & set\_fishing\ true\ wp
\end{array}
\tag{$\textsc{fish}_1$}
$$

Figure 9.19: Initiating a GUM work-request cycle

$$
\begin{array}{llll}
 & \multicolumn{3}{l}{Receive\ message\ code\ \ as\ \ rs\ \ us\ \ h\ \ t_{id}\ \ wp\ \ \sigma\ \ b_i} \\
\\
\multicolumn{4}{l}{\text{such that } message \equiv (j, i, Schedule\ a_{remote}\ (closure, mask))} \\
\\
\Longrightarrow & \multicolumn{3}{l}{Send\ ack\ code \qquad\quad as\ \ rs\ \ us\ \ h'\ \ t_{id}\ \ wp'\ \ \sigma\ \ b_i} \\
\text{where} & ack & = & (i, j, Ack\ a_{local}\ a_{remote}) \\
 & wp & \equiv & (threads,\ \ sparks,\ \ fishing) \\
 & wp' & = & (threads,\ \ sparks',\ \ false) \\
 & sparks' & = & insert_{spark}\ a_{local}\ sparks \\
 & (a_{local}, h') & = & unpack\ j\ closure\ mask\ h
\end{array}
\tag{$\textsc{recv\_work}$}
$$

Figure 9.20: Receiving work from a remote processor

$$
\boxed{
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{pack}\ a\ j\ h[a \mapsto (\mathit{vs}\ \pi\ \mathit{xs} \to \mathit{exp}, \mathit{ws})] = (\mathit{data}, h')} \\
\text{where } h' & = & \left\{ \begin{array}{ll} h[a \mapsto \mathit{Exported}\ j\ \mathit{closure}\ \mathit{bk}_{\mathit{empty}}], & \text{if } (\pi == \mathrm{u}) \\ h, & \text{otherwise} \end{array} \right. \\
\quad \mathit{data} & = & (a, \mathit{vs}\ \pi\ \mathit{xs} \to \mathit{exp}, \mathit{mask}, \mathit{ws}) \\
\quad \mathit{mask} & = & \mathit{mask}_1 \cdots \mathit{mask}_n \\
\quad \mathit{mask}_i & = & \left\{ \begin{array}{ll} 0, & \text{if } \vdash (\mathit{vs}\,!\,i) : \nu \\ 1, & \text{otherwise} \end{array} \right. \\
\quad n & = & \mathit{length}\ \mathit{vs} \\
\quad \mathit{bk}_{\mathit{empty}} & = & (\langle\rangle_{\mathit{threads}}, \langle\rangle_{\mathit{fetches}})_{\mathit{blocked}}
\end{array}
}
$$

$$
\boxed{
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{unpack}\ j\ \mathit{closure}\ \mathit{mask}\ h_0 = (a_{\mathit{local}}, h'_n)} \\
\multicolumn{3}{l}{\text{where}} \\
h'_n & = & h_n[a_{\mathit{local}} \mapsto (\mathit{lambda\_form}, w'_1 \cdots w'_n)] \\
(h_i, w'_i) & = & \left\{ \begin{array}{ll} (w_i, h_{i-1}), & \text{if } \mathit{mask}_i == 0 \\ (a_i, h_{i-1}[a_i \mapsto \mathit{FetchMe}\ j\ w_i\ \mathit{bk}_{\mathit{empty}}]), & \text{otherwise} \end{array} \right. \\
\mathit{closure} & \equiv & (\mathit{lambda\_form}, w_1 \cdots w_n) \\
\mathit{mask} & \equiv & \mathit{mask}_1 \cdots \mathit{mask}_n \\
\mathit{bk}_{\mathit{empty}} & = & (\langle\rangle_{\mathit{threads}}, \langle\rangle_{\mathit{fetches}})_{\mathit{blocked}}
\end{array}
}
$$

Figure 9.21: Incremental fetching: packing and unpacking *Schedule* messages

The unpacking process replaces all heap references contained within the new closure with local pointers to *FetchMe* closures. In addition to the main closure, GUM also packs some of the "nearby" reachable graph into each *Schedule* message [Trinder et al., 1996, section 2.4]. This improves the locality of reference, and reduces the impact of the fixed overhead of sending the message.

Load balancing – answering a request for work

When a processor receives a *Fish* request, and has surplus work, it returns a *Schedule* message containing a thunk for evaluation on the unemployed processor:

$$
(\textsc{send\_work}_1) \quad
\boxed{
\begin{array}{llll}
 & \multicolumn{2}{l}{\mathit{Receive}\ \mathit{message}\ \mathit{code}} & \mathit{as}\ \ \mathit{rs}\ \ \mathit{us}\ \ h\ \ t_{\mathit{id}}\ \ \mathit{wp}\ \ \sigma\ \ b_i \\
 & & & \\
\multicolumn{4}{l}{\text{such that}\ \ \mathit{message} \equiv (j, i, \mathit{Fish}\ \mathit{age}\ \mathit{origin}),} \\
\multicolumn{4}{l}{\qquad\quad\ \text{and}\ \ \neg \mathit{is\_empty}_{\mathit{sparks}}\ \mathit{wp}} \\
 & & & \\
\Longrightarrow & \mathit{code}' & & \mathit{as}\ \ \mathit{rs}\ \ \mathit{us}\ \ h'\ \ t_{\mathit{id}}\ \ \mathit{wp}'\ \ \sigma\ \ b_i \\
\text{where} & \mathit{code}' & = & \left\{ \begin{array}{ll} \mathit{send\_work}, & \text{if } (\pi == \mathrm{u}) \\ \mathit{retry}, & \text{otherwise} \end{array} \right. \\
 & \mathit{send\_work} & = & \mathit{Send}\ \mathit{work}\ \mathit{code} \\
 & \mathit{work} & = & (i, j, \mathit{Schedule}\ a\ \mathit{data}) \\
 & (\mathit{data}, h') & = & \mathit{pack}\ \mathit{spark}\ j\ h \\
 & (\mathit{vs}\ \pi\ \mathit{xs} \to e, \mathit{ws}) & = & h\ \mathit{spark} \\
 & (\mathit{spark}, \mathit{wp}') & = & \mathit{dequeue}_{\mathit{spark}}\ \mathit{wp} \\
 & \mathit{retry} & = & \mathit{Receive}\ \mathit{message}\ \mathit{code}
\end{array}
}
$$

$$
\textsc{(recv\_ack)}\quad
\boxed{
\begin{array}{lll}
 & \multicolumn{2}{r}{\mathit{Receive}\ \mathit{message}\ \mathit{code}\quad \mathit{as}\quad \mathit{rs}\quad \mathit{us}\quad h\quad t_{id}\quad \mathit{wp}\quad \sigma\quad b_i} \\[1em]
\text{such that} & \multicolumn{2}{l}{\mathit{message} \equiv (j, i, \mathit{Ack}\ a_{remote}\ a_{local})} \\
\text{and} & \multicolumn{2}{l}{h[a_{local} \mapsto \mathit{Exported}\ j\ \mathit{thunk}\ \mathit{blocked}]} \\[1em]
\Longrightarrow & \multicolumn{2}{r}{\mathit{code}'\quad \mathit{as}\quad \mathit{rs}\quad \mathit{us}\quad h'\quad t_{id}\quad \mathit{wp}\quad \sigma\quad b_i} \\
\text{where } \mathit{code}' & = & \begin{cases} \mathit{Send}\ \mathit{fetch}\ \mathit{code}, & \text{if } \neg \mathit{is\_empty}\ \mathit{blocked} \\ \mathit{code}, & \text{otherwise} \end{cases} \\
h' & = & h[a_{local} \mapsto \mathit{FetchMe}\ j\ a_{remote}\ \mathit{blocked}] \\
\mathit{fetch} & = & (i, j, \mathit{Fetch}\ a_{remote}\ a_{local})
\end{array}
}
$$

Figure 9.22: Receiving acknowledgments for the safe-arrival of *Schedule* messages

The *pack* routine converts the local thunk into a form suitable for transmission (see figure 9.21 for details). Furthermore, to avoid duplicating work, *pack* will convert thunks into *Exported* closures. This is a temporary measure until the destination processor acknowledges receipt of the *Schedule* message. If the message is lost, or the recipient cannot unpack the message for any reason, the original closure can be recovered.[1] Assuming that nothing does go wrong, the acknowledgement is handled by rule RECV_ACK, as shown in figure 9.22.

Representing and requesting remote references

Values stored on remote processors are represented by *FetchMe* closures [Trinder et al., 1996, figure 2, section 2.3]. The related STG definitions are show in table 9.8. Upon entry to a remote-reference, the processor will send the owner a *Fetch* request, and suspend the current thread pending arrival of the value. Figure 9.25 shows these interactions, which are initiated by the FETCH$_1$ rule, shown in figure 9.24.

How are these remote references created in the first place? There are three main sources: the partitioning of the top-level bindings as specified by the INIT rule; closure migration as a result of a *Schedule* message; and the directed allocation of dynamic values by, for example, para-functional Haskell's **on** expression [Mirani and Hudak, 1995, section 4].

Replying to a *Fetch* request

The reply to a request for a local value is very similar to that used to send work to an unemployed processor (see the SEND_WORK rule). Essentially, both messages contain a packaged closure, although *Schedule* messages will contain thunks, while *Resume* messages will typically contain closures in head-normal form (i.e. they will be re-entrant, and

[1] Mattson's *grey hole* [Mattson Jr., 1993a, figure 4.3, page 79] is another example of a reversible update.

$$(\textsc{fish}_2)\quad \boxed{\begin{array}{l}
\qquad\quad \mathit{Receive\ message\ code}\ \ as\ \ rs\ \ us\ \ h\ \ t_{id}\ \ wp\ \ \sigma\ \ b_i \\[1em]
\text{such that}\ \ message \equiv (j, i, Fish\ age\ origin) \\
\qquad\quad \text{and}\ \ is\_empty_{sparks}\ wp \\[1em]
\Longrightarrow\quad \mathit{Send\ fish'\ code}\qquad\ \ as\ \ rs\ \ us\ \ h\ \ t_{id}\ \ wp\ \ \sigma\ \ b_i \\
\text{where}\quad fish' \ = \left\{\begin{array}{ll} (i, j, Fish\ (age+1)\ origin), & \text{if } age < age_{stale} \\ (i, origin, Fish\ age\ origin), & \text{otherwise} \end{array}\right. \\
\qquad\qquad j \qquad = \ 1 + (i \bmod n)
\end{array}}$$

$$(\textsc{fish}_3)\quad \boxed{\begin{array}{l}
\qquad\quad \mathit{Receive\ message\ code}\ \ as\ \ rs\ \ us\ \ h\ \ t_{id}\ \ wp\ \ \sigma\ \ b_i \\[1em]
\text{such that}\ \ message \equiv (j, i, Fish\ age\ origin) \\
\qquad\quad \text{and}\ \ (i == origin) \\[1em]
\Longrightarrow\quad code\qquad\qquad\qquad\quad as\ \ rs\ \ us\ \ h\ \ t_{id}\ \ wp'\ \ \sigma\ \ b_i \\
\text{where}\ \ wp' = set\_fishing\ false\ wp
\end{array}}$$

Figure 9.23: GUM load balancing: the other FISH rules

	specification	description
code	*Unblock blocked code*	used to re-activate blocked threads and *Fetch* messages upon arrival of a remote-closure's value (also used when updating black holes).
closure	*FetchMe i* a_{remote} *blocked*	a reference to a value stored on a remote processor
	Exported i closure blocked	work that has been exported to processor j in response to a *Fish* message, but receipt of which has not yet been acknowledged
blocked	$(threads, fetches)$	used to store details of any threads and *FetchMe* messages which have become blocked on a closure. When the closure is updated, the threads will be re-awakened, and replies made to the *FetchMe* messages.
threads	queue of t_{id}	an unordered collection of threads.
fetches	queue of (i, a_{remote})	an unordered collection of *FetchMe* requests

Table 9.8: Representing remote references with GUM

$$
(\textsc{Fetch}_1)\quad
\boxed{\begin{array}{l}
\quad Enter\ a_{local}\quad as\quad rs\quad us\quad h\quad t_{id}\quad wp\quad \sigma\quad b_i \\[1ex]
\text{such that } h[a_{local} \mapsto FetchMe\ j\ a_{remote}\ blocked] \\[1ex]
\Longrightarrow\quad code\qquad as\quad rs\quad us\quad h'\quad t_{id}\quad wp'\quad \sigma\quad b_i \\
\text{where} \\
\begin{array}{lcl}
h' & = & h\left[\begin{array}{lcl} a_{local} & \mapsto & FetchMe\ j\ a_{remote}\ blocked' \\ t_{id} & \mapsto & TSO\ (Enter\ a_{local}, as, rs, us) \end{array}\right] \\
blocked' & = & enqueue_{thread}\ t_{id}\ blocked \\
code & = & \left\{\begin{array}{ll} Send\ fetch\ GetThread, & \text{if } is\_empty\ blocked \\ GetThread, & \text{otherwise} \end{array}\right. \\
fetch & = & (i, j, Fetch\ a_{remote}\ a_{local})
\end{array}
\end{array}}
$$

Figure 9.24: Handling remote references in a distributed-memory architecture

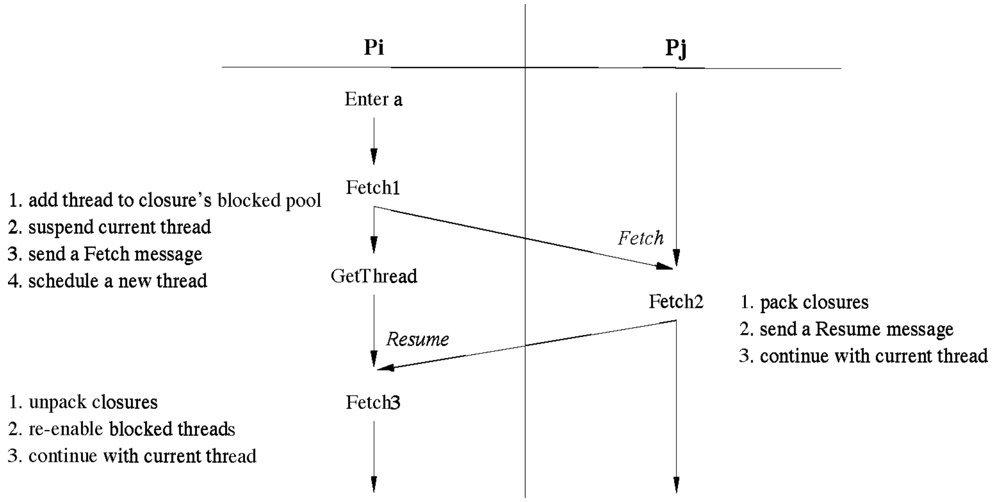

Pi
Pj
Enter a
Fetch1
1. add thread to closure's blocked pool
2. suspend current thread
3. send a Fetch message
4. schedule a new thread
Fetch
GetThread
Fetch2
1. pack closures
2. send a Resume message
3. continue with current thread
Resume
Fetch3
1. unpack closures
2. re-enable blocked threads
3. continue with current thread

Figure 9.25: Accessing remote references with GUM

so their update flags will be $\mathtt{r}$). The following rule handles the packing and reply:

$$\text{(FETCH}_2\text{)} \quad \boxed{\begin{array}{llll}
 & \multicolumn{3}{l}{\mathit{Receive}\ \mathit{message}\ \mathit{code}\quad \mathit{as}\ \mathit{rs}\ \mathit{us}\ h\ t_{id}\ \mathit{wp}\ \sigma\ b_i} \\
\multicolumn{4}{l}{\text{such that } \mathit{message} \equiv (j, i, \mathit{Fetch}\ a_{local}\ a_{remote}) \text{ and } (\pi == \mathtt{r})} \\
\Longrightarrow & \mathit{Send}\ \mathit{resume}\ \mathit{code} & & \mathit{as}\ \mathit{rs}\ \mathit{us}\ h'\ t_{id}\ \mathit{wp}\ \sigma\ b_i \\
\text{where} & \mathit{resume} & = & (i, j, \mathit{Resume}\ a_{remote}\ \mathit{data}) \\
 & (\mathit{data}, h') & = & \mathit{pack}\ a_{local}\ j\ h \\
 & (\mathit{vs}\ \pi\ \mathit{xs} \rightarrow e, \mathit{ws}) & = & h\ a_{local}
\end{array}}$$

Remote references – receiving remote values

Upon arrival of the *Resume* message, the remote-value is unpacked, and the *FetchMe* closure updated with an indirection to the new closure:

$$\text{(FETCH}_3\text{)} \quad \boxed{\begin{array}{llll}
 & \multicolumn{3}{l}{\mathit{Receive}\ \mathit{message}\ \mathit{code}\quad \mathit{as}\ \mathit{rs}\ \mathit{us}\ h\ t_{id}\ \mathit{wp}\ \sigma\ b_i} \\
\multicolumn{4}{l}{\text{such that}\ \ \mathit{message} \equiv (j, i, \mathit{Resume}\ a_{local}\ (\mathit{closure}, \mathit{mask}))} \\
\multicolumn{4}{l}{\qquad\quad \text{and}\ \ h[a_{local} \mapsto \mathit{FetchMe}\ j\ a_{remote}\ \mathit{blocked}]} \\
\Longrightarrow & \multicolumn{3}{l}{\mathit{Unblock}\ \mathit{blocked}\ \mathit{code}\quad \mathit{as}\ \mathit{rs}\ \mathit{us}\ h''\ t_{id}\ \mathit{wp}\ \sigma\ b_i} \\
\text{where} & h'' & = & h'[a_{local} \mapsto \mathit{Ind}\ a'] \\
 & (a', h') & = & \mathit{unpack}\ j\ \mathit{closure}\ \mathit{mask}\ h
\end{array}}$$

The *Unblock* phase is responsible for awakening any threads that were waiting for the remote value (there will be at least one, otherwise the *Fetch* would never have been sent). In addition, it also replies to any blocked fetches (the following section detail how this can happen):

$$\text{(UNBLOCK)} \quad \boxed{\begin{array}{llll}
 & \mathit{Unblock}\ \mathit{blocked}\ \mathit{code}_0 & & \mathit{as}\ \mathit{rs}\ \mathit{us}\ h_0\ t_{id}\ \mathit{wp}\ \sigma\ b_i \\
\Longrightarrow & \qquad\qquad\qquad \mathit{code}_n & & \mathit{as}\ \mathit{rs}\ \mathit{us}\ h_n\ t_{id}\ \mathit{wp}'\ \sigma\ b_i \\
\text{where} & \mathit{wp}' & = & \mathit{insert}_{\mathit{threads}}\ \mathit{threads}\ \mathit{wp} \\
 & \mathit{code}_k & = & \mathit{Send}\ \mathit{resume}_k\ \mathit{code}_{k-1} \\
 & \mathit{resume}_k & = & (i, \mathit{source}_k, \mathit{Resume}\ a_k\ \mathit{data}_k) \\
 & (\mathit{data}_k, h_k) & = & \mathit{pack}\ a_u\ \mathit{source}_k\ h_{k-1} \\
 & (\mathit{source}_k, a_k) & = & \mathit{fetches}\ !\ k \\
 & n & = & \mathit{length}\ \mathit{fetches} \\
 & (\mathit{threads}, \mathit{fetches}) & = & \mathit{blocked}
\end{array}}$$

The rule for updating shared thunks, BH$_3$, uses the *unblock* rule to re-awaken the threads and fetches which have been waiting for the local evaluation to complete.

Remote references – requesting black-holed values

Having described the basic mechanism for dealing with remote references, one complication remains. It is possible that a processor is asked for a value which is still being evaluated, i.e. it has been black holed. In this case, the *Fetch* message is simply added to the black

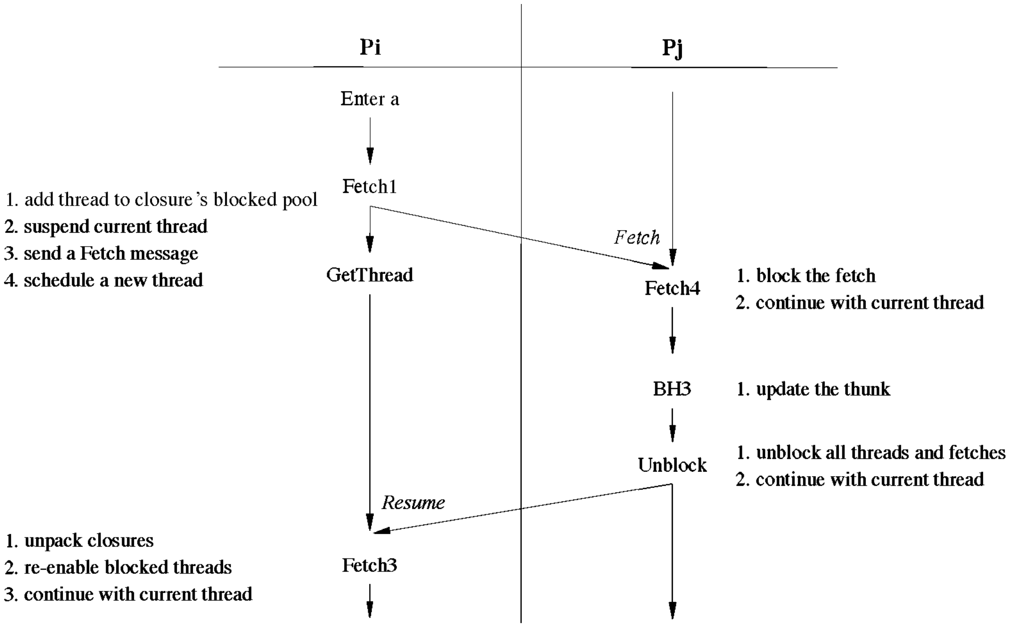

Pi
Pj
Enter a
Fetch1
1. add thread to closure's blocked pool
2. suspend current thread
3. send a Fetch message
4. schedule a new thread
GetThread
Fetch
Fetch4
1. block the fetch
2. continue with current thread
BH3
1. update the thunk
Unblock
1. unblock all threads and fetches
2. continue with current thread
Resume
1. unpack closures
2. re-enable blocked threads
3. continue with current thread
Fetch3

Figure 9.26: Black holes and *Fetch* messages

holes blocking pool:

$$\text{(FETCH4)} \quad \begin{array}{|l|} \hline
\qquad \mathit{Receive\ message\ code} \quad as \quad rs \quad us \quad h \quad t_{id} \quad wp \quad \sigma \quad b_i \\
\\
\text{such that} \quad message \equiv (j, i, \mathit{Fetch}\ a_{local}\ a_{remote}) \\
\qquad\quad \text{and} \quad h[a_{local} \mapsto \mathit{BlackHole\ blocked}] \\
\\
\Longrightarrow \quad code \qquad\qquad\qquad as \quad rs \quad us \quad h' \quad t_{id} \quad wp \quad \sigma \quad b_i \\
\text{where} \quad \begin{array}{lcl} h' & = & h[a_{local} \mapsto \mathit{BlackHole\ blocked'}] \\ blocked' & = & insert_{fetches}\ (j, a_{remote})\ blocked \end{array} \\ \hline
\end{array}$$

When the closure is finally updated, via the BH3 rule, the UNBLOCK rule will ensure the suspended *Fetch* messages are replied to. Figure 9.26 provides an example of this sort of interaction.

Initialisation

As discussed in section 6.3.4, GUM replicates all top-level closures on all processors. While this is expensive in terms of memory, it does improve locality, thereby avoiding processors becoming inundated with requests for "popular" global values. By copying constant applicative forms (CAFs [Peyton Jones, 1987, section 13.2, page 224])), there is a risk of duplicating work. However, if this should become a problem, it is straightforward to re-write an STG′ program such that the CAF becomes a local shared value. GUM's INIT rule is shown in figure 9.27.

$$(\textsc{init})\quad \boxed{\begin{array}{lcl}
G & = & (P_1, \ldots, P_n)\ (b_1, \ldots, b_n) \\
\text{where} & & \\
P_i & = & (\mathit{GetThread}, \langle\rangle, \langle\rangle, \langle\rangle, t_{none}, wp_i, h_i, \sigma) \\
b_i & = & (\langle\rangle_{in}, \langle\rangle_{out})_i \\
wp_i & = & (\langle\rangle, sparks_i, false) \\
sparks_i & = & \begin{cases} \langle a_{main} \rangle, & \text{if } i == 1 \\ \langle\rangle, & \text{otherwise} \end{cases} \\
h_i & = & \begin{cases} h, & \text{if } i == 1 \\ h[a_{main} \mapsto \mathit{FetchMe}\ 1\ a_{main}], & \text{otherwise} \end{cases} \\
h & = & \left[\begin{array}{lll} a_1 & \mapsto & (vs_1\ \pi_1\ vs_1 \to exp_1, \sigma\ vs_1) \\ \ldots, & & \\ a_n & \mapsto & (vs_n\ \pi_n\ vs_n \to exp_n, \sigma\ vs_n) \end{array}\right] \\
\sigma & = & \left\{\begin{array}{lll} g_1 & \mapsto & a_1, \\ \ldots, & & \\ g_n & \mapsto & a_n \end{array}\right\}
\end{array}}$$

Figure 9.27: GUM initialisation

Termination

As with the previous case study, the computation is finished whenever the main thread terminates:

$$(\textsc{finish}_1)\quad \boxed{\begin{array}{llllllllll}
 & \mathit{Return}_\chi\ c\ ws & \langle\rangle & \langle \mathit{Finish} \rangle & \langle\rangle & h & t_{main} & wp & \sigma & b_i \\
\Longrightarrow & \mathit{Broadcast\ Exit\ Stop} & \langle\rangle & \langle\rangle & \langle\rangle & h' & t_{main} & wp & \sigma & b_i \\
\multicolumn{10}{l}{\text{where } h' = h[a_{status} \mapsto \mathit{Stopped}]}
\end{array}}$$

However, rather than relying on a global variable to indicate the end of the evaluation, an *Exit* message is broadcast to all other processors:

$$(\textsc{finish}_2)\quad \boxed{\begin{array}{llllllllll}
 & \mathit{Receive\ Exit\ code} & as & rs & us & h & t_{id} & wp & \sigma & b_i \\
\Longrightarrow & \mathit{Stop} & as & rs & us & h & t_{id} & wp & \sigma & b_i
\end{array}}$$

Distributed garbage collection

The remote-reference mechanisms described in the previous sections completely ignored the implications of global garbage collection. While it would be possible to develop a distributed collector that could handle this situation, it is likely that it would be horribly inefficient. The real GUM implementation provides better support for its global collector by maintaining three tables [Trinder, Hammond, Partridge, Peyton Jones and others, 1996, sections 2.3.1 and 2.3.2]:

GIT the global indirection table identifies all local closures that are globally visible.

GA↦LA this maps a remote reference to a local closure.

LA↦GA this maps local address to their global addresses.

This information allows the collector to identify all global addresses and to efficiently determine whether any of them can be reclaimed. While it would be possible to extend the GUM model to record this information, it is beyond the scope of this thesis.

register	1–22	23	24	25	26	27	28	29	30	31
use	general purpose	Ret	Np	StkA	StkA Base	StkB	StkB Base	Tp	HLimit	Hp

Ret	stores the address of the return handler for the current evaluation (which may be a generic update-handler, when evaluating a polymorphic thunk).
Np	points to the closure which is currently being evaluated, and is used to access an expression's free variables.
StkA	points to the next available slot on the A stack. This is used in conjunction with StkB to detect stack overflow.
StkABase	points to the lower limit of the A stack, and is used to detect stack underflow.
StkB	points to the next available slot on the B stack.
StkBBase	points to the upper limit of the B stack, and is used to detect stack underflow.
Tp	points to the current thread's *TSO* closure.
HLimit	identifies the maximum extent of the local heap, and is used in conjunction with Hp to determine if the garbage collector should be invoked.
Hp	points to the next word of available memory in the local heap. Allocation simply involves incrementing the pointer and the using the space reserved (plus the necessary heap-overflow check).

Table 9.9: The register map for compiling GUM expressions

9.3.3 Compilation rules

Developing the GUM compilation rules was straightforward as it primarily involved minor modifications to the compilation rules and run-time support developed as part of the previous case study. The only significant changes included the addition of a number of extra entry points in the info tables (to support packing and fetching), and the integration of the message-passing routines. These are discussed in the following sections, and use the register map shown in table 9.9.

Sending and receiving messages

The API for the send, receive, and poll primitives used by the architecture simulator are shown in table 7.3. These deal with blocks of words, onto which GUM imposes the following structure:

word	0	1	2	3+
content	source	destination	message tag	message-specific content

This format is slightly inefficient in that the sender/receiver pair occupies two words, when it could be packed into one or two bytes (depending upon the total number of processors). However, this change would increase the complexity of the message-handling routines for only a small return in space saved. Note that each message is tagged with its type, allowing the receiver to efficiently dispatch the message to the correct handler. Figure 9.28 shows

RISC code

```
Lmessage_handlers:

        dw Lrecv_Fish;                       // the Fish handler
        dw Lrecv_Schedule;                   // the Schedule handler
        dw Lrecv_Ack;                        // the Ack handler
        dw Lrecv_Fetch;                      // the Fetch handler
        dw Lrecv_Resume;                     // the Resume handler
        dw Lrecv_Exit;                       // the Exit handler

Lrecv_buffer:

        dw   5000;                           // length of buffer
        data 5000;                           // temporary storage for messages

LGUM_recv:

        branch_link Lpoll, R21;              // poll the network
        branch_x>=0 R22, LGUM_recv_message;  // branch if there is a message
        jump R17;                            // return to the caller

LGUM_recv_message:

        load_high Lrecv_buffer(R0), R22;     // load the address of the fixed
        load_address +4(R22), R22;           // buffer
        load -4(R22), R21;                   // load the buffer length
        branch_link Lrecv, R20;              // call the system recv primitive
        load_high Lmessage_handlers(R0), R18; // load the message-hanlder
        load_address +0(R18), R18;
        load +8(R22), R19;                   // extract the message tag
        add R8, R19, R19;                    // calculate the table index
        load +0(R19), R19;                   // extract the handler's address
        jump R19;                            // ...and tail-call it
```

Figure 9.28: The RISC implementation of the STG RECV rule

the RISC code which implements the operational RECV rule, and the corresponding tags are shown below:

message	*Fish*	*Schedule*	*Ack*	*Fetch*	*Resume*	*Exit*
tag	0	4	8	12	16	20

Packing and fetching

The behaviour of the fetching and packing mechanisms is closure-specific. For example, the STG FETCH_2 and FETCH_4 rules specify how to fetch standard closures and black holes respectively. Rather than tagging each closure, the standard approach to handling closure-specific code is to add a new entry method (see section 6.4.3). To this end, figure 9.29 shows the info table for a standard closure. Note that the pack and fetch methods are bundled together with the garbage collection operations (see section 6.3.3). While each STG′ binding will generate a unique info table (and associated entry code), they can share a small collection of GC and communication methods[2]. Figure 9.30 shows the pack operation for handling re-entrant closures.

[2] The literal- and boxed-counts stored in the main info table make this sharing possible. Given these two pieces of information, it is possible to infer the exact layout of the closure.

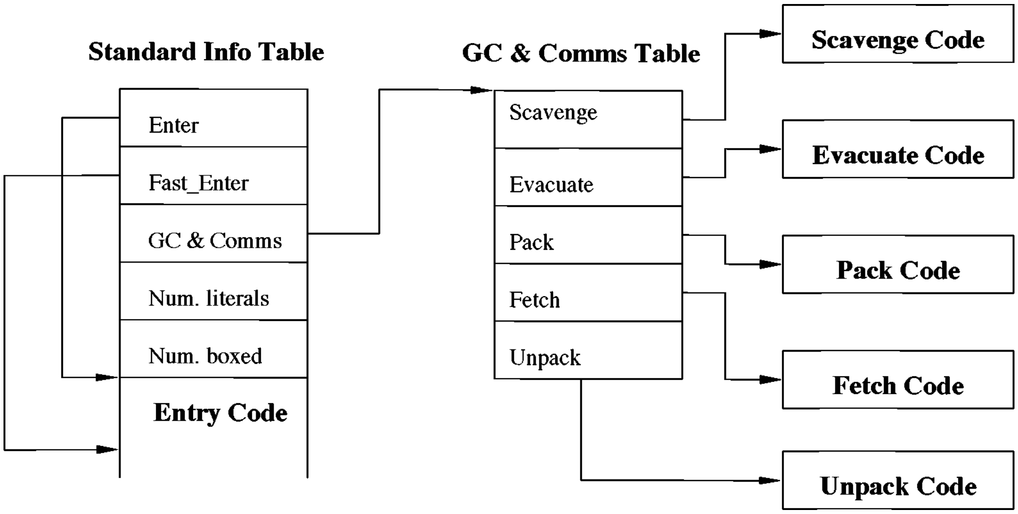


Figure 9.29: Layout of the GUM info tables for a standard closure

RISC code

```
  //
  // pack API:          In                        Out
  //           R19 return continuation    R1, R2, R3, & R19 corrupted
  //           R21 buffer                 R21 pointer to end of data
  //           R22 buffer length          R22 space remaining

Lpack_reentrant_closure:

        load (RNp), R1;                     // load the info table
        store R1, +4(R21);                  // and store it in the buffer
        load +12(R1), R2;                   // load the number of literals
        load +16(R1), R1;                   // and the number of boxed values
        add R1, R2, R1;                     // find the total size
        store R1, (R21);                    // store it in the buffer
        add R21, +8, R21;                   // move the index forward
        subtract R22, +8, R22;              // decrement the space remaining
        add RNp, +4, R2;                    // point to the free vars

Lpack_re_loop:

        branch_x>0 R1, Lfinish_re_pack;     // exit if no more values
        load (R2), R3;                      // obtain the next value
        store R2, (R21);                    // pack it
        add R2, +4, R2;                     // move the index forward
        add R21, +4, R21;                   // and the buffer index
        subtract R1, +1, R1;                // decrement the counter
        subtract R22, +4, R22;              // and the space remaining
        branch Lpack_re_loop;               // and repeat.

Lfinish_re_pack:

        jump R19;
```

Figure 9.30: The RISC implementation of the *pack* method for re-entrant closures

9.3.4 Other message-passing systems

Although a quarter of the 92 ν-STG machine rules involve some form of message passing, the abstract state [Hwang and Rushall, 1992, section 3] does not include a communications component. Instead, sending is specified via a (side-effecting) auxiliary function, `sendMessage`, and a dedicated mode handles the implicit reception of each kind of message. The capabilities of the resulting system are, however, similar to those outlined in this section.

The Alfalfa system [Goldberg and Hudak, 1987, section 4.6, pages 106–107] uses three types of messages: *system messages*, containing load information and other administrative details used by the scheduling system; *reducer messages*, similar to GUM's *Schedule*, except they are sent pro-actively; and *storage messages*, which are the main component of the reference-counting garbage collector, and are generated whenever a reference to a closure is either replicated or deleted (GUM uses a *Free* message to implement a similar system.)

Concurrent Clean [Nöcker, Smetsers, Plasmeijer and van Eekelen, 1991, section 5.0, pages 215–216] uses *channel* nodes to handle remote references. These are almost identical to GUM's *FetchMe* closures.

9.3.5 Performance

As with the previous case study, the `fib` and `queens` benchmarks are used to evaluate the performance of the GUM model. Both the STG and RISC animations allows the costs of sending and receiving messages to be modified (using the *LogP* communication model [Culler et al., 1993] – see section 2.2.1). As such, the performance evaluations consider the effect of transmission time on the models performance. To simplify the presentation of the results, the following categories are used to describe the various costs:

cost	description
0–50	very low
50–150	low
150–500	medium
500+	high

As the STG and RISC animations produce very similar results, only those for the STG simulation are presented here.

The `fib` benchmark

The relative speedup curve for the optimised `fib` 15 STG benchmark is shown in figure 9.32 (with medium communication costs). The system achieves a maximum speedup of just over two, which compares poorly with the speculative GMSV model (this runs approximately eighteen times faster on twenty processors). However, the STG animation does exhibit the performance trail off associated with adding surplus processors. With the speculative system, only the RISC animation was accurate enough to reproduce this phenomena.

Upon further analysis of the animation traces, it quickly became obvious that the work-distribution mechanism was stripping processors of their `fib_n_less1` and `fib_n_less2` sparks, leaving them with just the addition operation. This couldn't proceed until the two sparks were evaluated, so many processors spent significant portions of their time idling. The solution was to re-write the `fib` benchmark, resulting in the `fib2` program shown in

STG′ code

```
value = [] \r [] -> Int [15#];
main  = [] \u [] -> fib value ;

fib = [] \r [n] -> case  n  of { Int n' -> fib.wrk n'; };
fib.wrk = [] \r [n'] -> case leInt# [n', 1#] of
 { True  -> Int [1#];
   False -> let { fib_n_less1 = [] \u [] -> let# n'_less_1 = minusInt# [n', 1#]
                                            in fib.wrk n'_less_1; } in
            let# n'_less_2 = minusInt# [n', 2#] in
            letpar fib_n_less2 = fib.wrk n'_less_2 in
            case fib_n_less1 of { Int fib_n'_less_1 ->
            case fib_n_less2 of { Int fib_n'_less_2 ->
            let# sum_2_fibs' = plusInt# [fib_n'_less_1, fib_n'_less_2] in
            let# result = plusInt# [sum_2_fibs', 1#]
            in Int [result];          }; };
 };
```

Figure 9.31: The parallel STG′ **fib2** -O benchmark

figure 9.31. This simple change ensures that a processor retains a significant portion of the work for itself, irrespective of the distribution mechanism. This produced the speedup curves shown in figure 9.32, which exhibit far better scalability than the **fib** benchmark. By re-writing the SEND_WORK rule to retain sufficient local work, it is possible to achieve similar results for the **fib** benchmark. However, it is easy to envisage programs where this policy would be equally damaging.

The sensitivity of the system to changes in the grain size and ordering is not surprising. The communication overheads are sufficiently high such that a spark has to represent a significant amount of work before it is worth distributing it. It is therefore not surprising that there is a significant body of work dealing with estimating the grain size of general expressions – Sands [1990] provides an excellent introduction to this field.

Figure 9.34 shows the speedup curves for **fib2** 15 for a range of communication costs (the key details the message-latency parameter, L). All curves achieve significant speedups, with similar results being observed for low numbers of processors. The best overall result is obtained when the costs are very low, and performance is only slightly inferior to that for the GMSV system. However, the results for the low-cost situation are poor when compared to both the medium- and high-cost situations. Further investigation revealed the source of this unexpected result: the load-distribution mechanism. The passive load-balancing works well when there is sufficient work available for all of the processors. This is the situation in the early and mid phases of the computation, or when the number of processors is low. However, as soon as work becomes scarce, a large number of *Fish* messages are injected into the system. This hinders the processors that are performing useful computations. For the low communication costs, more *Fish* messages can be generated and re-spawned within a fixed period than for the mid- and high-cost scenario. Table 9.35 lists the total number of messages sent during two particular runs, and figure 9.36 histograms the number of *Fish* messages for a range of costs (each run used 15 processors). The figures shows that, on average, a processor will receive over ten times the number of *Fish* messages with low-cost communications. As the overhead parameter, o, is comparable for all of the runs (except for the very low cost model), this causes the observed poor performance.

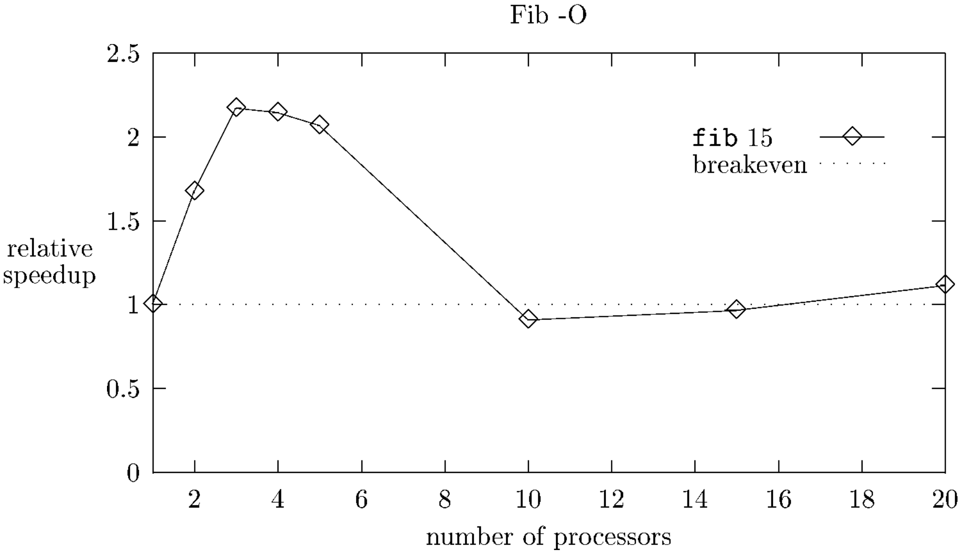


Figure 9.32: Relative speedups for the conservative `fib` -O benchmark

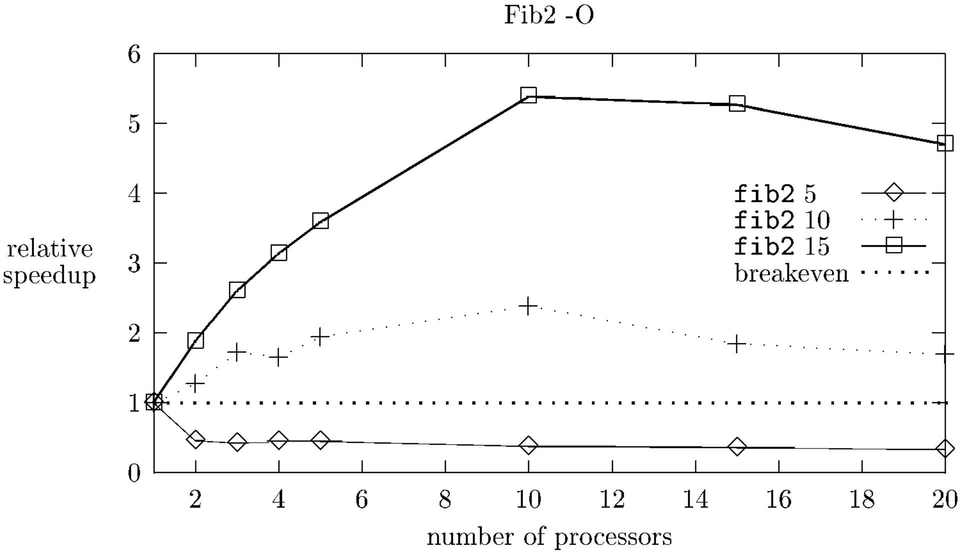


Figure 9.33: Relative speedups for the conservative `fib2` -O benchmark

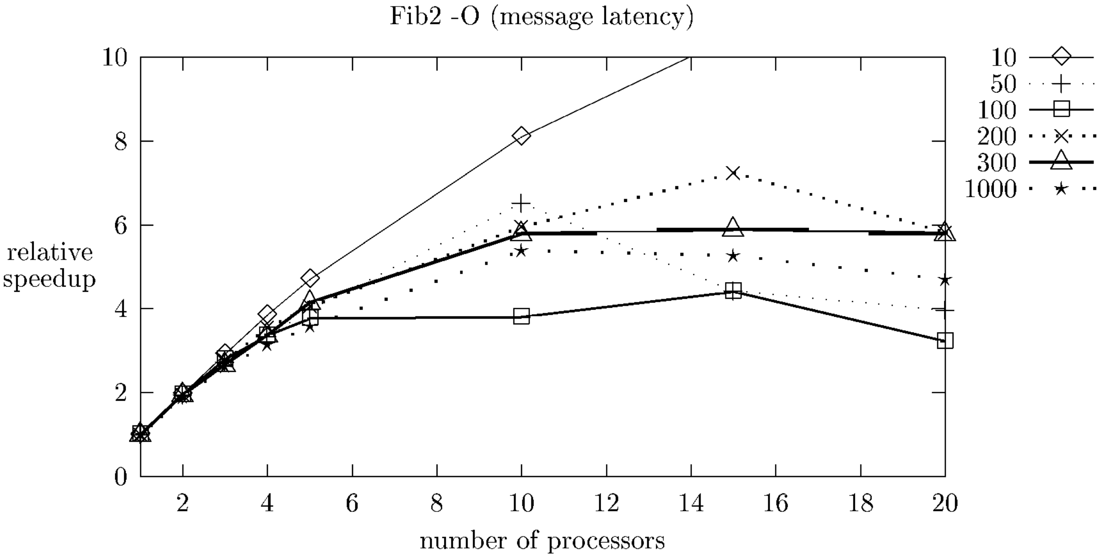
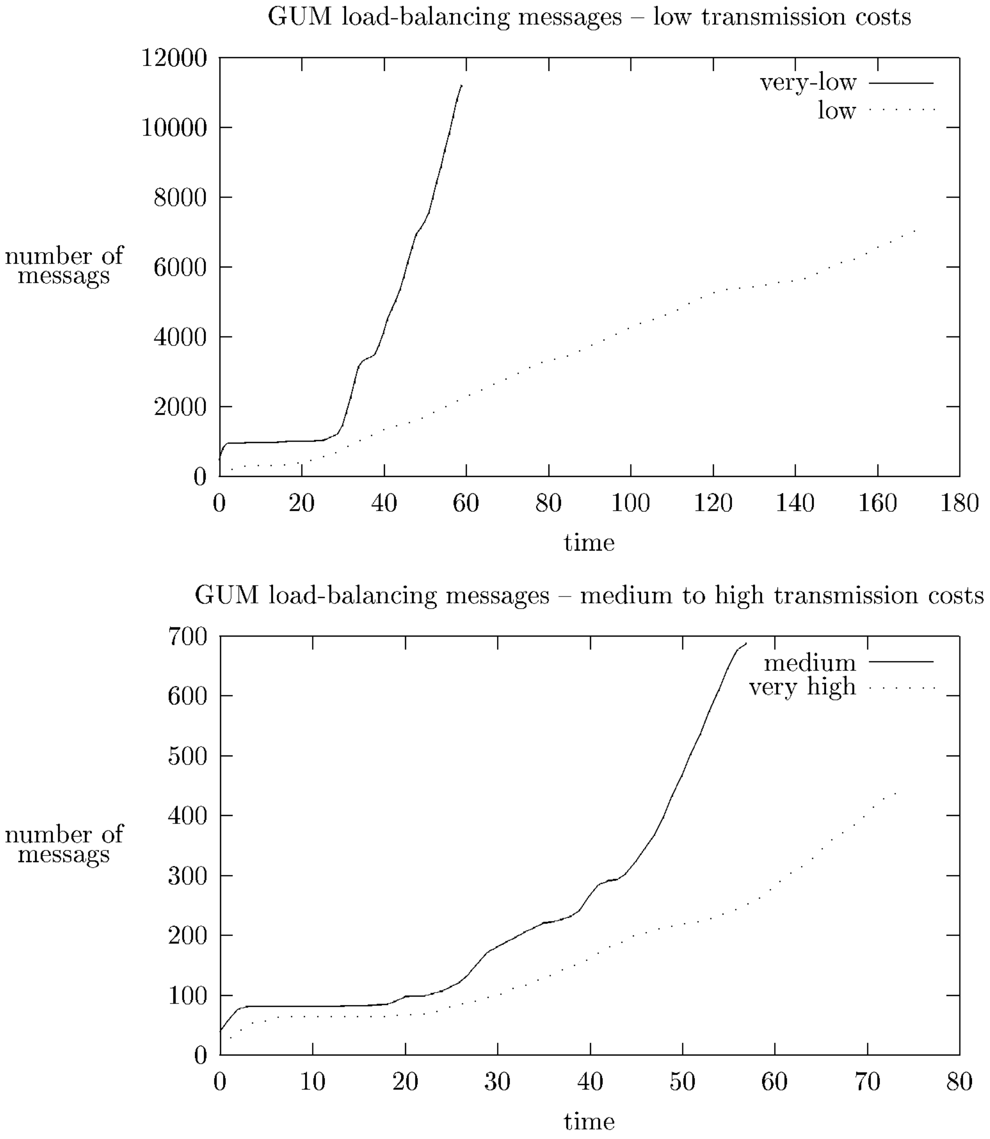


Figure 9.34: The impact of message latency on the `fib2` -O benchmark

message type	number of messages	
	$L = 50$	$L = 200$
Fish	7020	688
Schedule	217	93
Ack	217	93
Fetch	264	98
Resume	264	98
Exit	14	14

Figure 9.35: Total messages sent during the `fib2` 15



GUM load-balancing messages – low transmission costs

GUM load-balancing messages – medium to high transmission costs

Figure 9.36: Communication costs and GUM load-balaning messages

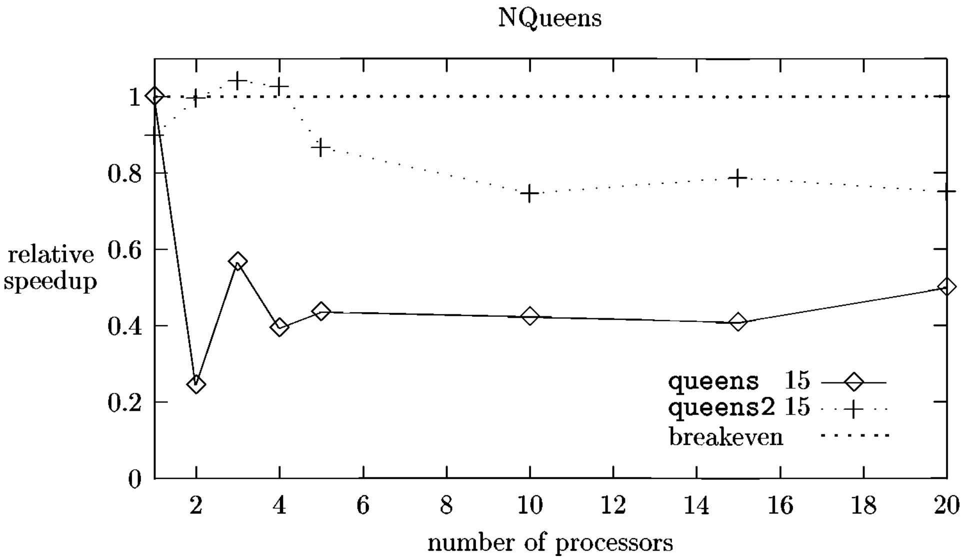


Figure 9.37: Speedups for the queens and queens2 -O benchmarks

The queens benchmark

Figure 9.37 shows the speedup curve for the queens 6 benchmark. The results are worse than even those for the unmodified fib benchmark. As with fib2, queens2 is a modified version of the benchmark, which attempts to increase the grain size of the computation by strictly evaluating the list comprehensions for the subproblems:

STG′ code

```
gen_comprehension' = [] \r [nq ds] ->
  case  ds  of {
    Nil    -> Nil [];
    : x xs -> letpar tl = gen_comprehension' nq xs in
              let { qs = [nq] \u [] -> const.Int.enumFromTo one nq; } in
              let { n  = [tl x nq qs] \u [] -> sc.TB5n tl x nq qs; } in
              case length v of { Int x -> n; };
  };
```

The length method is used to force the computation of the entire list, and therefore plays the role of an evaluation transformer, as described by Burn [1991] (see section 2.4.4). The speedup curve for queens2 is also shown in figure 9.37, and is almost twice as efficient, despite performing some unnecessary work. However, these results are still poor. The combination of the low number of available threads and the high degree of interaction between them is such that GUM's unstructured placement and scheduling of tasks is sub-optimal. It is likely, however, that increasing the problem size would significantly improve the speedup curves – queens 10 is typically used to for benchmarking real GMSV and DMMP implementations. The HDG-machine [Kingdon, Lester and Burn, 1991] is one of the few DMMP exceptions, achieving a speedup of just under three for queens 6 on four processors. However, as the HDG-machine uses a primitive model of graph-reduction, combined with the fast Transputer communication network, any comparison would be unfair.

9.3.6 Assessment

GUM's operational model is considerably more complex than that of the speculative GMSV case study. Fortunately, the scheduling and synchronisation rules from the previous study could be re-used after only minor modifications. However, designing and testing the load-balancing and remote-reference mechanisms was sufficiently involved that it would have been almost impossible without the use of the UML interaction diagrams and the STG animation. Using these tools, most of the complexity disappeared, and the techniques described in chapter 6 proved satisfactory. Indeed, the animation quickly revealed unthought of run-time interactions, and led directly to the development of the FETCH_4 and UNBLOCK rules. Throughout the development, the denotational semantics provided reference points against which the operational model could be tested for correctness.

The main limitation with the testing was the problem size that could be handled by the STG animation (the RISC animation was used only to confirm that the STG animation was producing credible results). While this is a common problem with simulations, the STG animation could cope with larger problems sizes if used on a more powerful workstation (see table 4.8, which includes sequential results for `queens` 8). Furthermore, larger problem sizes tend to hide the effect of inefficient use of the communication network, and it could be argued that small problems are therefore better for debugging performance.

9.4 Para-functional Haskell: data placement

Para-functional Haskell [Hudak, 1991] is the composition of two meta-languages, one for defining what is to be computed (Haskell), and the other for specifying how it is to be evaluated, i.e. a co-ordination language [Gelernter and Carriero, 1992]. The co-ordination operators fall into two categories: data placement, for controlling where the evaluation should take place; and scheduling, for specifying the order of evaluation. This case study is concerned with data placement, and, more specifically, the on operator.

While para-functional Haskell has only been implemented on a GMSV architecture (the Encore Multimax), ParAfl (a pre-cursor to para-functional Haskell, [Hudak, 1988]) was ported to the Intel iPSC (DMMP). Both implementations exhibited significant relative speedups [Hudak, 1988, figure 2, page 57] for a matrix-multiplication benchmark. The GMSV system achieved almost linear speedup with upto twelve processors, while The DMMP implementation managed a maximum speedup of just under three on fifteen processors. More recently, Mirani and Hudak [1995] have used monads [Peyton Jones and Wadler, 1993] to structure and enhance the communication language. The GMSV implementation (running on a sixteen processor Silicon Graphics Challenge) has demonstrated relative speedups on a range of benchmarks: matrix multiplication, `fib`, `queens`, and a sorting algorithm.

9.4.1 Static semantics

This section looks at the static semantics of a para-functional STG′ language. The following informal description of para-functional Haskell will serve as the motivator for the remainder of the section.

The informal semantics of para-functional Haskell

Para-functional Haskell uses mapped expressions to control the placement (and evaluation) of expressions: *exp* on *proc*. This declares that *exp* should be evaluated on the processor identified by *proc*. Consider the following example:

para-functional Haskell

```
let x   = f 2
    y   = f 3
    f a = a * a
in  (+ (x on 1) (y on 2)) on 3
```

The (simplified) allocation of tasks to processors is as follows:

processor	task
1	$2*2$
2	$3*3$
3	$4+6$

However, the evaluation will still proceed sequentially. Para-functional Haskell used `sched` expressions to spark parallel tasks. As a simple example, $f\ x$ `sched` Dx, denotes that x can be evaluated in parallel with the application $f\ x$. To allow the construction of topologies, the `self` operator returns the id of the local processor:

para-functional Haskell

```
divconq split combine endtest endval = f
  where
  f x | endtest x = endval x
      | otherwise = combine left right sched Dleft|Dright
                    where (l, r) = split x
                          left   = f l on self - 1
                          right  = f r on self + 1
```

Note that these are virtual topologies, as para-functional Haskell does not provide access to the current number of real processors. The run-time system is responsible for the mapping between real and virtual processors.

The para-functional STG′ language

Having had an informal look at the on, `sched`, and `self` expressions, there are two possible approaches to developing an STG′ variant. The first would be to preserve the distinction between mapping and scheduling expressions, while the second would combine them. Both would result in new variants of the `let` expression. However, looking at the limited collection of para-functional programs, the on operator never appears apart from the `sched` construct. The second approach was therefore selected, and figure 9.38 shows the abstract syntax, and free-variable, and type-inference rules of the `leton` expression.

With regards to the `self` construct, either an expression or primitive function could be used. However, section 5.2 clearly recommends the use an expression, and figure 9.39 shows the necessary definitions. Note that processor ids are represented by unboxed integers, allowing the full complement of arithmetic operators to be used to manipulate them. While the type system could have been extended to include a `Pid#` type, this would complicate the specification of virtual topologies for no real benefit.

The new expressions are related to the traditional `letpar` as follows:

$$\begin{array}{lcl} \texttt{letpar}\ \textit{simple\_bind exp} & \equiv & \texttt{let\#}\ \textit{pid} = \texttt{self in} \\ & & \texttt{leton}\ \textit{pid simple\_bind exp} \end{array}$$

abstract syntax	$exp \ \Longrightarrow\ $ `leton` $atom\ simple\_bind\ exp \mid \cdots$	task-mapping expression
free variables	$\mathcal{FV}_{exp}[\![\texttt{leton}\ pid\ var = exp_{rhs}\ exp_{body}]\!]\ g$ $= \mathcal{FV}_{atom}[\![pid]\!]\ g \cup \mathcal{FV}_{exp}[\![exp_{rhs}]\!]\ g \cup (\mathcal{FV}_{exp_{body}}[\![exp]\!]\ g \setminus \{var\})$	
type inference	$LETON\text{-}EXP\ \dfrac{\begin{array}{l} TE \overset{\text{atom}}{\vdash} pid : \texttt{Int\#} \\ TE \overset{\text{simplebind}}{\vdash} simplebind : (var, \chi\ \pi_1 \ldots \pi_v) \\ LVE = \{var \mapsto \chi\ \pi_1 \ldots \pi_v\} \\ TE \overset{\rightarrow}{\oplus} LVE \overset{\text{exp}}{\vdash} exp : \tau_{exp} \end{array}}{TE \overset{\text{exp}}{\vdash} \texttt{leton}\ pid\ simplebind\ exp : \tau_{exp}}$	

Figure 9.38: The **leton** expression: abstract syntax, free variables, and type inference

abstract syntax	$exp \ \Longrightarrow\ $ **self** $\mid \cdots$	processor id
free variables	$\mathcal{FV}_{exp}[\![\texttt{self}]\!]\ g = \{\}$	
type inference	$SELF\text{-}EXP\ TE \overset{\text{exp}}{\vdash} \texttt{self} : \texttt{Int\#}$	

Figure 9.39: The **self** expression: abstract syntax, free variables, and type inference

Denotational semantics

Based on the informal description given at the start of the section, it would appear that the `self` expression has introduced non-determinism into the language. To avoid this unwanted situation, Hudak [1986] extended the denotational semantics of a simple functional language to include the notion of location. Most of the semantic functions are extended to take the current processor id as an additional argument:

$$\boxed{\mathcal{E}[\![exp]\!] \;:\; \mathbf{Env} \to \mathbf{Pid} \to \mathbf{Val}}$$

At the start of evaluation, the current processor is set to zero:

$$\begin{array}{rcl} \mathcal{P}\mathit{rogram}[\![program]\!] & : & \mathbf{Val} \\ \mathcal{P}\mathit{rogram}[\![typedecls\ binds]\!] & = & \mathcal{E}[\![\texttt{letrec}\ binds\ \texttt{main}]\!]\ \{\}_{env}\ 0_{pid} \end{array}$$

The majority of the semantic equations either ignore the processor id, or simply thread it through their sub-expressions. Only the `leton` and `self` expressions actually manipulate the new parameter:

$$\begin{array}{rcl} \mathcal{E}[\![\texttt{self}]\!]\ \rho\ cpid & = & cpid \\ \mathcal{E}[\![\texttt{leton}\ pid\ var = exp_{rhs}\ exp_{body}]\!]\ \rho\ cpid & = & case\ (\mathcal{E}[\![exp_{rhs}]\!]\ \rho\ cpid')\ of \\ & & \quad \begin{array}{rcl} \bot & \to & \bot \\ \epsilon & \to & \mathcal{E}[\![exp_{body}]\!]\ \rho'\ cpid \end{array} \\ & & \begin{array}{rcl} \text{where } \rho' & = & \rho \overset{\to}{\oplus} \{var \mapsto \epsilon\} \\ cpid' & = & \mathcal{A}\mathit{tom}[\![pid]\!]\ \rho \end{array} \end{array}$$

The following para-functional STG$'$ program, for example, denotes the value `Int` 0:

STG$'$ code

```
main = [] \u [] -> let#  x = self# in
                   let#  right =  plusInt# [x, int 1#] in
                   let#  left  = minusInt# [x, int 1#] in
                   leton right a = let# here = self# in Int [here] in
                   leton left  b = let# here = self# in Int [here] in
                   const.Int.+ a b;
```

Execution-tree semantics

In addition to the standard denotational semantics, Hudak [1986, section 5, pages 113–119] also provided an execution-tree semantics for a simple para-functional language. Loosely based on his work on pomsets [Hudak and Anderson, 1987], the semantics generates an evaluation history for the program. However, it does not model sharing within the programs, so the execution path will always be a tree. This history can be visualised and used to ensure the operational model and compilation rules are behaving correctly. For example, figures 9.40 and 9.41 show the execution trees for `fib` 5 and `queens` 2. Notice that the `queens` benchmark has a significantly more complex and irregular tree than that for the `fib` benchmark. It is highly likely that the execution-tree semantics could be refined to provide programmers with a tool for identifying and controlling potential parallel tasks.

The execution-tree semantics, $\mathcal{T}[\![\text{exp}]\!]$, builds upon the denotational semantics, using it to determine which `case` alternatives will be selected during evaluation. Figure 9.42 shows the additional domain equations used by the semantics. A behaviour comprises

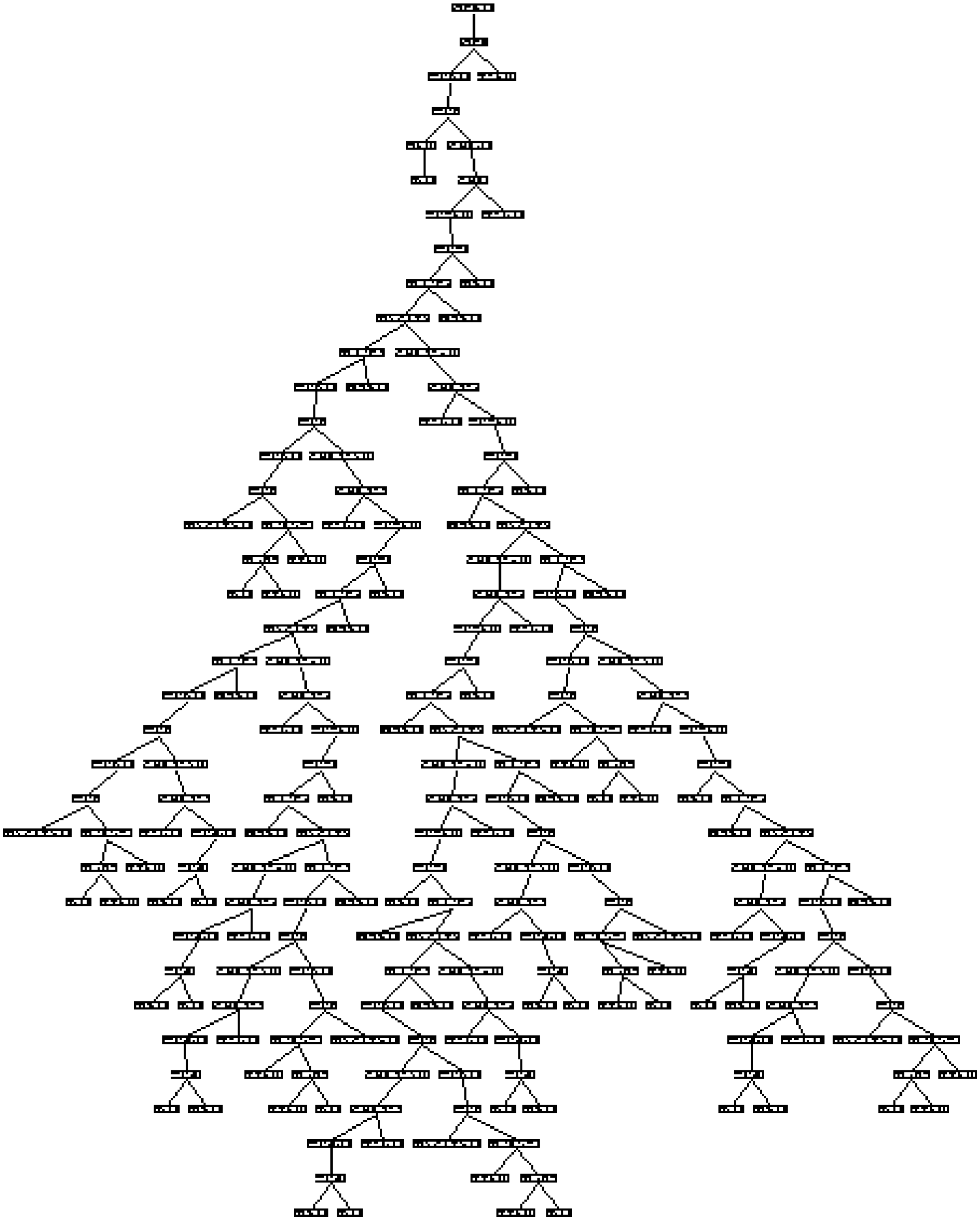


Figure 9.40: The execution tree for `fib` 5

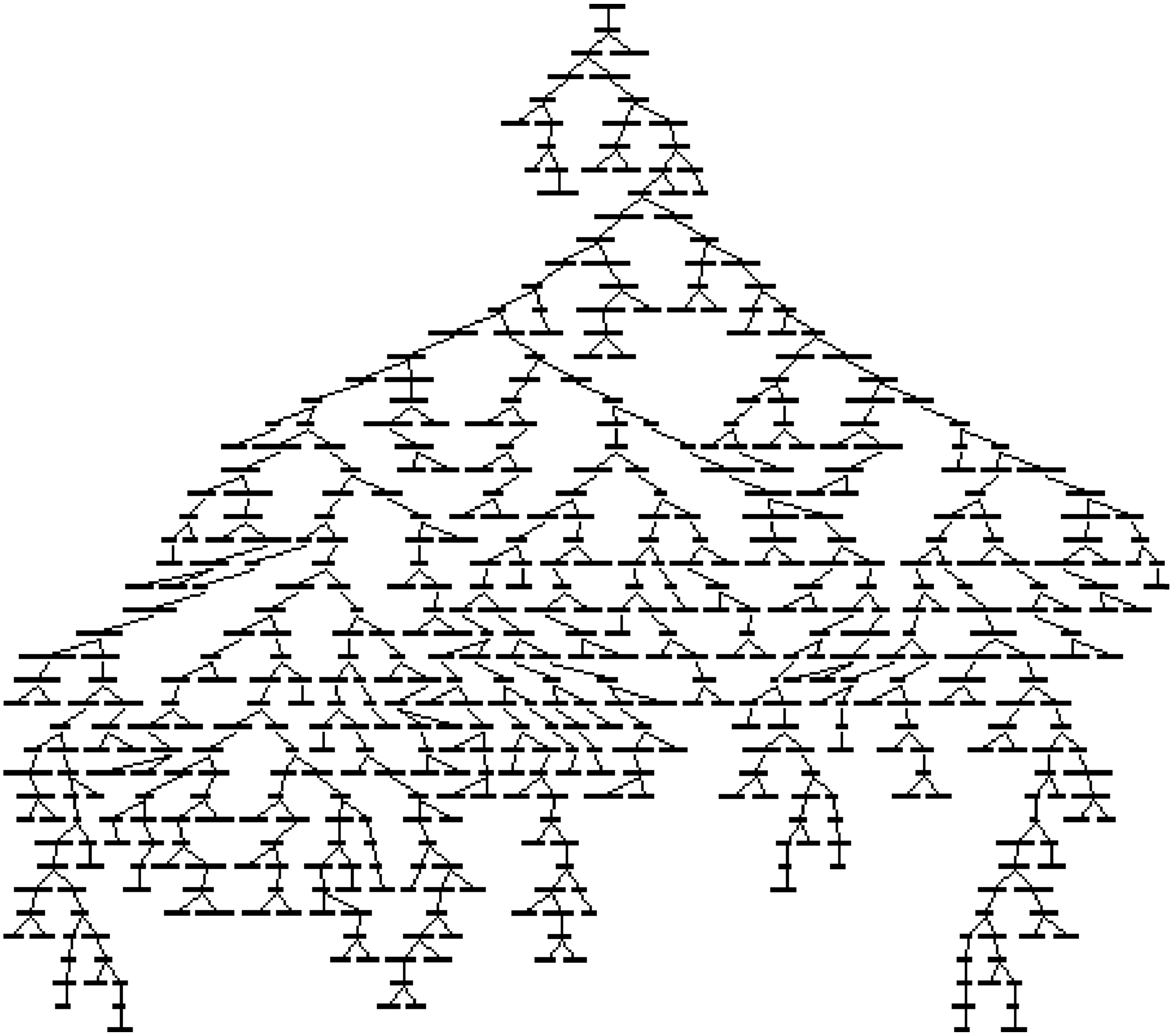


Figure 9.41: The execution tree for queens 2

Beh	=	**Etree** × **AbsBeh**	Behaviours
Etree	=	**Pid**	Simple evaluation
	∪	(**Pid** × **Etree**)	Sequential evaluation
	∪	(**Pid** × **Etree** × **Etree**)	Sub-expression evaluation
Pid	≡	**I#**	Processor ids
AbsBeh	=	**Beh** → **Val** → **Pid** → **Beh**	Function behaviours
	∪	**Id** → **Beh**	Constructor behaviours
BEnv	=	**Id** → **Beh**	Behaviour env.

Figure 9.42: Recursive execution-tree domain equations

$$
\begin{array}{rcl}
\mathcal{T}_p[\![\mathit{program}]\!] & : & \mathbf{Beh} \\
\mathcal{T}_p[\![\mathit{typedecls\ binds}]\!] & = & \mathcal{T}[\![\texttt{letrec}\ \mathit{binds}\ \texttt{main}]\!]\ \{\}_{env}\ \{\}_{env}\ 0_{pid} \\
\\
\mathcal{T}_{binds}[\![\mathit{binds}]\!] & : & \mathbf{BEnv} \rightarrow \mathbf{Env} \rightarrow \mathbf{Pid} \rightarrow \mathbf{BEnv} \\
\mathcal{T}_{binds}[\![\mathit{bind}_1 \ldots \mathit{bind}_n]\!]\ \mathit{be}\ \rho\ \mathit{cp} & = & \bigoplus_{i \leq n} \mathcal{T}_{bind}[\![\mathit{bind}_i]\!]\ \mathit{be}\ \rho\ \mathit{cp} \\
\\
\mathcal{T}_{bind}[\![\mathit{bind}]\!] & : & \mathbf{BEnv} \rightarrow \mathbf{Env} \rightarrow \mathbf{Pid} \rightarrow \mathbf{BEnv} \\
\mathcal{T}_{bind}[\![\mathit{var} = \mathit{lambda\_form}]\!]\ \mathit{be}\ \rho\ \mathit{cp} & = & \{\mathit{var} \rightarrow \mathcal{T}_{lf}[\![\mathit{lambda\_form}]\!]\ \mathit{be}\ \rho\ \mathit{cp}\} \\
\\
\mathcal{T}_{lf}[\![\mathit{lambda\_form}]\!] & : & \mathbf{BEnv} \rightarrow \mathbf{Env} \rightarrow \mathbf{Pid} \rightarrow \mathbf{Beh} \\
\mathcal{T}_{lf}[\![\mathit{fvs}\ \pi\ v_1 \cdots v_n \rightarrow \mathit{exp}]\!]\ \mathit{be}\ \rho\ \mathit{cp} & = & (\mathit{cp}, \lambda b_1\ \epsilon_1\ p_1 \ldots b_n\ \epsilon_n\ p_n . \mathcal{T}[\![\mathit{exp}]\!]\ \mathit{be}'\ \rho'\ p_n) \\
 & & \text{where } \mathit{be}' = \mathit{be} \overset{\rightarrow}{\oplus} \{v_1 \mapsto b_1, \ldots, v_n \mapsto b_n\} \\
 & & \qquad\quad \rho' \ = \rho \ \ \overset{\rightarrow}{\oplus} \{v_1 \mapsto \epsilon_1, \ldots, v_n \mapsto \epsilon_n\}
\end{array}
$$

Figure 9.43: Execution-tree semantics of para-functional STG′ programs and bindings

two parts, the first is the execution tree, and the second is an abstract behaviour. The abstract behaviour is used by `let` and `case` expressions to model the non-strict evaluation ordering. Using Hudak's notation, for a behaviour b, b_t denotes its execution tree, and b_f denotes its abstract behaviour.

Each semantic equation typically returns a behaviour, and, in addition to the value environment, ρ, and current processor id, cp, maintains a behaviour environment, be:

$$\mathcal{T}[\![\mathit{exp}]\!] \ : \ \mathbf{BEnv} \rightarrow \mathbf{Env} \rightarrow \mathbf{Pid} \rightarrow \mathbf{Beh}$$

Figures 9.43, 9.44, and 9.45 show the various equations used by the execution-tree semantics. In most instances, there is a close correspondence between Hudak's definitions and those used here (for a comprehensive description of the ideas and techniques, the interested reader is referred to [Hudak, 1986]). The main extension corresponds to the STG′ language's use of algebraic constructors. While `let(rec)` expressions are the only source of delayed evaluation within the STG′ language, constructors and `case` expressions need to maintain the associated non-strict behaviours across function calls. To this end, the domain of abstract behaviours has been extended to include a constructor map (see the constructor and algebraic-alternative equations in figures 9.44 and 9.45). Further modifications were required to model the strict evaluation of `let#`, `letstrict`, and `leton` expressions.

9.4.2 The operational semantics

The operational semantics presented here builds upon those used in the previous section. The most obvious strategy is to modify the LETPAR rule so that it sends a *Schedule* message to the targeted processor, as shown in figure 9.46. The LETON$_1$ rule ensures that work targeted for the local processor is directly added to the work pool. The LETON$_2$ rules packs up the work and sends it to the specified processor. The SELF rule returns the processor's id when evaluating a `self` expression. Note, however, that this simple approach is only correct with regards to the denotational semantics when there are more physical processors than virtual processors. Consider, for instance, the following code:

$$\mathcal{T}[\![exp]\!] : \mathbf{BEnv} \rightarrow \mathbf{Env} \rightarrow \mathbf{Pid} \rightarrow \mathbf{Beh}$$

$$\begin{aligned}
&\mathcal{T}[\![\texttt{let}\ binds\ exp]\!]\ be\ \rho\ cp = \mathcal{T}[\![exp]\!]\ be'\ \rho'\ cp\\
&\quad \text{where } be' = be \overset{\rightarrow}{\oplus} \mathcal{T}_{binds}[\![binds]\!]\ be\ \rho\ cp\\
&\qquad\qquad \rho' \ = \rho \ \overset{\rightarrow}{\oplus} \mathcal{B}inds[\![binds]\!]\ \rho
\end{aligned}$$

$$\begin{aligned}
&\mathcal{T}[\![\texttt{letrec}\ binds\ exp]\!]\ \rho = \mathcal{T}[\![exp]\!]\ (be \overset{\rightarrow}{\oplus} be')\ (\rho \overset{\rightarrow}{\oplus} \rho')\ cp\\
&\quad \text{where } (be', \rho') = fix\ (\lambda(be', \rho').(\mathcal{T}_{binds}[\![binds]\!]\ (be \overset{\rightarrow}{\oplus} be')\ (\rho \overset{\rightarrow}{\oplus} \rho')\ cp,\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathcal{B}inds[\![binds]\!]\ (\rho \overset{\rightarrow}{\oplus} \rho')))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{T}[\![\texttt{let\#}\ var = exp_{rhs}\ exp]\!]\ be\ \rho\ cp = (cp : (b1_t, b2_t), b2_f)\\
&\quad \text{where } b1 = \mathcal{T}[\![exp_{rhs}]\!]\ be\ \rho\ cp\\
&\qquad\qquad b2 = case\ \ (\mathcal{E}[\![exp_{rhs}]\!]\ \rho\ cp)\ of\\
&\qquad\qquad\qquad\quad \bot \rightarrow \bot\\
&\qquad\qquad\qquad\quad \epsilon\ \ \rightarrow \mathcal{T}[\![exp_{body}]\!]\ (be \overset{\rightarrow}{\oplus} \{var \mapsto (cp, err)\})\ (\rho \overset{\rightarrow}{\oplus} \{var \mapsto \epsilon\})\ cp
\end{aligned}$$

$$\begin{aligned}
&\mathcal{T}[\![\texttt{letstrict}\ var = exp_{rhs}\ exp_{body}]\!]\ be\ \rho\ cp = (cp : (b1_t, b2_t), b2_f)\\
&\quad \text{where } b1 = \mathcal{T}[\![exp_{rhs}]\!]\ be\ \rho\ cp\\
&\qquad\qquad b2 = case\ \ (\mathcal{E}[\![exp_{rhs}]\!]\ \rho\ cp)\ of\\
&\qquad\qquad\qquad\quad \bot \rightarrow \bot\\
&\qquad\qquad\qquad\quad \epsilon\ \ \rightarrow \mathcal{T}[\![exp_{body}]\!]\ (be \overset{\rightarrow}{\oplus} \{var \mapsto (cp, err)\})\ (\rho \overset{\rightarrow}{\oplus} \{var \mapsto \epsilon\})\ cp
\end{aligned}$$

$$\begin{aligned}
&\mathcal{T}[\![\texttt{leton}\ pid\ var = exp_{rhs}\ exp_{body}]\!]\ be\ \rho\ cp = (cp : (b1_t, b2_t), b2_f)\\
&\quad \text{where } b1 = \mathcal{T}[\![exp_{rhs}]\!]\ be\ \rho\ (\mathcal{A}tom[\![pid]\!]\ \rho)\\
&\qquad\qquad b2 = case\ \ (\mathcal{E}[\![exp_{rhs}]\!]\ \rho\ cp)\ of\\
&\qquad\qquad\qquad\quad \bot \rightarrow \bot\\
&\qquad\qquad\qquad\quad \epsilon\ \ \rightarrow \mathcal{T}[\![exp_{body}]\!]\ (be \overset{\rightarrow}{\oplus} \{var \mapsto (cp, err)\})\ (\rho \overset{\rightarrow}{\oplus} \{var \mapsto \epsilon\})\ cp
\end{aligned}$$

$$\begin{aligned}
&\mathcal{T}[\![\texttt{case}\ exp\ alts\ default]\!]\ be\ \rho\ cp = (cp : (bexp_t, balt_t), balt_f)\\
&\quad \text{where } balt \quad\ \ = \ \mathcal{T}_{alts}[\![alts]\!]\ be\ \rho\ cp\ bexp\ b_{default}\ \epsilon_{exp}\\
&\qquad\qquad bexp \quad\ \ = \ \mathcal{T}[\![exp]\!]\ be\ \rho\ cp\\
&\qquad\qquad b_{default} \ = \ \mathcal{T}_{default}[\![default]\!]\ be\ \rho\ cp\\
&\qquad\qquad \epsilon_{exp} \quad\ \ \ = \ \mathcal{E}[\![exp]\!]\ \rho\ cp
\end{aligned}$$

$$\begin{aligned}
&\mathcal{T}[\![var_{fun}\ atom_1 \ldots atom_n]\!]\ \rho\ cp = (cp : (bfun_t, bresult_t), bresult_f)\\
&\quad \text{where } \epsilon_i \qquad\ = \ \mathcal{A}tom[\![atom_i]\!]\ \rho,\ \ (1 \leq i \leq n)\\
&\qquad\qquad b_i \qquad\ = \ \mathcal{T}_{atom}[\![atom_i]\!]\ be\ \rho\ cp,\ \ (1 \leq i \leq n)\\
&\qquad\qquad bfun_0 \quad = \ be\ var_{fun}\\
&\qquad\qquad bfun_i \quad\ = \ bfun_{i-1}\ b_i\ \epsilon_i\ cp,\ \ (1 \leq i \leq n)\\
&\qquad\qquad bresult \ \ = \ bfun_n
\end{aligned}$$

$$\begin{aligned}
&\mathcal{T}[\![cons\ atom_1 \ldots atom_n]\!]\ be\ \rho\ cp = (cp, \{1 \mapsto b_1, \ldots, n \mapsto b_n\})\\
&\quad \text{where } b_i = \mathcal{T}_{atom}[\![atom_i]\!]\ be\ \rho\ cp
\end{aligned}$$

$$\mathcal{T}[\![literal]\!]\ be\ \rho\ cp = (cp, err)$$

$$\mathcal{T}[\![\mathtt{self}]\!]\ be\ \rho\ cp = (cp, err)$$

$$\begin{aligned}
&\mathcal{T}_{default}[\![default]\!] : \mathbf{BEnv} \rightarrow \mathbf{Env} \rightarrow \mathbf{Pid} \rightarrow \mathbf{Beh}\\
&\mathcal{T}_{default}[\![\_ \rightarrow exp]\!]\ be\ \rho\ cp \rightarrow \mathcal{T}[\![exp]\!]\ be\ \rho\ cp
\end{aligned}$$

$$\begin{aligned}
&\mathcal{T}_{atom}[\![atom]\!] : \mathbf{BEnv} \rightarrow \mathbf{Env} \rightarrow \mathbf{Pid} \rightarrow \mathbf{Beh}\\
&\mathcal{T}_{atom}[\![var]\!]\ be\ \rho\ cp = be\ var\\
&\mathcal{T}_{atom}[\![literal]\!]\ be\ \rho\ cp = (cp, err)
\end{aligned}$$

Figure 9.44: Execution-tree semantics of para-functional STG$'$ expressions

$$\mathcal{T}_{alts}[\![alts]\!] \;:\; \mathbf{BEnv} \to \mathbf{Env} \to \mathbf{Pid} \to \mathbf{Beh} \to \mathbf{Beh} \to \mathbf{Val} \to \mathbf{Beh}$$

$$\mathcal{T}_{alts}\left[\!\!\left[\begin{array}{lcl} cons_1\ var_{11}\ldots var_{1a_1} & \to & exp_1 \\ & \vdots & \\ cons_n\ var_{n1}\ldots var_{na_n} & \to & exp_n \end{array}\right]\!\!\right] be\ \rho\ cp\ bexp\ b_{default}\ \epsilon_{exp}$$

$$\begin{array}{l} = case\ \epsilon_{exp}\ of \\ \quad \begin{array}{lll} \Box\ \bot & \to & \bot \\ \Box\ \langle cons_1, \epsilon_{11}, \ldots, \epsilon_{1a_1}\rangle & \to & \mathcal{T}[\![exp_1]\!]\ be_1\ \rho_1\ cp \\ \Box\ \ldots & & \\ \Box\ \langle cons_n, \epsilon_{n1}, \ldots, \epsilon_{na_n}\rangle & \to & \mathcal{T}[\![exp_n]\!]\ be_n\ \rho_n\ cp \\ \Box\ else & \to & b_{default} \end{array} \\ \begin{array}{rrcl} \text{where} & be_i & = & be \overset{\to}{\oplus} \{var_{i1} \mapsto b_{i1}, \ldots, var_{ia_i} \mapsto b_{ia_i}\} \\ & b_{ij} & = & bexp_f\ j \\ & \rho_i & = & \rho \overset{\to}{\oplus} \{var_{i1} \mapsto \epsilon_{i1}, \ldots, var_{ia_i} \mapsto \epsilon_{ia_i}\} \end{array} \end{array}$$

$$\mathcal{T}_{alts}\left[\!\!\left[\begin{array}{lcl} literal_1 & \to & exp_1 \\ & \vdots & \\ literal_n & \to & exp_n \end{array}\right]\!\!\right] be\ \rho\ cp\ bexp\ b_{default}\ \epsilon_{exp}$$

$$= case\ \epsilon_{exp}\ of\ \begin{array}{lll} \Box\ \bot & \to & \bot \\ \Box\ literal_1 & \to & \mathcal{T}[\![exp_1]\!]\ be\ \rho\ cp \\ \Box\ \ldots & & \\ \Box\ literal_n & \to & \mathcal{T}[\![exp_n]\!]\ be\ \rho\ cp \\ \Box\ else & \to & b_{default} \end{array}$$

Figure 9.45: Execution-tree semantics of para-functional STG′ `case` alternatives

(LETON$_1$)

$$\begin{array}{llllllllll} & Eval\ (\texttt{leton}\ pid\ v = e_1\ e_2)\ \rho & as & rs & us & h & t_{id} & wp & \sigma & b_i \\ \multicolumn{10}{l}{\text{such that } pid' == i} \\ \Longrightarrow & Eval\ e_2\ (\rho \overset{\to}{\oplus} \{v \mapsto a\}) & as & rs & us & h' & t_{id} & wp' & \sigma & b_i \end{array}$$

$$\begin{array}{rrcl} \text{where} & h' & = & h[a \mapsto create\_closure\ e_1\ \rho] \\ & wp' & = & insert_{spark}\ a\ wp \\ & pid' & = & (val\ \rho\ \sigma\ pid)\%n \end{array}$$

(LETON$_2$)

$$\begin{array}{llllllllll} & Eval\ (\texttt{leton}\ pid\ v = e_1\ e_2)\ \rho & as & rs & us & h & t_{id} & wp & \sigma & b_i \\ \Longrightarrow & Send\ schedule\ afterwards & as & rs & us & h' & t_{id} & wp & \sigma & b_i \end{array}$$

$$\begin{array}{rlcl} \text{where} & afterwards & = & Eval\ e_2\ (\rho \overset{\to}{\oplus} \{v \mapsto a\}) \\ & schedule & = & (i, pid', Schedule\ a\ (pack\ closure)) \\ & closure & = & create\_closure\ e_1\ \rho \\ & h' & = & h[a \mapsto Exported\ pid'\ closure\ bk_{empty}] \\ & pid' & = & (val\ \rho\ \sigma\ pid)\%n \end{array}$$

(SELF)

$$\begin{array}{llllllllll} & Eval\ \texttt{self}\ \rho & as & rs & us & h & t_{id} & wp & \sigma & b_i \\ \Longrightarrow & Return_{\texttt{Int\#}}\ i & as & rs & us & h & t_{id} & wp & \sigma & b_i \end{array}$$

Figure 9.46: Initial operational rules for the para-functional STG′ language

	specification	description
P	$(code, \ldots, h, t_{id}, wp, \sigma, vp_{id})$	the standard GUM abstract state extended to include a virtual processor id, vp_{id}.
vp_{id}	**Int#**	the virtual processor id associated with the current evaluation.
update frame	(a, as, rs, vp_{id})	A standard update marker extended to include the virtual id active before entry to the updatable closure.
closure	$VThunk\ vp_{id}\ exp\ \rho$	a virtual thunk.

Table 9.10: State components for supporting virtual topologies

STG′ code

```
main = [] \u [] -> leton int 1000# a = let# y = self# in Int [y] in a;
```

The denotational semantics specify that the correct result is `Int` 1000, while a one-processor operational model will produce the result `Int` 0. The load-balancing mechanism also causes problems by transporting thunks to other processors. There are three possible solutions to this problem:

1. extend the denotational semantics to take an extra argument: the maximum number of available processors. An attempt to create a virtual topology with more processor would result in an error. The STG′ language would also have to be extended to provide a way for the programmer to determine the number of available processors.[3]

2. modify the operational behaviour so that it correctly implements the denotational semantics.

3. accept the fact that the operational model is a weak model of the denotational semantics and is also non-deterministic (due to the load balancer).

From a language-design perspective, the second option is the only acceptable alternative as both of the other solutions fall short of the goals set out in table 5.1.

In order to provide support for virtual topologies a number of minor modification need to be made to the GUM operational model. Essentially, each mapped thunk is annotated with its virtual processor id. Whenever such a thunk is entered, the processor updates its virtual id, and it is this value that the `self` expression will equate to. This strategy copes with a mismatch between the number of virtual and real processors, and is also not affected by dynamic load balancing. However, one problem remains: after evaluation of a mapped thunk, the processor needs to revert its virtual id to the value it was using before the thunk was entered. The solution is to store the original virtual id inside the thunk's update frame, where it can be recovered after evaluation is complete. The modifications to the GUM's state components are shown in table 9.10, and a selection of the associated rules are shown in figure 9.47.

[3] Such an extension could also be useful in an implementation supporting truly virtual topologies. It would allow programmers to generate topologies optimised for the available number of processor. It would be almost identical in form to the `self` expression, and figures 9.39 and 9.46 could serve as a template.

(LETON'$_2$)

$$\boxed{\begin{array}{llllllllllll}
 & \mathit{Eval}\ (\texttt{leton}\ \mathit{pid}\ v = e_1\ e_2)\ \rho & as & rs & us & h & t_{id} & wp & \sigma & vp_{id} & b_i \\
\Longrightarrow & \mathit{Send}\ \mathit{schedule}\ \mathit{afterwards} & as & rs & us & h' & t_{id} & wp & \sigma & vp_{id} & b_i \\
\multicolumn{11}{l}{\text{where}\ \begin{array}{lcl}
\mathit{afterwards} & = & \mathit{Eval}\ e_2\ (\rho \overrightarrow{\oplus} \{v \mapsto a\}) \\
\mathit{schedule} & = & (i, \mathit{pid}', \mathit{Schedule}\ a\ (\mathit{pack}\ \mathit{closure})) \\
\mathit{closure} & = & \mathit{VThunk}\ vp'_{id}\ e_1\ \rho' \\
\mathit{dom}(\rho') & = & \mathcal{FV}[\![e_1]\!]\ \sigma \\
h' & = & h[a \mapsto \mathit{Exported}\ \mathit{pid}'\ \mathit{closure}\ bk_{empty}] \\
vp'_{id} & = & \mathit{val}\ \rho\ \sigma\ \mathit{pid} \\
\mathit{pid}' & = & vp'_{id}\%n \\
bk_{empty} & = & (\langle\rangle_{threads}, \langle\rangle_{fetches})_{blocked}
\end{array}}
\end{array}}$$

(BH$_5$)

$$\boxed{\begin{array}{llllllllll}
 & \mathit{Enter}\ a & as & rs & us & h & t_{id} & wp & \sigma & vp_{id}\ \ b_i \\
\multicolumn{10}{l}{\text{such that}\ h[a \mapsto \mathit{VThunk}\ vp'_{id}\ \mathit{exp}\ \rho]} \\
\Longrightarrow & \mathit{Eval}\ \mathit{exp}\ \rho & \langle\rangle_{stack} & \langle\rangle_{stack} & us' & h' & t_{id} & wp & \sigma & vp'_{id}\ \ b_i \\
\multicolumn{10}{l}{\text{where}\ \begin{array}{lcl}
us' & = & (as, rs, a, vp_{id}) : us \\
h' & = & h[a \mapsto \mathit{BlackHole} bk_{empty}] \\
bk_{empty} & = & (\langle\rangle_{threads}, \langle\rangle_{fetches})_{blocked}
\end{array}}
\end{array}}$$

(BH'$_3$)

$$\boxed{\begin{array}{llllllllll}
 & \mathit{Return}_\chi\ c\ ws & \langle\rangle & \langle\rangle & \begin{pmatrix} a_u, \\ as_u, \\ rs_u, \\ vp'_{id} \end{pmatrix} : us & h & t_{id} & wp & \sigma & vp_{id}\ \ b_i \\
\multicolumn{10}{l}{\text{such that}\ h[a_u \mapsto \mathit{BlackHole}\ \mathit{blocked}]} \\
\Longrightarrow & \mathit{unblock} & as_u & rs_u & us & h' & t_{id} & wp & \sigma & vp'_{id}\ \ b_i \\
\multicolumn{10}{l}{\text{where}\ \begin{array}{lcl}
\mathit{unblock} & = & \mathit{Unblock}\ \mathit{blocked}\ (\mathit{Return}_\chi\ c\ ws) \\
h' & = & h[a_u \mapsto (vs\ \texttt{r} \rightarrow c\ vs, ws)] \\
\mathit{length}\ vs & = & \mathit{length}\ ws \\
\multicolumn{3}{l}{vs\ \text{is a sequence of arbitrary distinct variables}}
\end{array}}
\end{array}}$$

(SELF')

$$\boxed{\begin{array}{llllllllll}
 & \mathit{Eval}\ \texttt{self}\ \rho & as & rs & us & h & t_{id} & wp & \sigma & vp_{id}\ \ b_i \\
\Longrightarrow & \mathit{Return}_{\texttt{Int\#}}\ vp_{id} & as & rs & us & h & t_{id} & wp & \sigma & vp_{id}\ \ b_i
\end{array}}$$

Figure 9.47: Operational rules for supporting virtual topologies

STG′ code

```
main  = [] \u [] -> leton int 1# fib13_1 = fib.wrk int 13# in
                    leton int 2# fib13_2 = fib.wrk int 13# in
                    case fib.wrk int 12# of { Int fib_12' ->
                    let# x = plusInt# [fib_12', int 2#] in
                    case fib13_1 of { Int fib_13_1' ->
                    let# y = plusInt# [x, fib_13_1'] in
                    case fib13_2 of { Int fib_13_2' ->
                    let# z = plusInt# [y, fib_13_2'] in
                    Int [z]; }; }; };


fib.wrk = [] \r [n'] -> case leInt# [n', 1#] of
 { True  -> Int [1#];
   False -> let# n'_less_2 = minusInt# [n', int 2#] in
            let# here = self# in
            leton self# fib_n_less2 = fib.wrk n'_less_2 in
            let# n'_less_1 = minusInt# [n', 1#] in
            case fib.wrk n'_less_1 of { Int fib_n'_less_1 ->
            case fib_n_less2 of { Int fib_n'_less_2 ->
            let# sum_2_fibs' = plusInt# [fib_n'_less_1, fib_n'_less_2] in
            let# result = plusInt# [sum_2_fibs', 1#]
            in Int [result];          }; };
 };
```

Figure 9.48: The para-functional STG′ `sfib3` -O benchmark

9.4.3 Compilation rules

The compilation rules should follow almost directly from those used with the GUM system. However, a more detail exposition is beyond the scope of this thesis.

9.4.4 Performance

If the `leton` expression is used only to schedule tasks on the local processor, then the performance is exactly the same as for the GUM system. However, by using the execution-tree semantics to structure the computation, improvements over the GUM system are possible. For example, figure 9.48 shows a specialised version of the `fib` benchmark, explicitly using three virtual processors and relying on load balancing to provide work for any remaining physical processors. The results for this benchmark and a four-processor variant, `sfib4`, are shown in figure 9.49. Both variants achieve modest improvements over their unstructured counterpart, and do not suffer from a performance degradation when the number of surplus processors is increased. This benefit is solely down to a reduced initialisation phase, where the available work spreads to the idle processors more quickly under the para-functional scheme. In effect, the first round of *Fish* messages is avoided, and the targeted processors are able to make their sparks available sooner. It is worth noting that to benefit from this speedup, the burden of supplying the mapping directives is placed upon the programmer (or, possibly, generated via an automatic mapping algorithm). Furthermore, with more complex benchmarks, running for greater periods of time, it is likely that the reduction in initialisation would only account for a minor fraction of the total runtime. However, as the complexity of the program increases, so does the scope to use the mapping directives – unfortunately a comprehensive study of the benefit of mapped expressions is beyond the scope of this thesis.

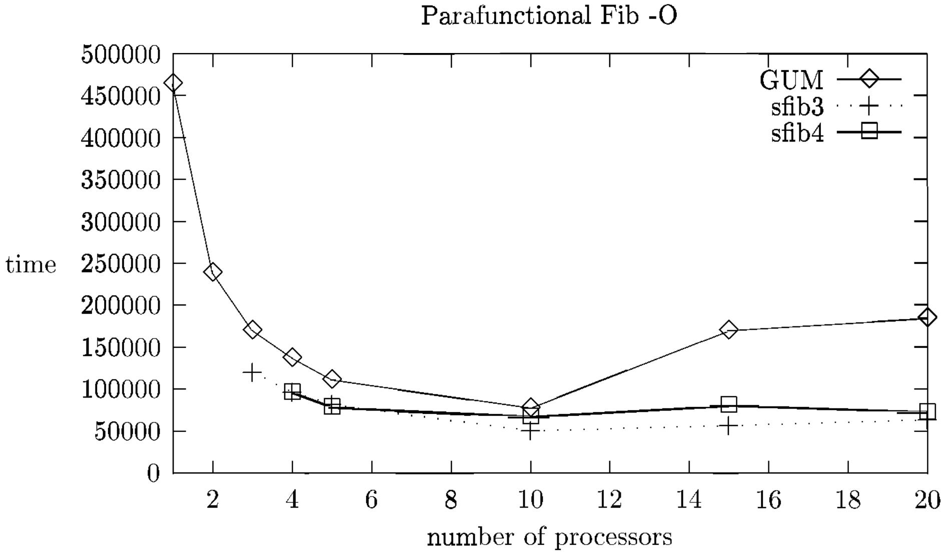


Figure 9.49: Speedup curves for the para-functional **fib** benchmarks

9.4.5 Related work

Burton [1987] was probably one of the first in the functional-programming community to look at imposing greater structure onto the traditional **par** combinator. However, his work was purely theoretical, and did not directly result in any real implementations. More practically, Hammond, Loidl and Partridge [1995a] GranSim simulator implemented the **parAt** operator, which has very similar semantics to Hudak's on expressions. However, this was not the primary focus of their work, and no results were reported. For the benchmark programs they were using, and the respective problem sizes, the dynamic load-balancing was sufficiently effective it is likely that further annotations were deemed unnecessary.

As previously mentioned, Mirani and Hudak [1995] had extended the work on para-functional Haskell to incorporate the recent advances in monads. This has allowed them to promote their scheduling annotations into first-class citizens, and provides access to run-time values (such as the current processor load) without compromising determinacy. It would be straightforward to modify the models presented here to reflect their advances.

Finally, Parrott [1993] avoided the problems associated with hand annotation by using profiling information to drive a heuristic scheduling algorithm. While his work concentrated upon when to run particular tasks, there is no obvious reason why it could not also take task locality into account. This could provide a pragmatic approach to generating initial mappings, which could then be refined by a programmer, significantly reducing the burden associated with such schemes.

9.4.6 Assessment

Following on from the GUM case study, incorporating mapped expressions into the STG′ language was straightforward: the denotational semantics are very similar, and the majority of the operational rules were used unmodified. Furthermore, the development of the execution-tree semantics provided an insight into the structure of the benchmark programs, and could be used to provide a useful tool for the parallel functional programmer.

Finally, the STG animation demonstrated the benefits in reduced initialisation offered by the simple `leton` mapped expression. However, a question does remain as to the scalability and general applicability of mapped expressions.

9.5 Kelly's Skeletons

This section concentrates on three particular skeletons used by Darlington, Field, Harrison, Kelly and others [1993]: `pipe`, `farm`, and `dc` (see section 2.4.3 for a general overview of algorithmic skeletons). These provide a representative sample of the current skeletal population, and Kelly's work is one of the few to attempt to integrate skeletons into a non-strict language. The following Haskell definitions are used to provide an informal sequential semantics of the three skeletons:

Haskell

```
pipe :: [a -> a] -> (a -> a)
pipe = foldr1 (.)

farm :: (a -> b -> c) -> a -> ([b] -> [c])
farm f env = map . (f env)

dc :: (a -> Bool) -> (a -> b) -> (a -> (a, a)]) -> ((b, b) -> b) -> a -> b
dc endtest endval split combine x
  | endtest x = endval x
  | otherwise = let (l, r) = split x
                in combine (dc endtest endval split combine l)
                           (dc endtest endval split combine r)
```

9.5.1 Static semantics

Following the guidelines laid down in section 5.2.1, the first attempt at incorporating skeletons into the STG′ language is shown below:

exp	$\longrightarrow$	*skeleton* $\mid \cdots$	skeletal expression
skeleton	$\longrightarrow$	`farm` var_{fun} exp	processor farm
	$\mid$	`dc` $var_{end?}$ var_{single} var_{divide} $var_{combine}$ exp	divide and conquer
	$\mid$	`pipe` $var_{stage_1} \cdots var_{stage_n}$ exp	process parallelism

Notice that the `farm` skeleton, unlike Kelly's version, does not need to take the `env` parameter as the STG′ language's sharing mechanism can be used to define an appropriate replacement function:

STG′ code

```
let { f' = [env] \r [x] -> f env x; } in farm f' xs
```

Also, the `dc` skeleton looks suspiciously complex, and can, in fact, be implemented as successfully using the mapping expressions from the previous case study – see figure 9.50. The benefit of maintaining this skeletal expression cannot be justified at the intermediate level – it is merely an artifact to be used during the initial phases of the compilation process. Having decided that it it may not be necessary to represent all skeletons within the intermediate language, the `farm` expression warrants further investigation. The denotational semantics is presented in figure 9.51, and is, in effect, a highly strict version of the `map` function. Again, it is easy to generate para-functional code to duplicate this behaviour:

STG′ code

```
divacon = [] \r [endtest endval split combine] ->
  letrec {
    f = [split combine endtest endval] \r [x] -> case endtest x of
      {
       True  -> endval ~x;
       False -> case split x of {
                 Tup2 l r -> let# here = self# in
                             let#  left_neighbour = minusInt# [here, 1#] in
                             let# right_neighbour =  plusInt# [here, 1#] in
                             leton  left_neighbour left  = f l in
                             leton right_neighbour right = f r in
                             combine left right;
                                };
      };
    } in f;
```

Figure 9.50: A para-functional STG′ replacement for the `dc` skeleton

$$
\begin{array}{lclcl}
\mathcal{S}keleton\llbracket \texttt{farm}\ var_{fun}\ exp\rrbracket\ \rho & = & let\ function & = & \rho\ var_{fun} \\
 & & \quad function' & = & compose\ \xi_{\infty}^{(\chi\ \pi_1\ldots\pi_n)} function \\
 & & \quad arguments & = & \mathcal{E}\llbracket exp\rrbracket\ \rho \\
 & & in\ \xi_{\infty}^{(\texttt{List}\ \pi)}\ (map\ function\ arguments) & &
\end{array}
$$

$$\xi_{\infty}^{(\chi\ \pi_1\ldots\pi_n)} \quad :: \quad \mathbf{Val} \rightarrow \mathbf{Val}$$

$$
\xi_{\infty}^{(\chi\ \pi_1\ldots\pi_n)}\ \epsilon = case\ \epsilon\ of\ \begin{array}{lcl} [\,]\ \bot & \rightarrow & \bot \\ [\,]\ \epsilon & \rightarrow & \epsilon \end{array}
$$

$$\xi_{\infty}^{(\texttt{List}\ \pi)} \quad :: \quad \mathbf{Val} \rightarrow \mathbf{Val}$$

$$
\xi_{\infty}^{(\texttt{List}\ \pi)}\ \epsilon = case\ \epsilon\ of\ \begin{array}{lcl} [\,]\ \bot & \rightarrow & \bot \\ [\,]\ \langle\texttt{Nil}\rangle & \rightarrow & \langle\texttt{Nil}\rangle \\ [\,]\ \langle\texttt{Cons}, x, xs\rangle & \rightarrow & \langle\texttt{Cons}, x, \xi_{\infty}^{(\texttt{List}\ \pi)}\ xs\rangle \end{array}
$$

Figure 9.51: Denotational semantics of the `farm` skeleton

STG′ code

```
farm = [] \r [f xs] -> case xs of
  {
    Nil       -> Nil [];
    Cons y ys -> let#  here = self# in
                 let#  neighbour = plusInt# [here, 1#] in
                 leton neighbour ys' = farm f ys in
                 letstrict y' = f y in
                 Cons [y', ys'];
  };
```

Finally, the following transformation shows how pipe expressions can also be removed from the intermediate language:

$$\begin{array}{l} \texttt{pipe}\ var_{stage_1} \cdots var_{stage_n}\ exp \implies \texttt{farm}\ fun\ exp \\ \quad \texttt{where}\ fun = [\,]\ \texttt{r}\ [var_{arg_1}] \rightarrow fun'\ var_{arg_1} \\ \qquad\qquad fun' = [\,]\ \texttt{r}\ [var_x] \rightarrow compose_n\ var_{stage_1} \cdots var_{stage_n}\ var_x \end{array}$$

9.5.2 Assessment

Perhaps surprisingly, it turns out that skeletal expressions should not form part of a parallel functional intermediate language. They are simply high-level constructs which provide the programmer with continent abstractions for developing parallel algorithms. Any skeleton-based compiler will certainly manipulate skeletal expressions, but they will have been reduce to more basic operations before the operational semantics will have to be considered.

9.6 Summary

This chapter has presented four case studies of the prototyping framework:

Mattson's speculative evaluation The first case study dealt with development of low-level synchronisation and scheduling constructs for a GMSV architecture. The performance results showed the expected near-ideal speedups associated with shared-memory architectures. However, only the RISC animation exhibited the second-order effects introduced by the necessary locking operations.

GUM Haskell This study built upon the work of the previous investigation, and extended it to DMMP architectures. The primary focus was the use of explicit communication to implement load-balancing and resource-sharing mechanisms. UML interaction diagrams, therefore, became an essential part of the development process. The performance results again closely agreed with those observed in real implementations. However, the STG animation was only capable of simulating relatively small benchmark problems. It was suggested that this should not be a cause for concern: larger problem sizes tend to exhibit better performance on parallel architectures, therefore hiding any inefficiencies of the operational model. The RISC animation was only used to verify that the STG animation was producing credible performance estimates.

Para-functional Haskell This extended the GUM model to include explicit mapped expressions. It was shown that such annotations can reduce the initialisation phase of the evaluation, and therefore lead to moderate improvements in performance. However, the added burden placed upon the programmer, combined with the inability to animate larger problems, raised some doubt as to the general applicability of mapped expressions. Finally, Hudak's execution-tree semantics was identified as a potentially useful tool to aid the parallel functional programmer.

Kelly's skeletons The final case study considered the representation of skeletal operators in the context of a parallel functional intermediate language. It was decided that the role of such expressions should be limited to the initial phases of the compilation process and should not interfere with the operational model of the intermediate language.

Chapter 10

Summary, evaluation and further work

10.1 Introduction

This chapter will begin with a brief summary of the work which has been presented in this thesis (section 10.2), followed by an evaluation of this work in section 10.3. Finally, in section 10.4, the limitations of this work are briefly discussed, and further potential avenues of exploration are suggested.

10.2 Summary

The contributions of this thesis are as follows:

- the presentation of a framework for rapidly prototyping parallel functional intermediate languages, driven by the development of semantic models for the three phases of a traditional compiler – the source, intermediate, and target languages.

- a number of prescriptive methods for animating denotational semantics, Hindley–Milner type-inference algorithms, and state-transition systems in the functional programming language Haskell.

- the development and informal validation of a static semantics for the sequential STG′ language. In addition, the development of an execution-tree semantics [Hudak, 1986] for a para-functional variant of the STG′ language.

- the use of a state-transition system to model multiprocessor systems, using shared-memory and/or message passing as the primary communication mechanisms.

- a state-transition model of an optimising compilation system for the STG′ language, closely based on the operational model.

10.3 Evaluation of the prototyping framework

This evaluation starts by comparing and contrasting the prototyping framework with the related techniques reviewed in section 2.5.3. It then goes on to attempt to evaluate the success of the framework both as a prototyping tool for parallel functional intermediate languages and as an animation system for static and operational semantics.

10.3.1 Comparison with other relevant work

The Haskell approach

pH [Nikhil et al., 1995, section 1, page 1], a Haskell derivative extended to include explicit parallelism, has as one of its goals:

> "To share infrastructure (compilers, systems, application programs), and to facilitate interesting research topics, such as comparing lazy evaluation *vs.* lenient evaluation..."

However, by necessity, the resulting compilers are written primarily for speed and efficiency, possibly at the expense of clarity – based on personal experience, this is certainly true of GHC! Moreover, the system will be sufficiently complex that familiarisation and development will take a significant amount of time.

Direct implementations

A number of compilation systems, based upon explicitly parallel versions of the STG language, have been developed, and each of the extension techniques described in section 2.4 is represented: Hill's data-parallel Haskell [Hill, 1994] introduces the POD (parallel object with arbitrary dimensions) data type and associated primitive functions; Chakravarty's JUMP* machine [Chakravarty, 1994] extends the *exp* rule with the *letpar* construct; Hwang and Rushall [1992] alter the semantics of the *case* expression in their ν-STG machine (this corresponds to the *if* construct in the language we have presented); and Hammond, Mattson Jr. and Peyton Jones [1994] add *par* and *seq* as primitive functions. The primary aim of these systems has been to demonstrate the usefulness of the implementors favourite language extension or run-time algorithm. In each case, little justification is provided as to why a particular approach was taken, and no real effort has gone into comparing and contrasting the features offered by each of these systems.

The approach outlined in this thesis enables the rapid prototyping a wide range of languages, which, in turn, allows one to examine these very issues. The case studies from chapter 9 demonstrated how a system could be developed incrementally, allowing competing approaches to be evaluated fairly. Furthermore, the generation of the semantic descriptions will serve as excellent documentation of the various design decisions and experiments.

Prototyping versus simulation

To date, simulation has been widely used by the community as a substitute for prototyping [Joy and Axford, 1992; Bennett, 1993; Hammond, Loidl and Partridge, 1995a]. However, as acknowledged by Deschner [1990, section 1, page 227], such systems tend to allow only a limited design to be explored:

> "Although initially the system is only capable of simulating conservative parallelism, with major adjustment it could also be used to analyse speculative evaluation strategies."

The work of Hammond, Loidl and Partridge [1995b] bears the closest resemblance to this work. They use an accurate multi-architecture simulation system, based on the Glasgow Haskell compiler, to study the effects of language annotations [Burton, 1984] on task granularity. Their overall aim is to develop heuristics for use with an automatically parallelising compiler. This work, on the other hand, encompasses the derivation of parallelism using both implicit and explicit techniques. In addition, their intermediate language identifies parallelism only through the use of primitive functions, and ignores the many alternatives.

10.3.2 Evaluation of the success of the prototyping framework

The four case studies from chapter 9 demonstrated the utility and sophistication of the framework. Based upon the experience gained during these case studies, it is not unreasonable to claim that the prototyping framework could be effectively applied to re-engineering any of the current crop of parallel implementations. Furthermore, the work of chapters 5 and 6 showed how the semantics models could cope with advances both in terms of language idioms and implementation techniques.

10.3.3 Animating static and operational semantics

As outlined by Goodman [1995, section 7.3.3], there are two distinct approaches to evaluating the success of an animation system: firstly, rating the framework against a number of theoretical concerns, including coverage, efficiency, and sophistication; and, secondly, evaluating its practical success as a method of software development. However, Goodman's assessment of the difficulties in implementing the latter approach is sufficiently complete (and relevant) for the purpose of this evaluation that only the former approach is pursued here.

Goodman uses the following eight concerns to rate an animation system: coverage, efficiency, sophistication, interactivity, transparency, operational equivalence, usability and utility. The animations used by the prototyping framework are scored as follows:

- *Coverage.* Good, as Haskell's semantics are very close to those used to model the static and operational semantics.

- *Efficiency.* The primary aim of the animation techniques is to maintain a close correspondence between the semantic descriptions and the Haskell code. While the efficiency of the resulting programs could be improved, this would interfere with the method's operational equivalence.

- *Sophistication.* Good. Examples include the case studies from chapter 9, and the work of Booth, Bruce and Ben-Dyke [1996] on the animation of an imperative parallel object-oriented language.

- *Interactivity.* The animation of the static semantics results in a non-interactive program, and must therefore score poorly. However, the state-transition animations are highly interactive, and the overall score can therefore deemed to be fair.

- *Transparency.* Reasonable. Haskell's pattern-matching semantics, combined with the libraries developed during the case studies, simplify the conversion process.

- *Operational equivalence.* As for transparency.

- *Usability.* Good. Few problems were encountered during the animation of the various case studies.

- *Utility.* Good. The framework used for the state-transition animations has been used successfully on a range of applications including the STG machine, a hybrid RISC architecture, and a compilation system.

10.4 Limitations and further work

The following important issues have been largely ignored throughout this thesis, but are deserving of further attention:

- the development of accurate yet concise models of the behaviour of shared-memory systems with respect to locking and concurrency control (see section 6.2.4).
- the development of one or more domain-specific languages for simplifying the construction of the various semantic models (with the possibility of automatically animating the results). Relevant research includes Navel [Michaelson, 1993] and Actress [Brown, Moura and Watt, 1992].
- the verification of the presented semantics, and the development of a prescriptive approach to proving the correctness of the various rule sets.
- the expansion of the framework to include imperative languages, or using a different approach to type inference. (An initial feasibility study has already been carried out [Booth, Bruce and Ben-Dyke, 1996].)

Appendix A

On the design of parallel functional intermediate languages

This paper [Ben-Dyke and Axford, 1995] was originally presented at HiPC '95, the International Conference on High Performance Computing [Sahni, Prasanna and Bhatkar, 1995]. The contribution of the two authors was as follows: Andy Ben-Dyke, 80%; Tom Axford, 20%.

A.1 Introduction

A.2 Defining the language

A.3 Developing the parser

A.4 Code generation

A.4.1 Operational semantics

A.4.2 Compilation rules

A.4.3 Architecture simulator

A.5 Relationship to other work

A.6 Concluding remarks

Appendix B

Example STG′ programs

This section presents a number of example STG′ programs, as generated by the Glasgow Haskell compiler (see section C.2 for further qualification). The Haskell source code is primarily taken from either the Haskell standard prelude [Hudak et al., 1992, appendix A], or the imaginary subset of the `nofib` benchmark suite (see appendix C).

Section B.1 looks at some of the prelude operations used to support integers, booleans, and lists. Three `nofib` programs, `fib`, `primes`, and `queens`, are then presented in sections B.2 through B.4. Finally, a solution to Hamming's problem, as developed by Hudak and Anderson [1988, section 3], is converted into STG′ code.

B.1 Prelude operations

This section looks at the STG′ definitions needed to support the three main data types of the Haskell language, namely integers, booleans, and lists. Where applicable, the equivalent Haskell code is also included. All of the STG′ bindings have been taken directly from the library of test routines used by the prototyping system (see section 3.4).

B.1.1 Integers

Primitive wrappers

The following data decleration and selected associated operations make up the interface for the primitive integer type, `Int#`:

STG′ code

```
data Int = Int Int#;

zero = [] \r []  ->  Int [0#];
one  = [] \r []  ->  Int [1#];

const.Int.+ = [] \r [x y] -> case x of
 { Int x' -> case y of { Int y' -> let# xy = plusInt# [x', y'] in Int [xy] ; };
 } ;

const.Int.> = [] \r [x y] ->
 case x of { Int x' -> case y of { Int y' -> gtInt# [x', y'] ; } ; } ;
```

Obviously, no Haskell equivalent exist for any of these definitions.

Quotients and signs

The STG′ definitions given below are based on `Int#`-specialised versions of the following prelude functions:

Haskell

```
quotRem :: Int -> Int -> (Int, Int)
quotRem n d = (n `quot` d, n `rem` d)

signum :: Int -> Int
signum x | x == 0    =  0
         | x > 0     =  1
         | otherwise = -1
```

`const.Int.quotRem` is a straightforward conversion of `quotRem`, but `const.Int.signum` makes use of the wrapper/worker optimisation [Peyton Jones and Launchbury, 1991, section 5.1]:

STG′ code

```
const.Int.quotRem = [] \r [n d] ->
 let { q = [n d] \u [] -> const.Int.quot n d;
       r = [n d] \u [] -> const.Int.rem  n d; } in Tup2 [q, r];

const.Int.signum = [] \r [x] -> case x of {Int x' -> const.Int.signum.wrk x';};

const.Int.signum.wrk = [] \r [x] -> case x of
 {0#  -> Int [0#];
  _   -> case  gtInt# [x, 0#] of { True  -> Int [  1#]; False -> Int [ -1#]; }
 };
```

B.1.2 Booleans

Haskell

```
data Bool = False | True

otherwise :: Bool
otherwise  = True

(&&) :: Bool -> Bool -> Bool
(&&)    False   x     = False
(&&)    True    x     = x

not :: Bool -> Bool
not    True  = False
not    False = True
```

Again, after the pattern-matching syntactic sugar is removed, the STG′ code is similar to the Haskell versions:

STG′ code

```
data Bool = True | False;

otherwise = [] \r [] -> True [];

&&  = [] \r [x y] -> case x of { False -> False [] ; True  -> y ; } ;

not = [] \r [x]   -> case x of { True  -> False [] ; False -> True  [] ; };
```

B.1.3 Lists

Rather than introducing special syntactic support, the following STG′ declaration is used to define the `List` algebraic data type:

STG′ code

```
data List a = Cons a (List a) | Nil;
```

The following sections look at some of Haskell's `PreludeList` [Hudak et al., 1992, section A.5, pages 106–114] functions.

Nill, null, head, and tail

Haskell

```
nil :: [a]
nil = []

null :: [a] -> Bool
null []      = True
null (_:_)   = False

head :: [a] -> a
head (x:_)   = x

tail :: [a] -> [a]
tail (_:xs)  = xs
```

STG′ code

```
nil = [] \r [] -> Nil [];

null = [] \r [xss] -> case  xss  of { Nil -> True []; Cons x xs -> False []; };

head = [] \r [xss] -> case  xss  of { Cons x xs -> x  ; Nil -> error# [] ; };

tail = [] \r [xss] -> case  xss  of { Cons x xs -> xs ; Nil -> error# [] ; };
```

Append

Haskell

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs++ys)
```

STG′ code

```
++ = [] \r [xss yss] -> case xss of
 { Nil       -> yss ;
   Cons x xs -> let { xs' = [yss xs] \u [] -> ++ xs yss; } in  Cons [x, xs'];
 };
```

Length

Rather than use the `foldl`-based Haskell version, the more traditional version is used:

Haskell

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

STG′ code

```
length = [] \r [xss] -> case xss of
 { Nil       -> Int [0#] ;
   Cons x xs -> case length xs of { Int l -> let# l’ = plusInt# [1#, l]
                                             in Int [l’] ; };
 };
```

Map

Haskell

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

STG′ code

```
map = [] \r [f xss] -> case xss of
 { Nil        -> Nil [] ;
   Cons x xs  -> let { x’ = [f  x] \u [] -> f x ;
                       xs’ = [f xs] \u [] -> map f xs ; } in Cons [x’, xs’] ;
 };
```

Foldl

Haskell

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

STG′ code

```
foldl = [] \r [f z xss] -> case xss of
 { Nil       -> z;
   Cons x xs -> let { x’ = [] \u [] -> f z x; } in foldl f x’ xs;
 };
```

Filter

Again, the `foldr`-based version is ignored in favour of the traditional definition:

Haskell

```
filter           :: (a -> Bool) -> [a] -> [a]
filter p []                = []
filter p (x:xs) | p x      = x : (filter xs)
                | otherwise =     filter xs
```

STG′ code

```
filter = [] \r [p xss] -> case xss of
 { Nil       -> Nil [] ;
   Cons x xs -> case p x of { True  -> let {xs’ = [p xs] \u [] -> filter p xs;}
                                       in Cons [x, xs’];
                              False -> filter p xs;
                            };
 };
```

B.2 Generating Fibonacci numbers

Haskell

```
fib :: Int -> Int
fib n = if n <= 1 then 1 else fib (n-1) + fib (n-2) + 1
```

Unoptimised version

STG′ code

```
fib = [] \r [n] -> case const.Int.<= n one of
 { True  -> one ;
   False -> let { sum_2_fibs = [n] \u [] ->
                   let { fib_n_less_2 = [n] \u [] ->
                           let { n_less_2 = [n] \u [] -> const.Int.- n two; }
                           in fib n_less_2;
                         fib_n_less_1 = [n] \u [] ->
                           let { n_less_1 = [n] \u [] -> const.Int.- n one; }
                           in  fib n_less_1; }
                   in  const.Int.+ fib_n_less_1 fib_n_less_2; }
            in const.Int.+ sum_2_fibs one;
 };
```

Optimised version

STG′ code

```
fib = [] \r [n] -> case  n  of { Int n' -> fib.wrk n'; };
fib.wrk = [] \r [n'] -> case leInt# [n', 1#] of
 { True  -> Int [1#];
   False -> let# n'_less_1 = minusInt# [n', 1#] in
            case fib.wrk n'_less_1 of { Int fib_n'_less_1 ->
            let# n'_less_2 = minusInt# [n', 2#] in
            case fib.wrk n'_less_2 of { Int fib_n'_less_2 ->
            let# sum_2_fibs' = plusInt# [fib_n'_less_1, fib_n'_less_2] in
            let# result = plusInt# [sum_2_fibs', 1#]
            in Int [result];          }; };
 };
```

B.3 Generating prime numbers – the sieve of Eratoshenes

Haskell

```
test :: Int -> Int
test a = let primes = map head (iterate the_filter (iterate succ 2))
         in  primes !! a

the_filter :: [Int] -> [Int]
the_filter (n:ns) = filter (isdivs n) ns

isdivs :: Int  -> Int -> Bool
isdivs n x = mod x n /= 0

succ :: Int -> Int
succ x = x + 1
```

Unoptimised version

STG′ code

```
test = [] \r [a] -> let { primes = [] \u [] ->
                            let { xs = [] \u [] ->
                                   let { from_2 = [] \u [] -> iterate succ two;}
                                   in  iterate the_filter from_2; }
                            in map head xs; }
                     in !! primes a;

the_filter = [] \r [nss] -> case nss of { Cons n ns ->
 let { isdivs_n =  [n] \r [x] -> isdivs n x; } in filter isdivs_n ns; };

isdivs = [] \r [n x] -> let { mod_x_n =  [n x] \u [] -> const.Int.mod x n; }
                        in  const.Int./= mod_x_n zero;
succ = [] \r [x] -> const.Int.+ x one;
```

Optimised version

STG′ code

```
test = [] \r [a] -> case a of { Int a' -> test.wrk a'; };
test.wrk = [] \r [a'] -> let { from_2 =  [] \u [] ->  iterate succ two; } in
                        letstrict forced_xs = iterate the_filter from_2  in
                        letstrict forced_primes = map head forced_xs
                        in !!.wrk forced_primes a';
the_filter = [] \r [nss] -> case nss of { Cons n ns ->
 let { isdivs_n = [n] \r [x] -> case n of { Int n' -> case x of { Int x' ->
                               isdivs.wrk n' x'; }; }; }
 in filter isdivs_n ns; };

isdivs = [] \r [n x] ->
 case n of { Int n' -> case x of { Int x' -> isdivs.wrk n' x'; }; };

succ = [] \r [x] ->
 case x of { Int x' -> let# succ_x = plusInt# [x', 1#] in Int [succ_x]; };

isdivs.wrk = [] \r [n' x'] -> case const.Int.mod.wrk x' n' of { Int mod' ->
                              case mod' of { 0# -> False [];
                                             _  -> True  [] };
                                          };
```

B.4 The queens problem

Haskell

```
nsoln :: Int -> Int
nsoln nq = length (gen nq nq)

safe :: Int -> Int -> [Int] -> Bool
safe x d []    = True
safe x d (q:l) = x /= q && x /= q+d && x /= q-d && safe x (d+1) l

gen :: Int -> Int -> [[Int]]
gen nq 0 = [[]]
gen nq n = [ (q:b) | b <- gen nq (n-1), q <- [1..nq], safe q 1 b]
```

Unoptimised version

STG′ code

```
nsoln     = [] \r [nq] -> let { solutions =  [nq] \u [] ->  gen nq nq;
                          } in  length solutions;
```

STG′ code

```
safe = [] \r [x d ds] -> case ds of
 { Nil      -> True [];
   Cons q l -> let
    { c1 = [x d q l] \u [] -> let
           { c2 = [x d q l] \u [] -> let
                  { c3 = [x d l] \u [] -> let
                         { d_plus_1 = [d] \u [] -> const.Int.+ d one;
                         } in safe x d_plus_1 l;
                  } in let { c4 = [x d q] \u [] -> let
                                  { q_less_d = [d q] \u [] -> const.Int.- q d;
                                  } in  const.Int./= x q_less_d;
                           } in && c4 c3;
           } in let { c5 = [x d q] \u [] -> let
                           { q_plus_d = [d q] \u [] -> const.Int.+ q d;
                           } in  const.Int./= x q_plus_d;
                    } in  && c5 c2;
    } in let { c6 = [x q] \u [] ->  const.Int./= x q;
             } in  && c6 c1;
 };
```

STG′ code

```
zero_soln = [] \r [] ->  Cons [nil,  nil];

gen = [] \r [nq ds] -> case ds  of { Int ds’ -> case ds’ of
 { 0# -> zero_soln ;
   _  -> letrec { f = [f nq] \u [] ->  let { one_to_nq = [nq] \u [] ->
                                             const.Int.enumFromTo one nq; } in
                                       let { g = ** see below ** } in g ;}  in
         let    { d = [ds nq] \u [] -> let { ds_less_1 = [ds] \u [] ->
                                                          const.Int.- ds one; }
                                       in  gen nq ds_less_1; }
         in f d };
 };
```

To improve readability, the definition of g, given below, was removed from the body of gen.

STG′ code

```
g = [f one_to_nq nq] \r [xss] -> case xss of
 { Nil -> Nil [];
   Cons x xs ->
    letrec { h = [f x h nq xs] \u [] ->
                   let { a = [f xs] \u [] -> f xs; } in
                   let { i = [x h nq a] \r [yss] -> case yss of
                              { Nil -> a ;
                                Cons y ys -> case safe y one x of
                                  { True -> let { b = [] \r [] -> Cons [y, x];
                                                  c = [ys h] \u [] -> h ys; }
                                            in Cons [b, c];
                                    False -> h ys;
                                  };
                              }; }
                   in i ; }
    in h one_to_nq;
  };
```

Optimised version

STG′ code

```
nsoln = [] \r [nq] -> case nq of { Int nq' -> nsoln.wrk nq'; };

nsoln.wrk = [] \r [nq'] -> let { nq = [] \r [] ->  Int [nq']; } in
                           letstrict solutions = gen.wrk nq nq'
                           in  length solutions;
```

STG′ code

```
safe = [] \r [x d ds] -> case ds of
 { Nil      -> True [];
   Cons q l -> case x of   { Int x' -> case q of { Int q' ->
               case neInt# [x', q'] of
               { False -> False [];
                 True  -> case d of { Int d' ->
                          let# q_plus_d = plusInt# [q', d'] in
                          case neInt# [x', q_plus_d] of
                          {
                          False -> False [];
                          True  -> let# q_less_d = minusInt# [q', d'] in
                                   case neInt# [x', q_less_d] of
                                   {
                                   False -> False [];
                                   True  -> let# d_plus_1'= plusInt# [d',1#] in
                                            let{ d_plus_1 = [] \r [] ->
                                                              Int [d_plus_1'];
                                                } in safe x d_plus_1 l;
                                   };
                          }; };
               }; }; };
 };
```

STG′ code

```
gen = [] \r [nq ds] -> case ds of { Int ds' -> gen.wrk nq ds'; };

gen.wrk = [] \r [nq upk] -> case upk of
 { 0# -> zero_soln ;
   _  -> let# upk_less_1 = minusInt# [upk,  1#]   in
         letstrict bs = gen.wrk nq upk_less_1 in
         let { one_to_nq = [nq] \u [] -> const.Int.enumFromTo one nq; }
         in gen_comprehension nq one_to_nq bs
 };
gen_comprehension = [] \r [nq one_to_nq dss] -> case dss of
 { Nil       -> Nil [];
   Cons d ds -> let { a = [nq one_to_nq ds] \u [] ->
                              gen_comprehension nq one_to_nq ds; }
                in  g a d nq one_to_nq;
 };
g = [] \r [a d nq one_to_nq] -> case  one_to_nq of
 { Nil -> a ;
   Cons x xs -> case safe x one d of
    { False -> g a d nq xs;
      True  -> let { b = []          \r [] -> Cons [x,  d];
                     c = [a d xs nq] \u [] -> g a d nq  xs; } in Cons [b, c];
    };
 };
```

B.5 Hamming's problem

The following program is directly based on that presented by Hudak and Anderson [1988, section 3].

Haskell

```
hamming :: [Int] -> [Int]
hamming primes = 1 : (foldl f [] primes)
  where f xs p = let h = merge (scale p (1 : h)) xs in  h

merge :: [Int] -> [Int] -> [Int]
merge []         bss        = bss
merge ass        []         = ass
merge ass@(a:as) bss@(b:bs) | a < b      = a : (merge as  bss)
                            | otherwise  = b : (merge ass bs)

scale :: Int -> [Int] -> [Int]
scale p xs = map (* p) xs

isdivs :: Int  -> Int -> Bool
isdivs n x = mod x n /= 0

the_filter :: [Int] -> [Int]
the_filter (n:ns) = filter (isdivs n) ns

test :: Int -> Int -> Int
test cut_off no_primes = length sequence
  where sequence   = takeWhile (< cut_off) (hamming few_primes)
        primes     = map head (iterate the_filter (iterate succ 2))
        few_primes = take no_primes primes
```

Unoptimised version

STG′ code

```
hamming = [] \r [primes] -> let { as =  [primes] \u [] -> foldl f Nil primes; }
                            in Cons [one, as];

f = [] \r [xs p] ->
 letrec { h = [h xs p] \u [] ->
                let { a = [h p] \u [] -> let { xs = [] \r [] -> Cons [one, h];}
                                         in scale p xs;}
                in  merge a xs; }
 in  h ;

merge = [] \r [ass bss] -> case ass of
 { Nil       -> bss ;
   Cons a as -> case bss of
    { Nil       -> Cons [a, as];
      Cons b bs -> case const.Int.< a b of
        { True  -> let { xs = [bss as] \u [] -> merge as bss;} in Cons [a, xs];
          False -> case otherwise of
                     { True   -> let { ys = [ass bs] \u [] -> merge ass bs; }
                                 in Cons [b, ys];
                        False -> error ;
                     };
        };
    };
 };

scale = [] \r [p xs] -> let {g =  [p] \r [a] -> const.Int.* a p;} in map g xs;

isdivs = [] \r [n x] -> let { a =  [n x] \u [] -> const.Int.mod x n; }
                        in const.Int./= a zero;

the_filter = [] \r [ds] -> case  ds  of
 { Nil       -> error ;
   Cons n ns -> let { a =  [n] \r [x] -> isdivs n x; } in filter a ns;
 };
```

STG′ code

```
test = [] \r [cut_off no_primes] ->
 let { primes = [] \u [] ->
         let { as = [] \u [] -> let { from_two = [] \u [] -> iterate succ two;}
                                in iterate the_filter from_two; }
         in map head as; } in
 let { few_primes = [primes no_primes] \u [] -> take no_primes primes; } in
 let { sequence   = [few_primes cut_off] \u [] ->
                      let { bs = [few_primes] \u [] -> hamming few_primes;
                            p  = [cut_off] \r [x] -> const.Int.< x cut_off; }
                      in  takeWhile p bs; }
 in  length sequence;
```

Optimised version

STG′ code

```
hamming = [] \r [primes] -> let { as =  [primes] \u [] -> foldl f Nil primes; }
                            in Cons [one,  as];
f = [] \r [xs p] -> letrec { h = [p h xs] \u [] ->
                                   let { as = [] \r [] -> Cons [one,  h]; } in
                                   letstrict ys = scale p as
                                   in  merge ys xs; }
                    in  h ;
merge = [] \r [ass bss] -> case ass of
 { Nil       -> bss ;
   Cons a as -> case bss of
    { Nil       -> Cons [a, as];
      Cons b bs -> case a of { Int a' -> case b of { Int b' ->
                   case ltInt# [a',  b'] of
                    { True  -> let { cs = [bss as] \u [] -> merge as bss; }
                               in Cons [a, cs];
                      False -> let { cs = [ass bs] \u [] -> merge ass bs; }
                               in  Cons [b, cs];
                    }; }; };
    };
 };
scale = [] \r [p xs] ->
 let { g = [p] \r [a] -> case a of { Int a' -> case p of { Int p' ->
                         let# a_times_p = timesInt# [a',  p']
                         in Int [a_times_p]; }; }; }
 in  map g xs;
isdivs = [] \r [n x] -> case n of { Int n' -> case x of { Int x' ->
                        isdivs.wrk n' x'; }; };
isdivs.wrk = [] \r [n' x'] -> case const.Int.mod.wrk x' n' of { Int mod_x_n ->
                              case mod_x_n of { 0# -> False [];
                                                _  -> True  []
                                              }; };
the_filter = [] \r [nss] -> case nss of
 { Nil       -> error ;
   Cons n ns -> let { is_divs_n = [n] \r [x] -> case n of { Int n' ->
                                                case x of { Int x' ->
                                                isdivs.wrk n' x'; }; }; }
                in  filter is_divs_n ns;
 };
```

STG′ code

```
test = [] \r [cut_off no_primes] ->
 let { few_primes = [no_primes] \s [] ->
        case no_primes of { Int no_primes' ->
        let { primes = [] \u [] -> let {from_two = [] \u [] ->
                                                   iterate succ two; } in
                                   letstrict  as = iterate the_filter from_two
                                   in map head as; }
        in  take.Int.!.wrk no_primes' primes; }; } in
 letstrict bs = hamming few_primes in
 let { i = [cut_off] \r [x] -> case x       of { Int x' ->
                               case cut_off of { Int cut_off' ->
                               ltInt# [x', cut_off']; }; }; } in
 letstrict sequence = takeWhile i bs
 in length sequence;
```

Appendix C

The STG′ language and the nofib benchmark suite

When developing a compiler for a particular language it is often helpful to have a feel for the types of usage of the constructs. To provide this empirical information for the STG′ language, elements of the `nofib` benchmark suite were compiled to STG′ code and statically analysed (the dynamic aspects of the benchmark suite have been explored by Santos [1995].) The collected data includes the distribution: arguments and free variables, algebraic data types, `case` and `letrec` expressions etc.

Sections C.1 and C.2 look at the `nofib` benchmark suite and the gathering of the data, sections C.3 to C.6 present the results, and section C.7 points out the limitations of the method.

C.1 The nofib benchmark suite

The `nofib` benchmark suite [Partain, 1993] is a publically available collection of small to large Haskell programs, split into three categories: the *imaginary* subset, containing toy programs useful for testing the correctness of a compilation system but of no real benchmarking worth; the *real* subset, made up of programs written to perform a useful task; and the *spectral* subset, which contains everything else, and includes the benchmark programs used by Hartel [1994]. For the purpose of this study, only the *real* subset of the suite has been used, a brief overview of which is given in table C.1.

C.2 Gathering the data

In order to generate the required statistics, version 0.23 of the Glasgow Haskell compiler was modified to bring its concrete syntax (of the STG language) into line with that used within this report, and the `-flet-no-escape` option removed from the optimisation package. Each of the benchmarks were then compiled to STG′ code (`ghc -O -ddump-stg`) and the resulting programs analysed using a combination of Unix shell scripts and Emacs Lisp macros.

program	description	STG′ lines
`anna`	strictness analyser	53 220
`bspt`	BSP-tree modeller	17 288
`compress`	text compression	2 224
`compress2`	text compression	2 026
`ebnf2ps`	BNF grammar to postscript utility	23 311
`fluid`	fluid-dynamics program	20 616
`fulsom`	solid modelling	13 223
`gg`	graphs from GRIP statistics	12 449
`grep`	simple version of the Unix command	861
`hidden`	hidden-line removal	7 415
`hpg`	Haskell program generator	7 485
`infer`	Hindley-Milner type inference	3 090
`lift`	fully-lazy lambda lifter	4 707
`maillist`	mailing-list generator	596
`mkhprog`	Haskell program skeletons	1 759
`parser`	partial Haskell parser	14 537
`pic`	particle in cell	5 285
`prolog`	"mini-Prolog" interpreter	2 606
`reptile`	Escher tiling program	12 527
`rsa`	RSA encryption/decryption	1 016
`symalg`	variable-precision calculator	11 114
`veritas`	theorem-prover	32 551

Table C.1: The *real* subset of the `nofib` benchmark suite

C.3 Algebraic data types

The number of constructors per data type for the 155 non-prelude definitions is distributed as follows:

constructors	1	2	3	4	5	6–10	11–20	21+	max.
absolute	54	26	17	19	11	18	8	2	36
percentage	34·8	16·8	11·9	12·3	7·1	11·6	5·2	1·3	

For the 21 prelude types used by the benchmark programs (*Array*, *Assoc*, *Bool*, *IOError*, *List*, *Ratio*, *Request*, *Response*, *Tup*0, *Tup*2–*Tup*10, *Tup*12, and *Tup*19) and the Glasgow specific *_CMP_TAG*, the distribution is as follows:

constructors	1	2	3	4	5	6–10	11–20	21+	max.
absolute	15	2	1	0	2	0	1	0	18
percentage	71·4	9·5	4·8	0	9·5	0	4·8	0	

The lifted versions of the primitive types, such as *Int* and *Char*, are not included in this data.

The distribution of the number of arguments of the 600 non-prelude constructor is as follows:

arguments	0	1	2	3	4	5	6–10	11–20	max.
absolute	233	203	85	42	15	10	9	3	20
percentage	38·8	33·8	14·2	7·0	1·9	1·3	1·1	0·4	

The distribution for the 50 prelude constructors is:

arguments	0	1	2	3	4	5	6–10	11–20	max.
absolute	10	18	12	1	1	1	5	2	19
percentage	20·0	26·0	24·0	2·0	2·0	2·0	10·0	4·0	

C.4 Bindings and `let(rec)` expressions

There are a total of 15 878 global definitions, 17 354 `let` bindings, and 127 `letrec` bindings, in addition to the 2 245 `letstrict` expressions and 2 125 `let#` expressions. The relative mixture of functions, constructors and thunks is shown below:

closure	constructor	function	thunk	other	total
absolute	13 235	6 571	12 861	692	33 359
percentage	39·7	19·7	38·6	2·1	

The *other* category is primarily niladic functions.

Of the 127 `letrec` expressions, the distribution of the number of bindings is as follows:

bindings	1	2	3	4	5	6–10	11–20	max.
absolute	82	21	10	4	1	4	5	20
percentage	61·2	15·7	7·5	3·0	0·7	3·0	3·7	

The distribution of the length of allocation chains (any uninterrupted series of `let` and `letrec` expressions) is shown below:

bindings	1	2	3	4	5	6–10	11–20	21+	max.
absolute	5 006	1 395	599	313	120	276	69	30	114
percentage	64·1	17·9	7·7	4·0	1·5	3·5	0·9	0·4	

In addition to explicit allocation, it may be necessary to heap allocate large constructors.

C.4.1 Free variables

By definition, global definitions do not have free variables (see section 4.5.4), so the information presented here relates to only the non-global bindings.

The distribution for the 1208 function bindings is as follows:

free variables	1	2	3	4	5	6–10	11–20	21+	max.
absolute	651	314	110	37	25	47	19	5	36
percentage	53·9	26·0	9·1	3·1	2·1	3·9	1·6	0·4	

For the 8401 thunks, the free-variable distribution is:

free variables	1	2	3	4	5	6–10	11–20	21+	max.
absolute	4 078	2 336	1 063	414	226	253	29	2	21
percentage	48·5	27·8	12·7	4·9	2·7	3·0	0·3	0·0	

The Glasgow Haskell compiler treats constructor bindings (anything of the form *var* = `r` → *cons atoms*) as a special case, so free-variable information was not recorded for these cases.

C.4.2 Function arguments

The distribution of the number of arguments for the 6 571 functions is given below:

arguments	1	2	3	4	5	6–10	11–20	21+	max.
absolute	3 213	1 856	732	320	152	264	32	2	22
percentage	48·9	28·2	11·1	4·9	2·3	4·0	·5	0·0	

C.5 case expressions

Note that for the purpose of this section, the original definition of the **case** expression is used (i.e. `let#` and `letstrict` are considered to be **case** expressions using named defaults).

Of the 12 709 **case** expressions which scrutinise prelude constructors, the number of constructors associated with the data type of the scrutinee is distributed as follows:

constructors	1	2	3	4	5	6–10	11–20	21+	max.
absolute	6 991	5 626	78	0	14	0	0	0	5
percentage	55·0	44·3	0·6	0	0·1	0	0	0	

With regards to the unary constructors, 3 687 of the **case** expressions are used to de-construct the lifted primitive types *Int*, *Char*, etc, and 492 de-construct the tuples used to support type classes.

The 3 279 **case** expressions which scrutinise the user-defined algebraic data types have the following distribution:

constructors	1	2	3	4	5	6–10	11–20	21+	max.
absolute	1 473	497	263	364	108	325	231	18	36
percentage	44·9	15·2	8·0	11·1	3·3	9·9	7·0	0·5	

The distribution for both prelude-defined and user-defined data types (15988 **case** expressions in total) is given, below:

constructors	1	2	3	4	5	6–10	11–20	21+	max.
absolute	8 464	6 123	341	364	122	325	231	18	36
percentage	52·9	38·3	2·1	2·3	0·8	2·0	1·4	0·1	

Of these, 2 245 (14·0 percent) are `letstrict` expressions.

The 2 245 literal `case` expressions take the following types:

type	`Char#`	`Int#`	`Float#`	`Double#`	`Word#`
absolute	209	1 407	266	360	3
percentage	9·3	62·7	11·8	16·0	0·1

Of these, 2125 (94·7 percent) are `let#` expressions.

C.6 Constructor application

The distribution of the number of arguments of the 3161 user-defined constructor applications is given below:

arguments	0	1	2	3	4	5	6–10	11–20	max.
absolute	197	1 348	928	355	122	131	74	6	20
percentage	6·2	42·6	29·4	11·2	3·9	4·1	2·3	0·2	

As for the prelude constructors, of which there are 7 998, the distribution is:

arguments	0	1	2	3	4	5	6–10	11–20	max.
absolute	1 725	12	1 683	4 257	193	24	101	3	19
percentage	21·6	0·2	21·0	53·2	2·4	0·3	1·3	0·0	

The following distribution illustrates the total number of constructors that belong to the same user-defined type as the constructor being applied:

constructors	1	2	3	4	5	6–10	11–20	21+	max.
absolute	716	661	334	218	289	542	375	26	36
percentage	22·7	20·9	10·6	6·9	9·1	17·1	11·9	0·8	

The same distribution for the prelude constructors is:

constructors	1	2	3	4	5	6–10	11–20	21+	max.
absolute	2 283	5 612	81	1	21	0	21	0	18
percentage	28·5	70·2	1·0	0·0	0·3	0	0·3	0	

C.7 Limitations of the analysis

A number of criticisms can be levelled at the data presented in the previous sections:

- static analysis is a poor indicator of what would actually happen during execution
- the style of STG$'$ code generated is an artifact of the Glasgow Haskell compiler, and should not be used to infer general patterns
- most of the `nofib` benchmarks were coded with a sequential architecture in mind, so the data has no meaning in a parallel context
- larger programs, such as `anna` and `veritas`, will dominate the results

To a certain extent, all of these points are valid. But as the data is only intended to serve as a rough guide, the problems of the collection method can be overlooked.

Appendix D

Polymorphic type rules for the STG′ language

This chapter presents the type rules discussed in section 4.5.3, with the order of presentation closely following that of the abstract syntax (see figure 4.1).

D.1 Terminology

The notation adopted here is based on that used by Peyton Jones and Wadler [1992].

Type rules

All of the rules take the following form:

type signature

$$\mathit{NAME}\ \frac{\begin{array}{c}\mathit{premise}_1\\ \vdots\\ \mathit{premise}_n\end{array}}{\mathit{conclusion}}$$

Usually, both the premises and conclusion will be *judgement forms*:

$$\mathit{environment}\ \overset{\text{rule group}}{\vdash}\ \mathit{construct} : \mathit{type}$$

More generally, for rules which make use of or generate more complex data, the *judgement form* will look like:

$$\mathit{inherited}\ \overset{\text{rule group}}{\vdash}\ \mathit{construct} : \mathit{synthesised}$$

Environments

An *environment* is a finite mapping, usually from identifiers to types, either explicitly constructed, e.g. $\{x \mapsto \tau_1, y \mapsto \tau_2\}$, or created by merging two existing environments:

$$(env_1 \oplus env_2)\ var \quad = \quad \begin{cases} env_1\ var, \text{if } var \in dom(env_1) \text{ and } var \notin dom(env_2) \\ env_2\ var, \text{if } var \in dom(env_2) \text{ and } var \notin dom(env_1) \end{cases}$$

$$(env_1 \overset{\rightarrow}{\oplus} env_2)\ var \quad = \quad \begin{cases} env_1\ var, \text{if } var \in dom(env_1) \text{ and } var \notin dom(env_2) \\ env_2\ var, \text{if } var \in dom(env_2) \end{cases}$$

where $dom(env)$ returns the domain of the environment. As shown previously, an identifier's value can be retrieved by applying it to the environment $(env\ var)$, but the preferred method is to treat the mapping as a set of tuples, and test for membership i.e. $(id, value) \in env$.

The environments used by the algorithm, as summarised in table 4.2, are as follows:

constructor environment for the purpose of typing, a constructor $cons\ \tau_1 \ldots \tau_n$, belonging to the algebraic data type $\chi\ \pi_1 \ldots \pi_m$, is treated as a function of type: $\tau_1 \rightarrow \cdots \tau_n \rightarrow \chi\ \pi_1 \ldots \pi_m$.

primitive environment rather than providing an explicit rule for every primitive function, this environment maps primitives to polymorphic types,

general type environment is used to store the polytype of all bound polymorphic variables currently in scope, including top-level definitions and all variable defined by `let(rec)` expressions.

local type environment stores the monotype of the formal arguments of the current binding, and any additional variables introduced by case alternatives, and `letstrict` or `let#` expressions.

type-constructor environment records the arity (the number of type arguments required) of each type constructor, along with the number and sequence of its constituent constructors.

Free variables

The free type variables of either a language construct or a whole environment may be determined using a named rule of the $\mathcal{FV}[\![\,]\!]$ algorithm.

Implicit conditions

To reduce the size of the presented rules, a number of conditions have been left implicit:

- where a type attribute has been inferred by two or more different rules, each of the resulting values must unify. The unified type is then used as the final result
- occurrences of $env_1 \oplus env_2$ require that: $dom(env_1) \cap dom(env_2) == \emptyset$

D.2 Programs

$$\mathbf{PE} \overset{\text{program}}{\vdash} \mathbf{program : GVE}$$

$$\textit{PROGRAM}\quad \frac{\begin{array}{l} \overset{\text{typedecls}}{\vdash} \; \mathit{typedecls} : (\mathit{TCE}, \mathit{CE}) \\ \mathit{TE} = (\mathit{CE}, \mathit{PE}, \emptyset, \emptyset, \emptyset) \\ \mathit{GVE}_0 = \emptyset \\ \langle \mathit{bindings}_1, \ldots, \mathit{bindings}_m \rangle = \mathit{partition}\ \mathit{bindings} \\ \mathit{TE} \oplus \mathit{GVE}_i \overset{\text{recbinds}}{\vdash} \mathit{bindings}_{i+1} : \mathit{GVE}'_{i+1} \\ \mathit{GVE}_{i+1} = \mathit{GVE}_i \oplus \mathit{GVE}'_{i+1} \\ (\mathit{main}, \mathit{Dialogue}) \in \mathit{GVE}_m \end{array}}{\mathit{PE} \overset{\text{program}}{\vdash} \mathit{typedecls}\ \mathit{bindings} : \mathit{GVE}_m}$$

The *partition* function first constructs a dependency graph of the mutually recursive definitions and uses this to breaks the bindings up into strongly-connected components. The resulting groups are then sorted into topological order. The total effect of this ordering is to convert the top-level bindings into a series of nested `letrec` expressions. This minimises the impact of the *monomorphism restriction* as described in section 4.5.3.

D.3 Algebraic data types

Type declarations

$$\overset{\text{typedecls}}{\vdash} \mathbf{typedecls : (TCE, CE)}$$

$$\textit{TYPEDECLS}\quad \frac{\begin{array}{l} \mathit{TCE} \overset{\text{typedecl}}{\vdash} \mathit{typedecl}_i : (\mathit{TCE}_i, \mathit{CE}_i) \\ \mathit{TCE} = \oplus_{0 \leq i \leq t} \mathit{TCE}_i \\ \mathrm{CE} \;\;= \oplus_{0 \leq i \leq t} \mathit{CE}_i \end{array}}{\overset{\text{typedecls}}{\vdash} \mathit{typedecl}_1 \ldots \mathit{typedecl}_t : (\mathit{TCE}, \mathit{CE})}$$

Individual type declarations

$$\mathbf{TCE} \overset{\text{typedecl}}{\vdash} \mathbf{typedecl : (TCE, CE)}$$

$$\textit{TYPEDECL}\quad \frac{\begin{array}{l} \mathcal{FV}_{\mathit{condecls}}[\![\mathit{condecls}]\!] == \{\alpha_1, \ldots, \alpha_v\} \\ \mathit{TCE}; \chi\ \alpha_1 \ldots \alpha_v \overset{\text{condecls}}{\vdash} \mathit{condecls} : (\mathit{CE}, n, \langle \mathit{cons}_1, \ldots, \mathit{cons}_n \rangle) \\ \mathit{TCE}' = \{\chi \mapsto (v, n, \langle \mathit{cons}_1, \ldots, \mathit{cons}_n \rangle)\} \end{array}}{\mathit{TCE} \overset{\text{typedecl}}{\vdash} \texttt{data}\ \chi\ \alpha_1 \ldots \alpha_v = \mathit{condecls} : (\mathit{TCE}', \mathit{CE})}$$

Constructor declarations

$$\mathbf{TCE};\tau_\chi \overset{\text{condecls}}{\vdash} \mathbf{condecls} : (\mathbf{CE}, \mathbf{n}, \langle\mathbf{cons}\rangle)$$

$$\mathit{CONDECLS}\qquad \frac{\begin{array}{l} \mathit{TCE};\tau_\chi \overset{\text{condecl}}{\vdash} \mathit{condecl}_i : (\mathit{CE}_i, \mathit{cons}_i)\\ \mathit{CE} = \oplus_{1\le i\le n}\mathit{CE}_i \end{array}}{\mathit{TCE};\tau_\chi \overset{\text{condecls}}{\vdash} \mathit{condecl}_1 \ldots \mathit{condecl}_n : (\mathit{CE}, n, \langle \mathit{cons}_1, \ldots, \mathit{cons}_n\rangle)}$$

Individual constructor declarations

$$\mathbf{TCE};\tau_\chi \overset{\text{condecl}}{\vdash} \mathbf{condecl} : (\mathbf{CE}, \mathbf{cons})$$

$$\mathit{CONDECL}\qquad \frac{\begin{array}{l} \mathit{TCE} \overset{\text{monotype}}{\vdash} \tau_i \quad (0 \le i \le f)\\ \emptyset \overset{\text{gen}}{\vdash} \tau_1 \to \cdots \to \tau_f \to \tau_\chi : \sigma\\ \mathit{CE} = \{\mathit{cons} \mapsto (f, \sigma)\} \end{array}}{\mathit{TCE};\tau_\chi \overset{\text{condecl}}{\vdash} \mathit{cons}\ \tau_1 \ldots \tau_f : (\mathit{CE}, \mathit{cons})}$$

Monotypes

$$\mathbf{TCE} \overset{\text{monotype}}{\vdash} \tau$$

$$\mathit{BOXED\text{-}MONO}\qquad \frac{\mathit{TCE} \overset{\text{boxedtype}}{\vdash} \pi}{\mathit{TCE} \overset{\text{monotype}}{\vdash} \pi}$$

$$\mathit{UNBOXED\text{-}MONO}\qquad \frac{}{\mathit{TCE} \overset{\text{monotype}}{\vdash} \nu}$$

Boxed types

$$\textbf{TCE} \overset{\text{boxedtype}}{\vdash} \pi$$

BOXED-VAR

$$\frac{}{TCE \overset{\text{boxedtype}}{\vdash} \alpha}$$

BOXED-FUN

$$\frac{TCE \overset{\text{monotype}}{\vdash} \tau_i \quad i \in \{1,2\}}{TCE \overset{\text{boxedtype}}{\vdash} \tau_1 \to \tau_2}$$

BOXED-CON

$$\frac{\begin{array}{c}(\chi, (v, n, \langle cons_1, \ldots, cons_n \rangle)) \in TCE \\ TCE \overset{\text{boxedtype}}{\vdash} \pi_i \quad (0 \le i \le v)\end{array}}{TCE \overset{\text{monotype}}{\vdash} \chi\, \pi_1 \ldots \pi_v}$$

D.4 Bindings and lambda forms

Recursive bindings

$$\textbf{TE} \overset{\text{recbinds}}{\vdash} \textbf{binds} : \textbf{GVE}$$

REC-BINDS

$$\frac{\begin{array}{l} TE \overset{\rightarrow}{\oplus} LVE \overset{\text{binds}}{\vdash} binds : GVE \\ LVE = \{var_i \mapsto \tau_i \mid (var_i, \sigma_i) \in GVE, \overset{\text{spec}}{\vdash} \sigma_i : \tau_i\}\end{array}}{TE \overset{\text{recbinds}}{\vdash} binds : GVE}$$

Bindings

$$\textbf{TE} \overset{\text{binds}}{\vdash} \textbf{binds} : \textbf{GVE}$$

BINDS

$$\frac{\begin{array}{l} TE \overset{\text{bind}}{\vdash} bind_i : (var_i, \tau_i) \\ TE \overset{\text{gen}}{\vdash} \tau_i : \sigma_i \\ GVE = \oplus_{i \le n} \{var_i \mapsto \sigma_i\}\end{array}}{TE \overset{\text{binds}}{\vdash} bind_1 \ldots bind_n : GVE}$$

Individual bindings

$$\textbf{TE} \overset{\text{bind}}{\vdash} \textbf{bind} : (\textbf{var}, \tau)$$

BIND

$$\frac{\begin{array}{l} TE \overset{\text{lambda}}{\vdash} lambda\_form : \tau \\ \tau \le \alpha \end{array}}{TE \overset{\text{bind}}{\vdash} var = lambda\_form : (var, \tau)}$$

Simple bindings

$$\mathbf{TE} \overset{\text{simplebind}}{\vdash} \mathbf{simplebind} : (\mathbf{var}, \tau)$$

$$\mathit{SIMPLE\text{-}BIND} \qquad \frac{TE \overset{\text{exp}}{\vdash} exp : \tau}{TE \overset{\text{simplebind}}{\vdash} var = exp : (var, \tau)}$$

Lambda forms

$$\mathbf{TE} \overset{\text{lambda}}{\vdash} \mathbf{lambda} : \tau$$

$$\mathit{LAMBDA} \qquad \frac{\begin{array}{c} LVE = \oplus_{i \leq n}\{arg_i \mapsto \tau_i\} \\ TE \overset{\rightarrow}{\oplus} LVE \overset{\text{exp}}{\vdash} exp : \tau_{exp} \end{array}}{TE \overset{\text{lambda}}{\vdash} vars_{free}\ \pi\ arg_1 \ldots arg_n \rightarrow exp : \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_{exp}}$$

D.5 Expressions

$$\mathbf{TE} \overset{\text{exp}}{\vdash} \mathbf{exp} : \tau$$

The `let` expression

$$\mathit{LET\text{-}EXP} \qquad \frac{\begin{array}{c} TE \overset{\text{binds}}{\vdash} bindings : GVE \\ TE \overset{\rightarrow}{\oplus} GVE \overset{\text{exp}}{\vdash} exp : \tau_{exp} \end{array}}{TE \overset{\text{exp}}{\vdash} \texttt{let}\ bindings\ exp : \tau_{exp}}$$

The `letrec` expression

$$\mathit{LETREC\text{-}EXP} \qquad \frac{\begin{array}{c} TE \overset{\text{recbinds}}{\vdash} bindings : GVE \\ TE \overset{\rightarrow}{\oplus} GVE \overset{\text{exp}}{\vdash} exp : \tau_{exp} \end{array}}{TE \overset{\text{exp}}{\vdash} \texttt{letrec}\ bindings\ exp : \tau_{exp}}$$

The `let#` expression

$$\mathit{LET\#\text{-}EXP} \qquad \frac{\begin{array}{l} TE \overset{\text{simplebind}}{\vdash} simplebind : (var, \nu) \\ LVE = \{var \mapsto \nu\} \\ TE \overset{\rightarrow}{\oplus} LVE \overset{\text{exp}}{\vdash} exp : \tau_{exp} \end{array}}{TE \overset{\text{exp}}{\vdash} \texttt{let\#}\ simplebind\ exp : \tau_{exp}}$$

The `letstrict` expression

$$\textit{LETSTRICT-EXP}\qquad \frac{\begin{array}{l} TE \overset{\text{simplebind}}{\vdash} \textit{simplebind} : (\textit{var}, \chi\ \pi_1 \ldots \pi_v) \\ LVE = \{\textit{var} \mapsto \chi\ \pi_1 \ldots \pi_v\} \\ TE \overset{\rightarrow}{\oplus} LVE \overset{\text{exp}}{\vdash} \textit{exp} : \tau_{exp} \end{array}}{TE \overset{\text{exp}}{\vdash} \texttt{letstrict}\ \textit{simplebind}\ \textit{exp} : \tau_{exp}}$$

The `case` expression

$$\textit{CASE-EXP}\qquad \frac{\begin{array}{l} TE \overset{\text{exp}}{\vdash} \textit{exp} : \tau_{exp} \\ TE \overset{\text{alts}}{\vdash} \textit{alts} : \tau_{exp} \rightarrow \tau_{result} \wedge \textit{no\_overlap alts} \\ TE \overset{\text{default}}{\vdash} \textit{default} : \tau_{exp} \rightarrow \tau_{result} \end{array}}{TE \overset{\text{exp}}{\vdash} \texttt{case}\ \textit{exp}\ \texttt{of}\ \textit{alts}\ \textit{default} : \tau_{result}}$$

The *no_overlap* function examines the left-hand side of each alternative making sure that there is no repetition.

Variable application

$$\textit{APPLY-EXP}\qquad \frac{\begin{array}{l} TE \overset{\text{var}}{\vdash} \textit{var}_{fun} : \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_{result} \\ TE \overset{\text{atom}}{\vdash} \textit{atom}_i : \tau_i \quad (0 \leq i \leq n) \end{array}}{TE \overset{\text{exp}}{\vdash} \textit{var}_{fun}\ \textit{atom}_1 \ldots \textit{atom}_n : \tau_{result}}$$

Constructor application

$$\textit{CONS-EXP}\qquad \frac{\begin{array}{l} (\textit{cons}, (n, \sigma)) \in CE \\ TE \overset{\text{spec}}{\vdash} \sigma : \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \chi\ \pi_1 \ldots \pi_v \\ TE \overset{\text{atom}}{\vdash} \textit{atom}_i : \tau_i \quad (0 \leq i \leq n) \end{array}}{TE \overset{\text{exp}}{\vdash} \textit{cons}\ \textit{atom}_1 \ldots \textit{atom}_n : \chi\ \pi_1 \ldots \pi_v}$$

Primitive functions

$$\textit{PRIM-EXP}\qquad \frac{\begin{array}{l} (\textit{primitive}, (n, \sigma)) \in PE \\ TE \overset{\text{spec}}{\vdash} \sigma : \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_{result} \\ TE \overset{\text{atom}}{\vdash} \textit{atom}_i : \tau_i \quad (0 \leq i \leq n) \end{array}}{TE \overset{\text{exp}}{\vdash} \textit{primitive}\ \textit{atom}_1 \ldots \textit{atom}_n : \tau_{result}}$$

Literal values

$$\textit{LIT-EXP}\qquad \frac{\overset{\text{literal}}{\vdash} \textit{literal} : \nu}{TE \overset{\text{exp}}{\vdash} \textit{literal} : \nu}$$

D.6 case alternatives

General alternatives

$$\mathbf{TE} \overset{\text{alts}}{\vdash} \mathbf{alts} : \tau \rightarrow \tau$$

$$\mathit{LIT\text{-}ALTS} \quad \frac{TE \overset{\text{lalt}}{\vdash} lalt_i : \nu \rightarrow \tau \quad (1 \leq i \leq n)}{TE \overset{\text{alts}}{\vdash} lalt_1 \ldots lalt_n : \nu \rightarrow \tau}$$

$$\mathit{ALG\text{-}ALTS} \quad \frac{TE \overset{\text{aalt}}{\vdash} aalt_i : \chi\ \pi_1 \ldots \pi_v \rightarrow \tau \quad (1 \leq i \leq n)}{TE \overset{\text{alts}}{\vdash} aalt_1 \ldots aalt_n : \chi\ \pi_1 \ldots \pi_v \rightarrow \tau}$$

Literal alternatives

$$\mathbf{TE} \overset{\text{lalt}}{\vdash} \mathbf{lalt} : \tau \rightarrow \tau$$

$$\mathit{LIT\text{-}ALT} \quad \frac{\begin{array}{c} \overset{\text{literal}}{\vdash} literal : \nu \\ TE \overset{\text{exp}}{\vdash} exp : \tau_{exp} \end{array}}{TE \overset{\text{lalt}}{\vdash} literal \rightarrow exp : \nu \rightarrow \tau_{exp}}$$

Algebraic alternatives

$$\mathbf{TE} \overset{\text{aalt}}{\vdash} \mathbf{aalt} : \tau \rightarrow \tau$$

$$\mathit{ALG\text{-}ALT} \quad \frac{\begin{array}{l} TE \overset{\text{pattern}}{\vdash} pattern : (\chi\ \pi_1 \ldots \pi_v, LVE) \\ TE \overset{\rightarrow}{\oplus} LVE \overset{\text{exp}}{\vdash} exp : \tau_{exp} \end{array}}{TE \overset{\text{aalt}}{\vdash} pattern \rightarrow exp : \chi\ \pi_1 \ldots \pi_v \rightarrow \tau_{exp}}$$

Constructor patterns

$$\mathbf{TE} \overset{\text{pattern}}{\vdash} \mathbf{cons\ vars} : (\chi\ \pi_{\mathbf{1}} \ldots \pi_{\mathbf{v}} \rightarrow \tau, \mathbf{LVE})$$

$$\mathit{PATTERN} \quad \frac{\begin{array}{l} (cons, (n, \sigma)) \in CE \\ TE \overset{\text{spec}}{\vdash} \sigma : \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \chi\ \pi_1 \ldots \pi_v \\ LVE = \oplus_{i \leq n} \{var_i \mapsto \tau_i\} \end{array}}{TE \overset{\text{pattern}}{\vdash} cons\ var_1 \ldots var_n : (\chi\ \pi_1 \ldots \pi_v, LVE)}$$

Default expressions

$$\mathbf{TE} \overset{\text{default}}{\vdash} \mathbf{default} : \tau \rightarrow \tau$$

$$\mathit{DEFAULT} \quad \frac{TE \overset{\text{exp}}{\vdash} exp : \tau_{exp}}{TE \overset{\text{default}}{\vdash} \_ \rightarrow exp : \tau \rightarrow \tau_{exp}}$$

D.7 Atoms, variables and literals

Atoms

$$\mathbf{TE} \overset{\text{atom}}{\vdash} \mathbf{atom} : \tau$$

$$\mathit{VAR\text{-}ATOM} \qquad \frac{TE \overset{\text{var}}{\vdash} var : \tau}{TE \overset{\text{atom}}{\vdash} var : \tau}$$

$$\mathit{LIT\text{-}ATOM} \qquad \frac{\overset{\text{literal}}{\vdash} literal : \nu}{TE \overset{\text{atom}}{\vdash} literal : \nu}$$

Variables

$$\mathbf{TE} \overset{\text{var}}{\vdash} \mathbf{var} : \tau$$

$$\mathit{LOCAL\text{-}VAR} \qquad \frac{(var, \tau) \in LVE}{TE \overset{\text{var}}{\vdash} var : \tau}$$

$$\mathit{GENERAL\text{-}VAR} \qquad \frac{\begin{array}{c}(var, \sigma) \in GVE \\ TE \overset{\text{spec}}{\vdash} \sigma : \tau\end{array}}{TE \overset{\text{var}}{\vdash} var : \tau}$$

Literal values

$$\mathbf{TE} \overset{\text{literal}}{\vdash} \mathbf{literal} : \nu$$

$$\begin{array}{llllll}
\mathit{INT\text{-}LIT} & \overset{\text{literal}}{\vdash} int & : Int\# & \mathit{STRING\text{-}LIT} & \overset{\text{literal}}{\vdash} string & : String\# \\
\mathit{FLOAT\text{-}LIT} & \overset{\text{literal}}{\vdash} float & : Float\# & \mathit{MACH\text{-}LIT} & \overset{\text{literal}}{\vdash} mach & : Mach\# \\
\mathit{CHAR\text{-}LIT} & \overset{\text{literal}}{\vdash} char & : Char\# & \mathit{ADDR\text{-}LIT} & \overset{\text{literal}}{\vdash} addr & : Addr\#
\end{array}$$

D.8 Generalisation and specialisation

$$\overset{\text{spec}}{\vdash}\ \sigma : \tau$$

$$\mathbf{TE}\ \overset{\text{gen}}{\vdash}\ \tau : \sigma$$

$$SPEC \quad \frac{\tau \leq \sigma}{\overset{\text{spec}}{\vdash}\ \sigma : \tau}$$

$$GEN \quad \frac{\forall \alpha_i \bullet \alpha_i \in (\mathcal{FV}_{monotype}[\![\tau]\!]\text{``}\mathcal{FV}_{type\_env}[\![TE]\!]) \quad (1 \leq i \leq n)}{TE\ \overset{\text{gen}}{\vdash}\ \tau : \forall \alpha_1 \ldots \alpha_n . \tau}$$

Appendix E

Free variables of the STG language

This chapter presents the free-variable algorithm discussed in section 4.5.4, with the order of presentation closely following that of the abstract syntax (see figure 4.1).

E.1 Programs

$$\mathcal{FV}_{program}\llbracket\,\rrbracket :: \mathit{program} \rightarrow \{\mathit{var}\}$$

$$\mathcal{FV}_{program}\left\llbracket \begin{array}{lcl} \mathit{var}_1 & = & \mathit{lambda}_1 \\ & \vdots & \\ \mathit{var}_n & = & \mathit{lambda}_n \end{array} \right\rrbracket \begin{array}{ll} = \{\} & \text{(definition)} \\ = \bigcup_{i \leq n} \mathcal{FV}_{lambda}\llbracket \mathit{lambda}_i \rrbracket \; \{\mathit{var}_1, \ldots, \mathit{var}_n\} & \text{(derived)} \end{array}$$

E.2 Algebraic data types

Constructor declarations

$$\mathcal{FV}_{condecls}\llbracket\,\rrbracket :: \mathit{condecls} \rightarrow \{\alpha\}$$

$$\mathcal{FV}_{condecl}\llbracket \mathit{condecl}_1 \ldots \mathit{condecl}_n \rrbracket = \bigcup_{i \leq n} \mathcal{FV}_{condecl}\llbracket \mathit{condecl}_i \rrbracket$$

Individual constructor declarations

$$\mathcal{FV}_{condecl}\llbracket\,\rrbracket :: \mathit{condecl} \rightarrow \{\alpha\}$$

$$\mathcal{FV}_{condecl}\llbracket \mathit{cons}\ \tau_1 \ldots \tau_f \rrbracket = \bigcup_{i \leq f} \mathcal{FV}_{monotype}\llbracket \tau_i \rrbracket$$

Monotypes

$$\mathcal{FV}_{monotype}\llbracket\,\rrbracket :: \tau \rightarrow \{\alpha\}$$

$$\begin{array}{ll} \mathcal{FV}_{monotype}\llbracket \pi \rrbracket & = \mathcal{FV}_{boxedtype}\llbracket \pi \rrbracket \\ \mathcal{FV}_{monotype}\llbracket \nu \rrbracket & = \{\} \end{array}$$

Boxed types

$$\mathcal{FV}_{boxedtype}[\![\,]\!] :: \pi \to \{\alpha\}$$

$$\begin{array}{rl}
\mathcal{FV}_{boxedtype}[\![\alpha]\!] & = \{\alpha\} \\
\mathcal{FV}_{boxedtype}[\![\tau_1 \to \tau_2]\!] & = \mathcal{FV}_{monotype}[\![\tau_1]\!] \cup \mathcal{FV}_{monotype}[\![\tau_2]\!] \\
\mathcal{FV}_{boxedtype}[\![\chi\ \pi_1 \ldots \pi_n]\!] & = \bigcup_{i \leq n} \mathcal{FV}_{boxedtype}[\![\pi_i]\!]
\end{array}$$

E.3 Lambda forms

$$\mathcal{FV}_{lambda}[\![\,]\!] :: lambda\_form \to \{var\} \to \{var\}$$

$$\mathcal{FV}_{lambda}\left[\!\!\left[\begin{array}{ll} var_{free_1} \ldots var_{free_m} & \\ \pi & \to exp \\ var_{arg_1} \ldots var_{arg_n} & \end{array}\right]\!\!\right] \begin{array}{l} g = \{var_{free_1}, \ldots, var_{free_m}\} \qquad \text{(definition)} \\ \quad = \mathcal{FV}_{exp}[\![exp]\!]\ g' \setminus var_{args} \qquad \text{(derived)} \\ \qquad \text{where} \\ \qquad g' \qquad = g \setminus vars_{args} \\ \qquad vars_{arg} = \{var_{arg_1}, \ldots, var_{arg_n}\} \end{array}$$

E.4 Expressions

$$\mathcal{FV}_{exp}[\![\,]\!] :: exp \to \{var\} \to \{var\}$$

The `let` expression

$$\mathcal{FV}_{exp}\left[\!\!\left[\texttt{let} \begin{array}{rcl} var_1 & = & lambda_1 \\ & \vdots & \\ var_n & = & lambda_n \end{array}\ exp\right]\!\!\right]\ g =$$

$$\begin{array}{ll}
\multicolumn{2}{l}{(free_{exp} \setminus vars_{bound}) \cup free_{lambdas}} \\
\multicolumn{2}{l}{\text{where}} \\
\ free_{exp} & = \mathcal{FV}_{exp}[\![exp]\!]\ g' \\
\ free_{lambdas} & = \bigcup_{i \leq n} \mathcal{FV}_{lambda}[\![lambda_i]\!]\ g \\
\ g' & = g \setminus vars_{bound} \\
\ vars_{bound} & = \{var_1, \ldots, var_n\}
\end{array}$$

The `letrec` expression

$$\mathcal{FV}_{exp}\left[\!\!\left[\texttt{letrec} \begin{array}{rcl} var_1 & = & lambda_1 \\ & \vdots & \\ var_n & = & lambda_n \end{array}\ exp\right]\!\!\right]\ g =$$

$$\begin{array}{ll}
\multicolumn{2}{l}{(free_{exp} \cup free_{lambdas}) \setminus vars_{bound}} \\
\multicolumn{2}{l}{\text{where}} \\
\ free_{exp} & = \mathcal{FV}_{exp}[\![exp]\!]\ g' \\
\ free_{lambdas} & = \bigcup_{i \leq n} \mathcal{FV}_{lambda}[\![lambda_i]\!]\ g' \\
\ g' & = g \setminus vars_{bound} \\
\ vars_{bound} & = \{var_1, \ldots, var_n\}
\end{array}$$

The `let#` expression

$$\begin{aligned}&\mathcal{FV}_{exp}[\![\texttt{let\#}\ (var = exp_{rhs})\ exp_{body}]\!]\ g = \\ &\mathcal{FV}_{exp}[\![exp_{rhs}]\!]\ g \cup (\mathcal{FV}_{exp}[\![exp_{body}]\!]\ g' \setminus \{var\}) \\ &\text{where } g' = g \setminus \{var\}\end{aligned}$$

The `letstrict` expression

$$\begin{aligned}&\mathcal{FV}_{exp}[\![\texttt{letstrict}\ (var = exp_{rhs})\ exp_{body}]\!]\ g \\ &\quad = \mathcal{FV}_{exp}[\![exp_{rhs}]\!]\ g \cup (\mathcal{FV}_{exp}[\![exp_{body}]\!]\ g' \setminus \{var\}) \\ &\quad\ \text{where } g' = g \setminus \{var\}\end{aligned}$$

The `case` expression

$$\begin{aligned}&\mathcal{FV}_{exp}[\![\texttt{case}\ exp\ \texttt{of}\ alts\ default]\!]\ g = \\ &\mathcal{FV}_{exp}[\![exp]\!]\ g \cup \mathcal{FV}_{alts}[\![alts]\!]\ g \cup \mathcal{FV}_{default}[\![default]\!]\ g\end{aligned}$$

Variable application

$$\mathcal{FV}_{exp}[\![var_{fun}\ atoms]\!]\ g = \mathcal{FV}_{atoms}[\![atoms]\!]\ g \cup \mathcal{FV}_{var}[\![var_{fun}]\!]\ g$$

Constructor application

$$\mathcal{FV}_{exp}[\![cons\ atoms]\!]\ g = \mathcal{FV}_{atoms}[\![atoms]\!]\ g$$

Primitive functions

$$\mathcal{FV}_{exp}[\![primitive\ atoms]\!]\ g = \mathcal{FV}_{atoms}[\![atoms]\!]\ g$$

Literal values

$$\mathcal{FV}_{exp}[\![literal]\!]\ g = \{\}$$

E.5 `case` alternatives

General alternatives

$$\mathcal{FV}_{alts}[\![]\!] :: alts \rightarrow \{var\} \rightarrow \{var\}$$

$$\begin{aligned}\mathcal{FV}_{alts}[\![lalt_1 \ldots lalt_n]\!]\ g &= \textstyle\bigcup_{i \le n} \mathcal{FV}_{lalt}[\![lalt_i]\!]\ g \\ \mathcal{FV}_{alts}[\![aalt_1 \ldots aalt_n]\!]\ g &= \textstyle\bigcup_{i \le n} \mathcal{FV}_{aalt}[\![aalt_i]\!]\ g\end{aligned}$$

Literal alternatives

$$\mathcal{FV}_{lalt}[\![]\!] :: lalt \rightarrow \{var\} \rightarrow \{var\}$$

$$\mathcal{FV}_{lalt}[\![literal \rightarrow exp]\!]\ g = \mathcal{FV}_{exp}[\![exp]\!]\ g$$

Algebraic alternatives

$$\mathcal{FV}_{aalt}[\![\,]\!] :: aalt \rightarrow \{var\} \rightarrow \{var\}$$

$$\begin{aligned}\mathcal{FV}_{aalt}[\![cons\ var_1 \ldots\ var_n \rightarrow exp]\!]\ g = \ & \mathcal{FV}_{exp}[\![exp]\!]\ g' \setminus vars_{bound} \\ & \text{where } g' = g \setminus vars_{bound} \\ & \text{and} \quad vars_{bound} = \{var_1, \ldots, var_n\}\end{aligned}$$

Default expressions

$$\mathcal{FV}_{default}[\![\,]\!] :: default \rightarrow \{var\} \rightarrow \{var\}$$

$$\mathcal{FV}_{default}[\![\_ \rightarrow exp]\!]\ g = \mathcal{FV}_{exp}[\![exp]\!]\ g$$

E.6 Atoms and variables

Atoms

$$\mathcal{FV}_{atoms}[\![\,]\!] :: atoms \rightarrow \{var\} \rightarrow \{var\}$$

$$\mathcal{FV}_{atoms}[\![atom_1 \ldots atom_n]\!]\ globals = \bigcup_{i \leq n} \mathcal{FV}_{atom}[\![atom_i]\!]\ globals$$

Individual atoms

$$\mathcal{FV}_{atom}[\![\,]\!] :: atom \rightarrow \{var\} \rightarrow \{var\}$$

$$\begin{aligned}\mathcal{FV}_{atom}[\![var]\!]\ globals &= \mathcal{FV}_{var}[\![var]\!]\ globals \\ \mathcal{FV}_{atom}[\![literal]\!]\ globals &= \{\}\end{aligned}$$

Variables

$$\mathcal{FV}_{var}[\![\,]\!] :: var \rightarrow \{var\} \rightarrow \{var\}$$

$$\mathcal{FV}_{var}[\![var]\!]\ globals = \{var\} \setminus globals$$

Appendix F

The RISC target language

F.1 Introduction

This chapter presents the instruction set used by the RISC-processor model outlined in chapter 7. Based on the Alpha instruction formats [DEC, 1992, figures 3-1 through 3-6, pages 3-8 to 3-12], the instructions are split into four categories: memory references, branches, operate instructions, and system instructions. The Haskell representation of the instruction set is as follows:

Haskell

```
data Instruction = MemoryInst  MemoryOpCode Register Register MemoryOffset |
                   BranchInst  BranchOpCode Register Register MemoryOffset |
                   Op1Inst    OperateOpCode Register Register Register     |
                   Op2Inst    OperateOpCode Register Word     Register     |
                   SysInst        SysOpCode          Word
```

F.2 Operand notation

The notation for the instruction-set operands is described in the following table:

notation	description
$name_{reg}$	one of the thirty-two general-purpose registers, which will be associated with the given *name*
$immediate_x$	a signed integer, made up of x bits
$offset_x$	a signed integer, made up of x bits, used as an address offset
$offset_{x\leftarrow y}$	a signed integer, made up of x bits, which will be shifted y bits to the left and used as an address offset
reg_imm_x	either the contents of a general-purpose register or an x-bit signed integer

F.3 Memory references

Haskell

```
data MemoryOpCode = LD  | LL | LA | LAH | ST | SC
```

	instruction		description
LD	*load*	$offset_{16}(base_{reg})$ $target_{reg}$	load a word from the address (word-aligned) formed by adding the 16-bit signed offset and the contents of the base register. The value is then stored in the target register
LL	$load_{linked}$	$offset_{16}(base_{reg})$ $target_{reg}$	in addition to performing a regular load (*LD*), the instruction indicates the start of a semaphore action. If the address is accessed between the exceution of this instruction and the matching conditional store (*SC*), the conditional store will fail.
LA	$load_{address}$	$offset_{16}(base_{reg})$ $target_{reg}$	this instruction does not access memory, it simply loads the target register with sum of the offset and the base register
LAH	$load_{high}$	$offset_{16\leftarrow 16}(base_{reg})$ $target_{reg}$	similar to the load address instruction, except the offset is first (arithmetically) shifted sixteen bits to the left
ST	*store*	$value_{reg}$ $offset_{16}(base_{reg})$	stores the word conatined in the value register into the address formed by adding the 16-bit signed offset and the contents of the base register
SC	$store_{linked}$	$value_{reg}$ $offset_{16}(base_{reg})$	this is the second instruction of a semaphore pair – if the memory location has not been accessed since the linked load, the word stored in the value register will be loaded into the memory address, and the value register will be set to one. If, however, the address has been accessed, no store will take place, and the value register will be set to zero

F.4 Branch instructions

Haskell

```
data BranchOpCode = JMP | JSR | BR  | BSR | CBR RISCCondition
```

The *condition x* functions (`RISCCondition`) are described in section F.7.

Unconditional branches

	instruction		description
JMP	*jump*	$offset_{16\leftarrow 2}(base_{reg})$	the 16-bit signed offset is first shifted two places to the left, then added to the base register to form the target address (which must be word aligned). The PC is set to this new address
JSR	$jump_{link}$	$offset_{16\leftarrow 2}(base_{reg})$ $link_{reg}$	in addition to performing a regular jump (*JMP*), the link register is loaded with the value of the current PC, allowing a subroutine to return control back to the caller
BR	*branch*	$offset_{21\leftarrow 2}$	similar to a jump, but the (larger) offset is added to the current PC to form the target address
BSR	$branch_{link}$	$offset_{21\leftarrow 2}$ $link_{reg}$	loads the link register with the value of the current PC, before branching

Conditional branches

	instruction			description
BEQ	$branch_{x==0}$	x_{reg}	$offset_{21\leftarrow 2}$	branch if x is zero
BNE	$branch_{x\neq 0}$	x_{reg}	$offset_{21\leftarrow 2}$	branch if x is not zero
BLT	$branch_{x<0}$	x_{reg}	$offset_{21\leftarrow 2}$	branch if x is less than zero
BLE	$branch_{x\leq 0}$	x_{reg}	$offset_{21\leftarrow 2}$	branch if x is less than or equal to zero
BGT	$branch_{x>0}$	x_{reg}	$offset_{21\leftarrow 2}$	branch if x is greater than zero
BGE	$branch_{x\geq 0}$	x_{reg}	$offset_{21\leftarrow 2}$	branch if x is greater than or equal to zero
BLBC	$branch_{bit0\_clear}$	x_{reg}	$offset_{21\leftarrow 2}$	branch if the low bit of x is zero
BLBS	$branch_{bit0\_set}$	x_{reg}	$offset_{21\leftarrow 2}$	branch if the low bit of x is one

F.5 Operate instructions

Haskell

```
data OperateOpCode = ADD   | S2ADD | ADDT      | SUB | S2SUB | SUBT | MUL| DIV |
                     CMOVE RISCCondition       |
                     AND   Bool    | BIS Bool  | XOR Bool |
                     SLL   | SRL   | SRA       |
                     CMPEQ | CMPLT | CMPLE       deriving Eq
```

The *condition x* functions (`RISCCondition`) are described in section F.7.

Arithmetic operations

	instruction		description
ADD	*add*	$value_{reg}$ reg_imm_8 $target_{reg}$	signed addition of the first two arguments, the result of which is stored in the target register
ADDT	add_{trap}	$value_{reg}$ reg_imm_8 $target_{reg}$	as for *ADD* but an over or underflow will generate an exception (that must be explicitly trapped with a $barrier_{trap}$ instruction – *TRAPB*)
S2ADD	add_{shift_2}	$value_{reg}$ reg_imm_8 $target_{reg}$	before performing the addition, the second argument is shifted left by two bits
SUB	*subtract*	$value_{reg}$ reg_imm_8 $target_{reg}$	signed subtraction of the second argument from the first, the result of which is stored in the target register
SUBT	$subtract_{trap}$	$value_{reg}$ reg_imm_8 $target_{reg}$	as for *SUB* but can generate an exception (see *ADDT* for further details)
S2SUB	$subtract_{shift\_2}$	$value_{reg}$ reg_imm_8 $target_{reg}$	before performing the subtraction, the second argument is shifted left by two bits
MUL	*multiply*	$value_{reg}$ reg_imm_8 $target_{reg}$	signed multiplication of the two arguments (no exception generation)
DIV	*multiply*	$value_{reg}$ reg_imm_8 $target_{reg}$	signed division of the two arguments (exception generated when dividing by zero)

Move instructions

	instruction	description
CMOVEQ	$move_{x==0}$ x_{reg} reg_imm_8 $target_{reg}$	if x is zero then set the target to the value of the second argument
CMOVNE	$move_{x\neq 0}$ x_{reg} reg_imm_8 $target_{reg}$	perform the move if x is not zero
CMOVLT	$move_{x<0}$ x_{reg} reg_imm_8 $target_{reg}$	perform the move if x is less than zero
CMOVLE	$move_{x\leq 0}$ x_{reg} reg_imm_8 $target_{reg}$	perform the move if x is less than or equal to zero
CMOVGT	$move_{x>0}$ x_{reg} reg_imm_8 $target_{reg}$	perform the move if x is greater than zero
CMOVGE	$move_{x\geq 0}$ x_{reg} reg_imm_8 $target_{reg}$	perform the move if x is greater than or equal to zero
CMOVLBC	$move_{bit0\_clear}$ x_{reg} reg_imm_8 $target_{reg}$	perform the move if the low bit of x is zero
CMOVLBS	$move_{bit0\_set}$ x_{reg} reg_imm_8 $target_{reg}$	perform the move if the low bit of x is one

An unconditional move from x_{reg} to y_{reg} is effected by the $move_{x==0}$ $zero_{reg}$ x_{reg} y_{reg}. The *condition* x (`RISCCondition`) functions are described in section F.7.

Logical instruction

	instruction	description
AND	and $value_{reg}$ reg_imm_8 $target_{reg}$	perform a bit-wise logical and of the two arguments and store the result in the target
BIS	or $value_{reg}$ reg_imm_8 $target_{reg}$	as for *and*, but use the bit-wise logical or operation
XOR	xor $value_{reg}$ reg_imm_8 $target_{reg}$	as for *and*, but use the bit-wise logical *xor* operation
BIC	and_{not} $value_{reg}$ reg_imm_8 $target_{reg}$	complement the second argument before performing the *and* operation
ORNOT	or_{not} $value_{reg}$ reg_imm_8 $target_{reg}$	complement the second argument before performing the *or* operation
EQV	xor_{not} $value_{reg}$ reg_imm_8 $target_{reg}$	complement the second argument before performing the *xor* operation

Comparisons

	instruction	description
CMPEQ	$compare_{x==y}$ $value_{reg}$ reg_imm_8 $target_{reg}$	if the two values are equal then set the target register to one, otherwise set it to zero
CMPLT	$compare_{x<y}$ $value_{reg}$ reg_imm_8 $target_{reg}$	if the first argument is less than the second then set the target register to one, otherwise set it to zero
CMPLE	$compare_{x\leq y}$ $value_{reg}$ reg_imm_8 $target_{reg}$	if the first argument is less than or equal to the second then set the target register to one, otherwise set it to zero

Negation of register x_{reg} is effected by the or_{not} $zero_{reg}$ x_{reg} instruction.

Shift instructions

	instruction	description
SLL	$shift_{left}$ x_{reg} reg_imm_5 $target_{reg}$	the first argument is shifted left by the number of bits specified by the second argument (up to a maximum of 32 places) and the result stored in the target
SRL	$shift_{right}$ x_{reg} reg_imm_5 $target_{reg}$	as for $shift_{left}$, but shift to the right
SRA	$shift_{arithmetic}$ x_{reg} reg_imm_5 $target_{reg}$	as for $shift_{right}$, but the sign bit is invariable

F.6 System instructions

Haskell

```
data SysOpCode = CALL_PAL | TRAPB | MB | MBW
```

	instruction	description
CALL_PAL	$syscall\ immediate_{26}$	cause a system-call exception
TRAPB	$barrier_{trap}$	if an arithmetic exception is pending, then skip the next instruction
MB	$barrier_{read}$	wait until all outstanding reads have completed (only applicable in a shared memory environment)
MBW	$barrier_{write}$	wait until all outstanding writes have completed (only applicable in a shared-memory environment)

F.7 Condition codes

```
data RISCCondition = EQ | NE | LT | LE | GT | GE | LBC | LBS
```

condition	branch instruction	move instruction	*condition x*
EQ	BEQ	CMOVEQ	$(x == 0)$
NE	BNE	CMOVNE	$(x \neq 0)$
LT	BLT	CMOVLT	$(x < 0)$
LE	BLE	CMOVLE	$(x \leq 0)$
GT	BGT	CMOVGT	$(x > 0)$
GE	BGE	CMOVGE	$(x \geq 0)$
LBC	BLBC	CMOVLBC	$(x\ mod\ 2) == 0$
LBS	BLBS	CMOVLBS	$(x\ mod\ 2) \neq 0$

Appendix G

State-transition rules for modelling a RISC processor

G.1 Introduction

This chapter presents the state-transition rules needed to complete the RISC-uniprocessor model outlined in chapter 7. The terminology follows that presented in section 4.8.2.

G.2 Decoding instructions

Pending exceptions

$$\begin{array}{|llllllr|}
\hline
\boxed{1} & \mathit{Decode} & \mathit{pc} & \mathit{regs} & \mathit{memory} & \mathit{semaphore} & (\mathit{pending}, \mathit{mask}, \mathit{counter}, \mathit{trigger}) \\
\multicolumn{7}{|l|}{\text{such that } \mathit{pending} \setminus (\mathit{mask} \cup \{\mathit{Overflow}\}) \neq \emptyset} \\
\Longrightarrow & \mathit{Exception} & \mathit{pc} & \mathit{registers} & \mathit{memory} & \mathit{semaphore} & (\mathit{pending}, \mathit{mask}, \mathit{counter}, \mathit{trigger}) \\
\hline
\end{array}$$

Instruction fetch and decode

$$\begin{array}{|llllll l|}
\hline
\boxed{2} & \mathit{Decode} & \mathit{pc} & \mathit{registers} & \mathit{memory} & \mathit{semaphore} & \mathit{exceptions} \\
\Longrightarrow & \mathit{Execute\ instruction} & \mathit{pc}' & \mathit{registers} & \mathit{memory} & \mathit{semaphore} & \mathit{exceptions} \\
\text{where} & \multicolumn{6}{l|}{\mathit{instruction} = \mathit{decode}\ \mathit{memory}(\mathit{pc})} \\
 & \multicolumn{6}{l|}{\mathit{pc}' = \mathit{pc} +_{32} 4} \\
\hline
\end{array}$$

G.3 The post-execution phase

$$\begin{array}{|llllll l|}
\hline
\boxed{3} & \mathit{PostExec} & \mathit{pc} & \mathit{registers} & \mathit{memory} & \mathit{semaphore} & (\mathit{pending}\ , \mathit{mask}, \mathit{counter}\ , \mathit{trigger}) \\
\Longrightarrow & \mathit{Decode} & \mathit{pc} & \mathit{registers} & \mathit{memory} & \mathit{semaphore} & (\mathit{pending}', \mathit{mask}, \mathit{counter}', \mathit{trigger}) \\
\text{where} & \multicolumn{6}{l|}{\mathit{counter}' = \mathit{counter} +_{32} 1} \\
 & \multicolumn{6}{l|}{\mathit{pending}' = \mathit{pending} \cup \mathit{clock\_interrupt}} \\
 & \multicolumn{6}{l|}{\mathit{clock\_interrupt} = \mathit{if}\ (\mathit{trigger} == \mathit{counter})\ \mathit{then}\ \{\mathit{Clock}\}\ \mathit{else}\ \emptyset} \\
\hline
\end{array}$$

G.4 Exceptions

4	$Exception$	pc	$registers$	$memory$	$semaphore$	$exceptions$
$\Longrightarrow$	$Exception$	pc	$registers$	$memory$	$semaphore$	$exceptions$

G.5 Memory references

Unaligned access

5	$Execute\ load/store\ offset(base)$	pc	$registers$	$memory$	$semaphore$	$(pending\ , mask, ctr, tr)$
such that $(load/store \in \{load, load_{linked}, store, store_{linked}\})$ and $(address\ mod\ 4 \neq 0)$						
$\Longrightarrow$	$PostExec$	pc	$registers$	$memory$	$semaphore$	$(pending', mask, ctr, tr)$
where	$pending' = pending \cup \{Unaligned_{data}\}$					
	$address = offset +_{32} registers(base)$					

Load instruction

6	$Execute\ load\ offset(base)\ target$	pc	$registers$	$memory$	$semaphore$	$exceptions$
$\Longrightarrow$	$PostExec$	pc	$registers'$	$memory$	$semaphore$	$exceptions$
where	$registers' = registers[target \mapsto value]$					
	$value = memory(address)$					
	$address = offset +_{32} registers(base)$					

Linked loads

7	$Execute\ load_{linked}\ offset(base)\ target$	pc	$registers$	$memory$	$semaphore$	$exceptions$
$\Longrightarrow$	$PostExec$	pc	$registers'$	$memory$	$(address, false)$	$exceptions$
where	$registers' = registers[target \mapsto value]$					
	$value = memory(address)$					
	$address = offset +_{32} registers(base)$					

Store instructions

8	$Execute\ store\ source\ offset(base)$	pc	$registers$	$memory$	$(address_{sem}, stale?)$	$exceptions$
$\Longrightarrow$	$PostExec$	pc	$registers$	$memory'$	$(address_{sem}, stale?')$	$exceptions$
where	$memory' = memory[address \mapsto registers(source)]$					
	$stale?' = if\ (address == address_{sem})\ then\ true\ else\ stale?$					
	$address = offset +_{32} registers(base)$					

Conditional store instruction

Clean address

9	$Execute\ store_{link}\ source\ offset(base)$	pc	$registers$	$memory$	$(address_{sem}, stale?)$	$except.$
such that $(address == address_{sem})$ and $(stale? == false)$						
$\Longrightarrow$	$PostExec$	pc	$registers'$	$memory'$	$(address_{sem}, true)$	$except.$
where	$memory' = memory[address \mapsto registers(source)]$					
	$registers' = registers[source \mapsto 1]$					
	$address = offset +_{32} registers(base)$					

Dirty or a non-equal address

10	$Execute\ store_{link}\ source\ offset(base)$	pc	$registers$	$memory$	$(address_{sem}, stale?)$	$except.$
$\Longrightarrow$	$PostExec$	pc	$registers'$	$memory$	$(address_{sem}, true)$	$except.$
where	$registers' = registers[source \mapsto 0]$					
	$address = offset +_{32} registers(base)$					

Load address

11	$Execute\ load_{address}\ offset(base)\ target$	pc	$registers$	$memory$	$semaphore$	$exceptions$
$\Longrightarrow$	$PostExec$	pc	$registers'$	$memory$	$semaphore$	$exceptions$
where	$registers' = registers[target \mapsto address]$					
	$address = offset +_{32} registers(base)$					

Load address high

12	$Execute\ load_{address\_high}\ offset(base)\ target$	pc	$registers$	$memory$	$semaphore$	$exceptions$
$\Longrightarrow$	$PostExec$	pc	$registers'$	$memory$	$semaphore$	$exceptions$
where	$registers' = registers[target \mapsto address]$					
	$address = offset' +_{32} registers(base)$					
	$offset' = shift_{left}\ offset\ 16$					

G.6 Branch instructions

Unaligned computed jumps

13	$Execute\ jump\ offset(base)$	pc	$registers$	$memory$	$semaphore$	$(pending\ , mask, ctr, tr)$
such that ($jump \in \{jump, jump_{link}\}$) and ($address\ mod\ 4 \neq 0$)						
$\Longrightarrow$	$PostExec$	pc	$registers$	$memory$	$semaphore$	$(pending', mask, ctr, tr)$
where	$pending' = pending \cup \{Unaligned_{instruction}\}$					
	$address = offset +_{32} register(base)$					

Computed jumps

14	$Execute\ jump\ offset(base)$	pc	$registers$	$memory$	$semaphore$	$exceptions$
$\Longrightarrow$	$PostExec$	pc'	$registers$	$memory$	$semaphore$	$exceptions$
where	$pc' = offset +_{32} registers(base)$					

Linked computed jumps

15	$Execute\ jump_{link}\ offset(base)\ link$	pc	$registers$	$memory$	$semaphore$	$exceptions$
$\Longrightarrow$	$PostExec$	pc'	$registers'$	$memory$	$semaphore$	$exceptions$
where	$pc' = offset +_{32} registers(base)$					
	$registers' = registers[link \mapsto pc]$					

Unconditional branches

16	$\mathit{Execute\ branch\ offset}$	pc	$\mathit{registers}$	memory	$\mathit{semaphore}$	$\mathit{exceptions}$
$\Longrightarrow$	$\mathit{PostExec}$	pc'	$\mathit{registers}$	memory	$\mathit{semaphore}$	$\mathit{exceptions}$
where	$\mathit{pc}' = \mathit{offset}' +_{32} \mathit{pc}$					
	$\mathit{offset}' = \mathit{shift}_{\mathit{left}}\ \mathit{offset}\ 2$					

Linked branches

17	$\mathit{Execute\ branch}_{\mathit{link}}\ \mathit{offset\ link}$	pc	$\mathit{registers}$	memory	$\mathit{semaphore}$	$\mathit{exceptions}$
$\Longrightarrow$	$\mathit{PostExec}$	pc'	$\mathit{registers}'$	memory	$\mathit{semaphore}$	$\mathit{exceptions}$
where	$\mathit{pc}' = \mathit{offset}' +_{32} \mathit{pc}$					
	$\mathit{registers}' = \mathit{registers}[\mathit{link} \mapsto \mathit{pc}]$					
	$\mathit{offset}' = \mathit{shift}_{\mathit{left}}\ \mathit{offset}\ 2$					

Conditional branches

Condition satisfied

18	$\mathit{Execute\ branch}_{\mathit{condition}}\ x\ \mathit{offset}$	pc	$\mathit{registers}$	memory	$\mathit{semaphore}$	$\mathit{exceptions}$
such that $(\mathit{condition\ registers}(x) == \mathit{true})$ and $(\mathit{address} \bmod 4 == 0)$						
$\Longrightarrow$	$\mathit{PostExec}$	pc'	$\mathit{registers}$	memory	$\mathit{semaphore}$	$\mathit{exceptions}$
where	$\mathit{pc}' = \mathit{offset}' +_{32} \mathit{pc}$					
	$\mathit{offset}' = \mathit{shift}_{\mathit{left}}\ \mathit{offset}\ 2$					

Condition not met

19	$\mathit{Execute\ branch}_{\mathit{condition}}\ x\ \mathit{offset}$	pc	$\mathit{registers}$	memory	$\mathit{semaphore}$	$\mathit{exceptions}$
$\Longrightarrow$	$\mathit{PostExec}$	pc	$\mathit{registers}$	memory	$\mathit{semaphore}$	$\mathit{exceptions}$

G.7 Operate instructions

Trapped addition

No overflow

20	$\mathit{Execute\ add}_{\mathit{trap}}\ \mathit{register}_1 \begin{Bmatrix} \mathit{immediate} \\ \mathit{register}_2 \end{Bmatrix} \mathit{target}$	pc	$\mathit{registers}$	memory	$\mathit{semaphore}$	ex
such that $(\mathit{sign\ argument}_1 \neq \mathit{sign\ argument}_2)$ or $(\mathit{sign\ argument}_1 == \mathit{sign\ result})$						
$\Longrightarrow$	$\mathit{PostExec}$	pc	$\mathit{registers}'$	memory	$\mathit{semaphore}$	ex
where	$\mathit{registers}' = \mathit{registers}[\mathit{target} \mapsto \mathit{result}]$					
	$\mathit{result} = \mathit{argument}_1 +_{32} \mathit{argument}_2$					
	$\mathit{argument}_1 = \mathit{registers}(\mathit{register}_1)$					
	$\mathit{argument}_2 = \begin{cases} \mathit{immediate} \\ \mathit{registers}(\mathit{register}_2) \end{cases}$					

Overflow occurs

21	$Execute\ add_{trap}$	pc	$registers$	$memory$	$semaphore$	$(pending\ , mask, counter, trigger)$
$\Longrightarrow$	$PostExec$	pc	$registers'$	$memory$	$semaphore$	$(pending', mask, counter, trigger)$
where	$pending' = pending \cup \{Overflow\}$					

Trapped subtraction

No overflow

22	$Execute\ subtract_{trap}\ reg_1 \begin{Bmatrix} imm \\ reg_2 \end{Bmatrix}\ target$	pc	$registers$	$memory$	$semaphore$	$exceptions$
such that $(sign\ argument_1 == sign\ argument_2)$ or $(sign\ argument_1 == sign\ result)$						
$\Longrightarrow$	$PostExec$	pc	$registers'$	$memory$	$semaphore$	$exceptions$
where	$registers' = registers[target \mapsto result]$					
	$result = argument_1 -_{32} argument_2$					
	$argument_1 = registers(reg_1)$					
	$argument_2 = \begin{cases} imm \\ registers(reg_2) \end{cases}$					

Overflow occurs

23	$Execute\ subtract_{trap}$	pc	$registers$	$memory$	$semaphore$	$(pending\ , mask, counter, trigger)$
$\Longrightarrow$	$PostExec$	pc	$registers'$	$memory$	$semaphore$	$(pending', mask, counter, trigger)$
where	$pending' = pending \cup \{Overflow\}$					

Move instructions

Condition satisfied

24	$Execute\ move_{condition}\ reg_1 \begin{Bmatrix} immed \\ reg_2 \end{Bmatrix}\ taget$	pc	$registers$	$memory$	$semaphore$	$except.$
such that $(condition\ argument_1 == true)$						
$\Longrightarrow$	$PostExec$	pc	$registers'$	$memory$	$semaphore$	$except.$
where	$registers' = registers[target \mapsto argument_2]$					
	$argument_1 = registers(reg_1)$					
	$argument_2 = \begin{cases} immed \\ registers(reg_2) \end{cases}$					

Condition not met

25	$Execute\ move_{condition}\ reg_1 \begin{Bmatrix} immed \\ reg_2 \end{Bmatrix}\ taget$	pc	$registers$	$memory$	$semaphore$	$exceptions$
$\Longrightarrow$	$PostExec$	pc	$registers$	$memory$	$semaphore$	$exceptions$

Arithmetic and logical operations, shifts and comparisons

The remaining operations are regular, so we only need to define one reduction rule:

26	$Execute\ operator\ register_1 \left\{\begin{matrix} immediate \\ register_2 \end{matrix}\right\} target$	pc	$registers$	$memory$	$semaphore$	ex
such that	$(operator \in \{add, add_{shift\_2}, subtract, subtract_{shift\_2}, multiply, divide,$ $and, and_{not}, or, or_{not}, xor, xor_{not},$ $shift_{left}, shift_{right}, shift_{arithmetic}, compare_{condition}\})$					
$\Longrightarrow$	$PostExec$	pc	$registers'$	$memory$	$semaphore$	ex
where	$registers' = registers[target \mapsto result]$					
	$result = operator\ argument_1\ argument_2$					
	$argument_1 = registers(register_1)$					
	$argument_2 = \left\{\begin{matrix} immediate \\ registers(register_2) \end{matrix}\right.$					

G.8 System instructions

System calls

27	$Execute\ syscall\ arg$	pc	$registers$	$memory$	$semaphore$	$(pending\ , mask, counter, trigger)$
$\Longrightarrow$	$PostExec$	pc	$registers$	$memory$	$semaphore$	$(pending', mask, counter, trigger)$
where	$pending' = pending \cup \{SysCall\}$					

Arithmetic trap

28	$Execute\ barrier_{trap}$	pc	$registers$	$memory$	$semaphore$	$(pending\ , mask, counter, trigger)$
$\Longrightarrow$	$PostExec$	pc'	$registers$	$memory$	$semaphore$	$(pending', mask, counter, trigger)$
where	$pc' = if\ (Overflow \in pending)\ then\ (pc +_{32} 4)\ else\ pc$					
	$pending' = pending \setminus \{Overflow\}$					

Read barrier

29	$Execute\ barrier_{read}$	pc	$registers$	$memory$	$semaphore$	$exceptions$
$\Longrightarrow$	$PostExec$	pc	$registers$	$memory$	$semaphore$	$exceptions$

Write barrier

30	$Execute\ barrier_{write}$	pc	$registers$	$memory$	$semaphore$	$exceptions$
$\Longrightarrow$	$PostExec$	pc	$registers$	$memory$	$semaphore$	$exceptions$

G.9 Condition codes

condition	branch instruction	move instruction	*condition x*
EQ	BEQ	CMOVEQ	$(x = 0)$
NE	BNE	CMOVNE	$(x \neq 0)$
LT	BLT	CMOVLT	$(x < 0)$
LE	BLE	CMOVLE	$(x \leq 0)$
GT	BGT	CMOVGT	$(x > 0)$
GE	BGE	CMOVGE	$(x \geq 0)$
LBC	BLBC	CMOVLBC	$(x\ mod\ 2) = 0$
LBS	BLBS	CMOVLBS	$(x\ mod\ 2) \neq 0$

Appendix H

Compilation rules of the STG′-machine language

This chapter presents the state-transition rules used to prototype a modern optimising compiler for functional languages. The notation used is described in section 4.8.2, while the rules themselves are introduced and described in chapter 8.

H.1 The initial state

INIT	Code	Expression-code stack	Continuation stack	Pending bindings	Code blocks	Globals
	$Continue$	$\langle\rangle_{stack}$	$\langle\rangle_{stack}$	$pending$	$blocks$	σ

where

$$\begin{aligned} pending &= \{bind_1, \ldots, bind_n\} \\ blocks &= \{label_{node\_g_1} \mapsto \langle label_{info\_g_1}, 0\rangle, \ldots, label_{node\_g_n} \mapsto \langle label_{info\_g_n}, 0\rangle\} \\ \sigma &= \{g_1 \mapsto label_{node\_g_1}, \ldots, g_n \mapsto label_{node\_g_n}\} \\ bind_i &\equiv (g_i = lambda\_form_i) \end{aligned}$$

H.2 The compiler framework

Fl_a	$Continue$	$exps$	$next : conts$	$pending$	$blocks$	σ
$\Longrightarrow$	$next$	$exps$	$conts$	$pending$	$blocks$	σ

Fl_b	$Continue$	$\langle\rangle_{stack}$	$\langle\rangle_{stack}$	$pending$	$blocks$	σ
such that $bind \in pending$						
$\Longrightarrow$	$CompileBind\ bind$	$\langle\rangle_{stack}$	$\langle\rangle_{stack}$	$pending'$	$blocks$	σ
where	$pending' = pending \setminus bind$					

Fl_c	$Continue$	$\langle\rangle_{stack}$	$\langle\rangle_{stack}$	$\langle\rangle_{stack}$	$blocks$	σ
$\Longrightarrow$	$Finish$	$\langle\rangle_{stack}$	$\langle\rangle_{stack}$	$\langle\rangle_{stack}$	$blocks$	σ

$$\begin{array}{lllllll}
\boxed{\text{F2}} & \mathit{CompileBind\ bind} & \mathit{exps} & \mathit{conts} & \mathit{pending} & \mathit{blocks} & \sigma
\end{array}$$

such that $\overset{\text{var}}{\vdash} \mathit{var}_{fun} : \tau_1 \to \cdots \to \tau_n \to \mathit{Int\#}, \ (n \geq 1)$

$$\begin{array}{lllllll}
\Longrightarrow & \mathit{CEval}\ \mathit{exp}_{fun}\ \rho_{init}\ \langle\rangle\ \langle \mathit{return}_{init} \rangle_{stack} & \mathit{exps} & \mathit{conts}' & \mathit{pending} & \mathit{blocks}' & \sigma
\end{array}$$

$$\begin{array}{llll}
\text{where} & \rho_{init} & = & \rho_{args} \oplus \rho_{frees} \oplus \{\mathit{var}_{return} \mapsto \mathit{register}_{24}, \mathit{var}_{node} \mapsto \mathit{register}_{25}\} \\
& \rho_{args} & = & \{\mathit{var}_{arg_1} \mapsto \mathit{operand}_{arg_1}, \ldots, \mathit{var}_{arg_n} \mapsto \mathit{operand}_{arg_n}\} \\
& \mathit{operand}_{arg_i} & = & \begin{cases} \mathit{stack}^A_{x_i} & \overset{\text{var}}{\vdash} \mathit{var}_{arg_i} : \alpha \\ \mathit{stack}^B_{y_i} & \overset{\text{var}}{\vdash} \mathit{var}_{arg_i} : \mathit{Int\#} \end{cases} \\
& \rho_{frees} & = & \{\mathit{var}_{free_1} \mapsto \mathit{operand}_{free_1}, \ldots, \mathit{var}_{free_m} \mapsto \mathit{operand}_{free_m}\} \\
& \mathit{operand}_{free_i} & = & \begin{cases} \mathit{memory}^{\mathit{var}_{node}}_{x_i} & \overset{\text{var}}{\vdash} \mathit{var}_{free_i} : \alpha \\ \mathit{memory}^{\mathit{var}_{node}}_{m-y_i} & \overset{\text{var}}{\vdash} \mathit{var}_{free_i} : \mathit{Int\#} \end{cases} \\
& \mathit{return}_{init} & = & (\mathit{var}_{return}, \mathit{register}_1)_{ext} \\
& \mathit{conts}' & = & (\mathit{SealEntry\ bind}) : (\mathit{ReturnBind\ bind}) : \mathit{conts} \\
& \mathit{bind} & \equiv & (\mathit{var}_{fun} = \mathit{var}_{free_1} \cdots \mathit{var}_{free_m}\ \mathtt{r}\ \mathit{var}_{arg_1} \cdots \mathit{var}_{arg_n} \to \mathit{exp}_{fun})
\end{array}$$

$$\begin{array}{lllllll}
\boxed{\text{F3}} & \mathit{ReturnExpression\ code} & \mathit{exps} & \mathit{conts} & \mathit{pending} & \mathit{blocks} & \sigma \\
\Longrightarrow & \mathit{Continue} & \mathit{code} : \mathit{exps} & \mathit{conts} & \mathit{pending} & \mathit{blocks} & \sigma
\end{array}$$

$$\begin{array}{llllll}
\boxed{\text{F4}} & \mathit{SealEntry\ bind} & \mathit{code}\ : \mathit{exps} & \mathit{conts} & \mathit{pending} & \mathit{blocks} \quad \sigma \\
\Longrightarrow & \mathit{Continue} & \mathit{code}' : \mathit{exps} & \mathit{conts} & \mathit{pending} & \mathit{blocks}' \quad \sigma
\end{array}$$

where $\mathit{code}' = \mathit{check\_args} +\!\!+\ \mathit{check\_stacks} +\!\!+\ \mathit{check\_heap} +\!\!+\ \mathit{code}$

$$\begin{array}{lllllll}
\boxed{\text{F5}} & \mathit{ReturnBind}\ (\mathit{var} = \mathit{lambda\_form}) & \mathit{code} : \mathit{exps} & \mathit{conts} & \mathit{pending} & \mathit{blocks} & \sigma \\
\Longrightarrow & \mathit{Continue} & \mathit{exps} & \mathit{conts} & \mathit{pending} & \mathit{blocks}' & \sigma
\end{array}$$

$$\begin{array}{llll}
\text{where} & \mathit{blocks}' & = & \mathit{blocks} \oplus \{\mathit{label}_{enter\_var} \mapsto \mathit{code}, \ldots, \mathit{label}_{info\_var} \mapsto \mathit{info\_table}\} \\
& \mathit{info\_table} & = & \langle \mathit{label}_{enter\_var}, \mathit{label}_{update\_var}, \ldots, \mathit{label}_{gc\_var} \rangle
\end{array}$$

H.3 Applications

$$\begin{array}{llllllllll}
\boxed{1} & \mathit{CEval}\ (f\ \langle \mathit{atom}_1, \ldots, \mathit{atom}_n \rangle) & \rho & \mathit{code} & \mathit{returns} & \mathit{exps} & \mathit{conts} & \mathit{pending} & \mathit{blocks} & \sigma
\end{array}$$

such that $\overset{\text{var}}{\vdash} f : \alpha$

$$\begin{array}{llllllllll}
\Longrightarrow & \mathit{CEnter\ register}_{25} & \rho' & \mathit{code}' & \mathit{returns} & \mathit{exps} & \mathit{conts} & \mathit{pending} & \mathit{blocks} & \sigma
\end{array}$$

$$\begin{array}{llll}
\text{where} & (\mathit{moves}', \rho') & = & \mathit{combine\_moves\ moves}\ \rho\ \sigma \\
& \mathit{code}' & = & \mathit{code} +\!\!+\ \mathit{moves}' \\
& \mathit{moves} & = & \mathit{load\_node} +\!\!+\ \mathit{push\_args} +\!\!+\ \mathit{save\_volatile\_vars} +\!\!+\ \mathit{stub\_dead\_A\_slots} \\
& \mathit{load\_node} & = & \langle \mathtt{move}\ (\mathit{val}\ \rho\ \sigma\ f), \mathit{register}_{25}; \rangle \\
& \mathit{push\_args} & = & \mathit{push\_arg}_1 : \cdots : \mathit{push\_arg}_n : \langle\rangle \\
& \mathit{push\_arg}_i & = & \mathtt{move}\ (\mathit{atom\_to\_operand\ atom}_i), \mathit{operand}_{arg_i}; \\
& \mathit{operand}_{arg_i} & = & \begin{cases} \mathit{stack}^A_{x_i} & \overset{\text{atom}}{\vdash} \mathit{atom}_i : \alpha \\ \mathit{stack}^B_{y_i} & \overset{\text{atom}}{\vdash} \mathit{atom}_i : \mathit{Int\#} \end{cases}
\end{array}$$

The auxillary function, *atom_to_operand*, is defined below:

$$\begin{array}{|lcl|}
\hline
atom\_to\_operand\ \rho\ \sigma\ literal & = & literal \\
atom\_to\_operand\ \rho\ \sigma\ var & = & val\ \rho\ \sigma\ var \\
\hline
\end{array}$$

$$\begin{array}{llcccccccc}
\boxed{\text{2A}} & \textit{Enter operand} & & \rho & code & returns & exps & conts & pending & blocks & \sigma
\end{array}$$

such that $returns \equiv \langle (var_{return}, \_)_{ext} \rangle_{stack}$

$$\begin{array}{llcccccc}
\Longrightarrow & ReturnExpression\ code' & & exps & conts & pending & blocks & \sigma
\end{array}$$

$$\begin{array}{lllll}
\text{where} & code' & = & code \mathbin{+\!\!+} load\_return \mathbin{+\!\!+} jump \\
 & load\_return & = & \texttt{move}\ (val\ \rho\ \sigma\ var_{return}), register_{24}; \\
 & jump & = & \texttt{move}\ memory_0^{operand}, register_{tmp} \\
 & & & \texttt{jump}\ (register_{tmp})
\end{array}$$

$$\begin{array}{llcccccccc}
\boxed{\text{2B}} & \textit{Enter operand} & \rho & code & returns & exps & conts & pending & blocks & \sigma
\end{array}$$

such that $head\ returns \equiv (k_1 \rightarrow exp_1 \ldots k_n \rightarrow exp_n, \_ \rightarrow exp_d, vars_{free})_{case}$

$$\begin{array}{llcccccccc}
\Longrightarrow & ReturnInt\ register_1 & \rho & \langle\rangle & returns & exps & conts' & pending & blocks & \sigma
\end{array}$$

$$\text{where} \quad conts' \;=\; (CJoinEnter\ (val\ \rho\ \sigma\ var_{node})\ code') : conts$$

$$\begin{array}{llcccccc}
\boxed{\text{2C}} & CJoinEnter\ operand_{node}\ code_{pre\_entry} & code_{post\_entry} : exps & conts & pending & blocks & \sigma
\end{array}$$

$$\begin{array}{llcccccc}
\Longrightarrow & ReturnExpression\ code' & & exps & conts & pending & blocks & \sigma
\end{array}$$

$$\begin{array}{llll}
\text{where} & code' & = & code_{pre\_entry} \mathbin{+\!\!+} jump \mathbin{+\!\!+} code_{post\_entry} \\
 & jump & = & \texttt{move}\ memory_0^{operand_{node}}, register_{tmp}; \\
 & & & \texttt{jump}_{link}\ (register_{tmp}), register_{24};
\end{array}$$

H.4 `let(rec)` expressions

$$\begin{array}{llcccccccc}
\boxed{3} & CEval\ (\texttt{let}\ bindings\ \texttt{in}\ exp) & \rho & code & returns & exps & conts & pending & blocks & \sigma \\
\Longrightarrow & CEval\ exp & \rho' & code' & returns & exps & conts & pending' & blocks & \sigma
\end{array}$$

$$\begin{array}{llll}
\text{where} & \rho' & = & \rho_{moves} \setminus vars_{dead} \\
 & code' & = & code \mathbin{+\!\!+} moves \\
 & pending' & = & \{binding_1, \ldots, binding_n\} \cup pending \\
 & (moves, \rho_{moves}) & = & allocate\_closures\ bindings\ \rho\ \sigma \\
 & vars_{dead} & = & \mathcal{FV}[\![bindings]\!] \setminus \mathcal{FV}[\![exp]\!]
\end{array}$$

The rule for `letrec` expressions is almost identical, requiring only a minor modification of the *allocate_closures* rule (see the description of this function for further details).

H.4.1 Variable bindings

$$allocate\_closures \begin{pmatrix} var_1 & = & lambda_1 \\ & \vdots & \\ var_n & = & lambda_n \end{pmatrix} \rho\ \sigma = combine\_moves\ \{moves_1, \ldots, moves_n\}\ \rho_{binds}\ \sigma$$

$$\begin{array}{lll}
\text{where } \rho_{binds} & = & \rho \overset{\rightarrow}{\oplus} \{var_1 \mapsto heap(offset_1), \ldots, var_n \mapsto heap(offset_n), heap_{max} \mapsto offset_{n+1}\} \\
moves_i & = & create\_closure\ var_i\ lambda_i\ offset_i\ \rho_{rhs}\ \sigma \\
\rho_{rhs} & = & \rho \\
offset_1 & = & val\ \rho\ \sigma\ heap_{max} \\
offset_i & = & offset_1 + \sum_{j<i} \max(closure\_size\ lambda_j, closure\_size_{min})
\end{array}$$

The rule for recursive bindings is almost identical, except that ρ_{rhs} is defined to be ρ_{binds} instead of ρ.

H.4.2 Closure layout

$$\begin{array}{lll}
create\_closure\ var\ base\ (vars_{free}\ \pi\ vars_{args} \rightarrow exp) = & \texttt{move}\ label_{info\_table_{var}}, & memory_0^{base}; \\
& \texttt{move}\ operand_1, & memory_1^{base}; \\
& \vdots & \\
& \texttt{move}\ operand_n, & memory_n^{base};
\end{array}$$

$$\text{where } operand_i \quad = \quad val\ \rho\ \sigma \begin{cases} i^{th} & \text{free variable of type } \pi \\ (n-i)^{th} & \text{free variable of type } \nu \end{cases} (1 \leq i \leq length\ vars_{free})$$

H.5 Case expressions

$$\begin{array}{llllllllll}
\boxed{4} & CEval\ (\texttt{case}\ exp\ \texttt{of}\ alts\ default) & \rho & code & returns & exps & conts & pending & blocks & \sigma \\
\Longrightarrow & CEval\ exp & \rho & code & returns' & exps & conts & pending & blocks & \sigma
\end{array}$$

$$\begin{array}{llll}
\text{where} & returns' & = & (alts, default, vars_{free})_{case} : returns \\
& vars_{free} & = & \mathcal{FV}[\![exp]\!] \cup \mathcal{FV}[\![alts]\!] \cup \mathcal{FV}[\![default]\!]
\end{array}$$

$$\begin{array}{llllllllll}
\boxed{4\text{B}} & CEval\ (\texttt{let\#}\ (var = exp_{rhs})\ exp_{body}) & \rho & code & returns & exps & conts & pending & blocks & \sigma \\
\Longrightarrow & CEval\ exp_{rhs} & \rho & code & returns' & exps & conts & pending & blocks & \sigma
\end{array}$$

$$\begin{array}{llll}
\text{where} & returns' & = & (var, exp_{body}, vars_{dead})_{assign} : returns \\
& vars_{dead} & = & \mathcal{FV}[\![exp_{rhs}]\!] \setminus \mathcal{FV}[\![exp_{body}]\!]
\end{array}$$

The rule for `letstrict` (rule 4A, see figure 4.12) is almost identical, with just the intial expressioon requiring modification.

H.6 Built-in operations

$$\begin{array}{llllllllll}
\boxed{9} & CEval\ (k) & \rho & code & returns & exps & conts & pending & blocks & \sigma \\
\Longrightarrow & CReturnInt\ k & \rho & code & returns & exps & conts & pending & blocks & \sigma
\end{array}$$

$$\boxed{10}\quad \mathit{CEval}\ (f\ \langle\rangle) \qquad \rho \quad \mathit{code} \quad \mathit{returns} \quad \mathit{exps} \quad \mathit{conts} \quad \mathit{pending} \quad \mathit{blocks} \quad \sigma$$

such that $\overset{\text{var}}{\vdash} f : \mathit{Int\#}$

$$\Longrightarrow \quad \mathit{CReturnInt}\ (\mathit{val}\ \rho\ \sigma\ f) \quad \rho \quad \mathit{code} \quad \mathit{returns} \quad \mathit{exps} \quad \mathit{conts} \quad \mathit{pending} \quad \mathit{blocks} \quad \sigma$$

$$\boxed{11\text{A}}\quad \mathit{CReturnInt}\ \mathit{operand} \qquad \rho \quad \mathit{code} \quad \mathit{returns} \quad \mathit{exps} \quad \mathit{conts} \quad \mathit{pending} \quad \mathit{blocks} \quad \sigma$$

such that $\mathit{returns} \equiv \langle(\mathit{var}_{return}, \mathit{register}_{return\_Int\#})_{ext}\rangle_{stack}$

$$\Longrightarrow \quad \mathit{ReturnExpression}\ \mathit{code}' \qquad \mathit{exps} \quad \mathit{conts} \quad \mathit{pending} \quad \mathit{blocks} \quad \sigma$$

$$\begin{array}{llll} \text{where} & \mathit{code}' & = & \mathit{code} \mathbin{+\!\!+} \mathit{moves} \mathbin{+\!\!+} \mathit{trim\_stacks} \mathbin{+\!\!+} \mathit{jump} \\ & (\mathit{moves}, \rho') & = & \mathit{combine\_moves} \begin{pmatrix} \texttt{move}\ \mathit{operand}, \mathit{register}_{return\_Int\#}; \\ \texttt{move}\ (\mathit{val}\ \rho\ \sigma\ \mathit{var}_{return}), \mathit{register}_{return}; \end{pmatrix} \rho\ \sigma \\ & \mathit{jump} & = & \texttt{jump}\ (\mathit{register}_{return}) \end{array}$$

$$\boxed{11\text{B}}\quad \mathit{CReturnInt}\ \mathit{operand} \quad \rho \quad \mathit{code} \quad \mathit{return} : \mathit{returns} \quad \mathit{exps} \quad \mathit{conts} \quad \mathit{pending} \quad \mathit{blocks} \quad \sigma$$

such that $\mathit{return} \equiv (k_1 \rightarrow \mathit{exp}_1 \ldots k_n \rightarrow \mathit{exp}_n, \_ \rightarrow \mathit{exp}_d, \mathit{vars}_{free})_{case}$

$$\Longrightarrow \quad \mathit{cont}_1 \qquad \mathit{exps} \quad \mathit{conts}' \quad \mathit{pending} \quad \mathit{blocks} \quad \sigma$$

$$\begin{array}{llll} \text{where} & \mathit{cont}_i & = & \mathit{CEval}\ \mathit{exp}_i\ \rho_i\ \langle\rangle\ \mathit{returns} \\ & \mathit{conts}' & = & \mathit{conts}_2 : \cdots : \mathit{conts}_n : \mathit{conts}_d : \mathit{join\_returns} : \mathit{conts} \\ & \rho_i & = & \rho \setminus (\mathit{vars}_{free} \setminus \mathcal{FV}[\![\mathit{exp}_i]\!]) \\ & \mathit{join\_returns} & = & \mathit{CJoinReturns}\ \mathit{operand}\ \mathit{code}\ \mathit{return} \end{array}$$

$$\boxed{12}\quad \mathit{CReturnInt}\ \mathit{operand} \quad \rho \quad \mathit{code} \quad \mathit{return} : \mathit{returns} \quad \mathit{exps} \quad \mathit{conts} \quad \mathit{pending} \quad \mathit{blocks} \quad \sigma$$

such that $\mathit{return} \equiv (\mathit{var}, \mathit{exp}_{body}, \mathit{vars}_{dead})_{assign}$

$$\Longrightarrow \quad \mathit{CEval}\ \mathit{exp}_{body} \quad \rho' \quad \mathit{code} \qquad \mathit{returns} \quad \mathit{exps} \quad \mathit{conts} \quad \mathit{pending} \quad \mathit{blocks} \quad \sigma$$

where $\rho' = (\rho \setminus \mathit{vars}_{dead}) \overset{\rightarrow}{\oplus} \{\mathit{var} \mapsto \mathit{operand}\}$

$$\boxed{13'}\quad \mathit{CJoinReturns}\ \mathit{operand}\ \mathit{code}\ \mathit{return} \quad \mathit{exps} \quad \mathit{conts} \quad \mathit{pending} \quad \mathit{blocks} \quad \sigma$$

$$\Longrightarrow \quad \mathit{ReturnExpression}\ \mathit{code}' \qquad \mathit{exps}' \quad \mathit{conts} \quad \mathit{pending} \quad \mathit{blocks}' \quad \sigma$$

$$\begin{array}{llll} \text{where} & \mathit{code}' & = & \mathit{code} \mathbin{+\!\!+} \mathit{select\_alt} \\ & \mathit{select\_alt} & = & \texttt{move}\ k_1, \mathit{register}_{tmp}; \\ & & & \texttt{subtract}\ \mathit{register}_{tmp}, \mathit{operand}, \mathit{register}_{tmp}; \\ & & & \texttt{branch}_{x=0}\ \mathit{register}_{tmp}, \mathit{label}_{unique_1}; \\ & & & \vdots \\ & & & \texttt{move}\ k_n, \mathit{register}_{tmp}; \\ & & & \texttt{subtract}\ \mathit{register}_{tmp}, \mathit{operand}, \mathit{register}_{tmp}; \\ & & & \texttt{branch}_{x=0}\ \mathit{register}_{tmp}, \mathit{label}_{unique_n}; \\ & & & \texttt{jump}\ \mathit{label}_{unique_d}; \\ & \mathit{blocks}' & = & \mathit{blocks} \oplus \{\mathit{label}_{unique_1} \mapsto \mathit{code}_1, \ldots, \\ & & & \qquad \mathit{label}_{unique_n} \mapsto \mathit{code}_n, \\ & & & \qquad \mathit{label}_{unique_d} \mapsto \mathit{code}_d\} \\ & \mathit{return} & \equiv & (k_1 \rightarrow \mathit{exp}_1 \ldots < k_n \rightarrow \mathit{exp}_n, \_ \rightarrow \mathit{exp}_d, \mathit{vars}_{free})_{\mathsf{case}} \\ & \mathit{exps} & \equiv & \mathit{code}_d : \mathit{code}_n : \cdots : \mathit{code}_1 : \mathit{exps}' \end{array}$$

Note, the above rule uses a linear search, which will be inefficient when dealing with large numbers of alternatives – Bernstein [1985] describes an algorithm for generating the optimal combintion of linear and binary searches, and jump tables.

$\boxed{14}$	$CEval\ primIntPlus\#\ atom_1\ atom_2$	ρ	$code$	$returns$	$exps$	$conts$	$pending$	$blocks$	σ
$\Longrightarrow$	$CReturnInt\ register_{tmp_1}$	ρ	$code'$	$returns$	$exps$	$conts$	$pending$	$blocks$	σ

$$\begin{array}{lll}
\text{where} & code' & = & code \mathbin{+\!\!+} add\_atoms \\
 & add\_atoms & = & \texttt{move}\ (atom\_to\_operand\ \rho\ \sigma\ atom_1), register_{tmp_1}; \\
 & & & \texttt{move}\ (atom\_to\_operand\ \rho\ \sigma\ atom_2), register_{tmp_2}; \\
 & & & \texttt{add}\ register_{tmp_1}, register_{tmp_2}, register_{tmp_1};
\end{array}$$

Appendix I

Example RISC programs

This section presents a number of RISC implementations of the STG′ routines from chapter B, as generated by the compilation routines from chapter 8 (including some hand editing).

Section I.1 looks at some of the prelude operations used to support integers, booleans, and lists. Two **nofib** programs, **fib**, and **primes**, are then presented in sections I.2 and through I.3. The remaining sections, I.4 and I.5, look at updating polymorphic algebraic constructors and partial applications respectively.

I.1 Prelude operations

This section looks at the RISC definitions needed to support the three main data types of the Haskell language, namely integers, booleans, and lists. Where applicable, the equivalent STG′ code is also included. All of the RISC bindings have been taken directly from the library of test routines used by the prototyping system (see section 3.4).

I.1.1 Integers

Constants

The following STG′ declerations define the constants zero and one:

STG′ code

```
zero = [] \r []  ->  Int [0#];
one  = [] \r []  ->  Int [1#];
```

The equivalent RISC code consists of two closure definitions, the reversed info table for integers, and the corresponding update code:

RISC code

```
closure zero          Linfo_table_Int,  0;
closure one           Linfo_table_Int,  1;

Linfo_table_Int:

        dw Lupdate_int;               // update routine
        dw Linfo_table_Int;           // fast entry
        dw Linfo_table_Int;           // stnd entry

        load +4(RNp), R1;             // load the integer value into R1
        jump +4 RRet;                 // and return
```

RISC code

```
Lupdate_Int:

        load_high Linfo_table_Int(R0), R2; // load the integer info table
        store R2,   (RNp);                 // and overwrite the closure's
        store R1, +4(RNp);                 // save the integer value
        jump +4 RRet;                      // invoke the actual return address
```

Addition

The STG′ definition for the addition operator is as follows:

STG′ code

```
const.Int.+ = [] \r [x y] -> case x of
 { Int x' -> case y of { Int y' -> let# xy = plusInt# [x', y'] in Int [xy] ; };
 } ;
```

The RISC equivalent includes the obligatory static closure, the reversed info table, and the associated code. The code itself is split into three main parts. The first part ensures there are sufficient arguments available to complete the operation, prepares the return vector (including saving the location of the second argument), and then initiates the evaluation of the first argument:

RISC code

```
closure const.Int.+    Linfo_table_Int_+;

Linfo_table_Int_+:

        dw Lupdate_Int_+;                 // update routine
        dw Linfo_table_Int_+ +12;         // fast entry
        dw Linfo_table_Int_+;             // stnd entry

        subtract RStkA, RStkABase, R1;    // calculate the number of args
        subtract R1, +8, R1;              // are there at least two?
        branch_x<0 R1, Lupdate_PAP;       // if not, perform an update
        load  -8(RStkA), RNp;             // load the node pointer of arg1
        load  -4(RStkA), R1;              // load the node pointer of arg2
        subtract RStkA, +4, RStkA;        // trim the A stack
        subtract RStkB, +4, RStkB;        // trim the B stack
        store R1, -4(RStkA);              // ...and save arg2
        store RRet, +4(RStkB);            // save the return pointer
        load (RNp), R1;                   // get the info table of arg1
        jump_link R1, RRet;               // enter the closure
        branch Lupdate_Int;               // handle an update request
```

The second part is called when the first argument has been evaluated, and it recovers the address of the second argument, prepares another return vector (including saving the integer value of the first argument), and initiates the evaluation of the second argument:

RISC code

```
        load  -4(RStkA), RNp;             // load the node pointer of arg2
        subtract RStkA, +4, RStkA;        // re-allocate stack space (from
        subtract RStkB, +4, RStkB;        // A to B)
        store R1, +4(RStkB);              // save the value of R1 on stack B
        load (RNp), R1;                   // get the info table of arg2
        jump_link R1, RRet;               // enter the closure
        branch Lupdate_Int;               // handle an update request
```

Finally, the two integers are added together and the return continuation invoked:

RISC code

```
        load +4(RStkB), R2;                 // recover the value of arg1
        add R1, R2, R1;                     // add the two values
        load +8(RStkB), RRet;               // recover the return register
        add RStkB, +8, RStkB;               // trim the B stack
        jump +4 RRet;                       // and return normally
```

The less-than operator

STG′ code

```
const.Int.< = [] \r [x y] ->
 case x of { Int x' -> case y of { Int y' -> ltInt# [x', y'] ; } ; } ;
```

The structure of the RISC code is almost identical to that of the addition operation from the previous section. The only major difference is the final operation performed on the two arguments:

RISC code

```
closure const.Int.<   Linfo_table_Int_<;

Linfo_table_Int_<:

        dw Lupdate_Int_<;                    // update routine
        dw Linfo_table_Int_< +1 2;           // fast entry
        dw Linfo_table_Int_<;                // stnd entry
        ...                                  // as for Int_+
        load +4(RStkB), R2;                  // recover the value of arg1
        compare_x<y R2, R1, R1;              // compare the two values
        load +8(RStkB), RRet;                // recover the return register
        add RStkB, +8, RStkB;                // trim the B stack
        jump +4 RRet;                        // and return normally
```

Quotients

The quotient function, quotRem, demonstrates the basic techniques of stack allocation and tail calling. The RISC definition given below is based on Int#-specialised versions of the following prelude function:

STG′ code

```
const.Int.quotRem = [] \r [n d] ->
 let { q = [n d] \u [] -> const.Int.quot n d;
       r = [n d] \u [] -> const.Int.rem  n d; } in Tup2 [q, r];
```

The RISC code simply allocates two thunks, containing the addresses of the two arguments, and simply returns a pair containing the thunks' locations:

RISC code

```
closure const.Int.quotRem   Linfo_table_Int_quotRem;

Linfo_table_Int_quotRem:

        dw Lupdate_Int_quotRem;                 // update routine
        dw Linfo_table_Int_quotRem +12;         // fast entry
        dw Linfo_table_Int_quotRem;             // stnd entry

        subtract RStkA, RStkABase, R1;          // calculate the number of args
        subtract R1, +8, R1;                    // are there at least two?
        branch_x<0 R1, Lupdate_PAP;             // if not, perform an update
```

After checking that there are sufficient arguments available, heap space for the two thunks is allocated (3 words of space for each):

RISC code

```
        add RHp, +24, RHp;                    // allocate space for 2 closures
        compare_x<y RHLimit, RHp, R1;         // ensure there's space
        branch_bit0_set R1, Lgarbage_collect; // otherwise invoke the GC
```

The thunk for q is then filled in:

RISC code

```
        load_high Linfo_table_Int_quotRem_1(R0), R1;
        load_address +0(R1), R1;
        store R1, -24(RHp);                   // set q's info table
        load -8(RStkA), R1;                   // recover the location of n
        store R1, -20(RHp);                   // store n as a free variable
        load -4(RStkA), R2;                   // recover the location of d
        store R2, -16(RHp);                   // store d as a free variable
```

Next, the thunk for r is filled in:

RISC code

```
        load_high Linfo_table_Int_quotRem_2(R0), R3;
        load_address +0(R3), R3;
        store R3, -12(RHp);                   // set r's info table
        store R1,  -8(RHp);                   // store n as a free variable
        store R2,  -4(RHp);                   // store d as a free variable
```

Finally, the tuples is constructed and the appropriate return entry called:

RISC code

```
        subtract RHp, +24, R1;                // set q as the fst pointer
        subtract RHp, +12, R2;                // set d as the snd pointer
        subtract RStkA, +8, RStkA;            // trim the A stack
        jump +4 RRet;                         // and return
```

The info tables and corresponding code for the two thunks, **quotRem_1** and **quotRem_2** are very similar, so only that for determining the quotient is reproduced here:

RISC code

```
Linfo_table_Int_quotRem_1:

        dw Lupdate_Int_quotRem_1;              // update routine
        dw Linfo_table_Int_quotRem_1 +12;      // fast entry
        dw Linfo_table_Int_quotRem_1;          // stnd entry
```

Upon entry to the thunk, an update frame is created:

RISC code

```
        subtract RStkB, +16, RStkB;             // decrease the B stack frame
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler
        store RStkABase, +16(RStkB);            // the A stack pointer
        store RStkBBase, +12(RStkB);            // the B stack pointer
        store RNp,        +8(RStkB);            // the node pointer
        store RRet,       +4(RStkB);            // the current return vector
        move RStkA, RStkABase;
        move RStkB, RStkBBase;
        move RUpdate, RRet;                     // set the return to an update
```

Then, sufficient space is allocated to allow the two free variables, n and d, to be pushed onto the A stack:

RISC code

```
        add RStkA, +8, RStkA;
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler
```

The two free variables are then recovered, and pushed onto the stack, before the division function is tail called:

RISC code

```
        load +4(RNp), R1;                       // retrieve n
        store R1, -8(RStkA);                    // push it on the A stack
        load +8(RNp), R1;                       // retrieve d
        store R1, -4(RStkA);                    // push it on the A stack

        branch Linfo_table_Int_/;               // and call divide
```

Signs

The sign function, signum, demonstrates how the RISC code handles conditionals, and the use of worker functions. The RISC definitions given below are based on **Int#**-specialised versions of the following prelude functions:

STG′ code

```
const.Int.signum = [] \r [x] -> case x of {Int x' -> const.Int.signum.wrk x';};

const.Int.signum.wrk = [] \r [x] -> case x of
 {0#  -> Int [0#];
  _   -> case  gtInt# [x, 0#] of { True  -> Int [  1#]; False -> Int [ -1#]; }
 };
```

The first part of the code forces evaluation of the argument, and then tail calls the worker function, LInt_signum_wrk:

RISC code

```
closure const.Int.signum    Linfo_table_Int_signum;

Linfo_table_Int_signum:

        dw Lupdate_Int_signum;                  // update routine
        dw Linfo_table_Int_signum +12;          // fast entry
        dw Linfo_table_Int_signum;              // stnd entry

        subtract RStkA, RStkABase, R1;          // calculate the number of args
        subtract R1, +4, R1;                    // is there at least one?
        branch_x<0 R1, Lupdate_PAP;             // if not, perform an update

        load -4(RStkA), RNp;                    // pop the arg
        load (RNp), R1;
        subtract RStkA, +4, RStkA;              // re-allocate stack space
        subtract RStkB, +4, RStkB;
        store RRet, +4(RStkB);                  // save the return register
        jump_link R1, RRet;
        branch Lupdate_Int;                     // evaluate the arg

        load +4(RStkB), RRet;                   // recover the return register
        add RStkB, +4, RStkB;                   // trim the stack

        branch LInt_signum_wrk;                 // call the worker function
```

The worker function then determines whether the value is zero, negative or positive, and returns the corresponding integer value:

RISC code

```
LInt_signum_wrk:

        branch_x=0 R1, LInt_signum_wrk_1;       // return 0 if it's zero
        branch_x<0 R1, LInt_signum_wrk_2;       // return -1 if it's negative
        add R0, +1, R1;                         // otherwise return 1
        jump +4 RRet;

LInt_signum_wrk_1:

        move R0, R1;                            // return 0
        jump +4 RRet;

LInt_signum_wrk_2:

        subtract R0, +1, R1;                    // return -1
        jump +4 RRet;
```

I.1.2 Booleans

STG′ code

```
data Bool = True | False;

true      = [] \r [] -> True  [];
false     = [] \r [] -> False [];
otherwise = [] \r [] -> True  [];
```

RISC code

```
closure true            Linfo_table_Bool_True;
closure false           Linfo_table_Bool_False;
closure otherwise       Linfo_table_Bool_True;
```

There are two info tables for dealing with boolean values: one for true, and the other for false. Note, that the code uses the convention that true is represented by the integer one and false by zero:

RISC code

```
Linfo_table_Bool_True:

        dw Lupdate_Bool;                    // update routine
        dw Linfo_table_Bool_True;           // fast entry
        dw Linfo_table_Bool_True;           // stnd entry

        add R0, +1, R1;                     // set R1 (true)
        jump +4 RRet;                       // and return

Linfo_table_Bool_False:

        dw Lupdate_Bool;                    // update routine
        dw Linfo_table_Bool_False;          // fast entry
        dw Linfo_table_Bool_False;          // stnd entry

        move R0, R1;                        // clear R1 (false)
        jump +4 RRet;                       // and return
```

The update code for boolean values is straightforward, simply overwriting the thunks info table with either that for true or false:

RISC code

```
Lupdate_Bool:

        branch_x=0 R1, Lupdate_Bool_False; // determine if it's True or False
        load_high Linfo_table_Bool_True(R0), R2; // load True's info table
        store R2, (RNp);                   // and overwrite the closure's
        jump +4 RRet;                      // invoke the actual return address

Lupdate_Bool_False:

        load_high Linfo_table_Bool_False(R0), R2; // load False's info table
        store R2,   (RNp);                 // and overwrite the closure's
        jump +4 RRet;                      // invoke the actual return address
```

Logical negation

STG′ code

```
not = [] \r [x]   -> case x of { True  -> False [] ; False -> True  [] ; };
```

RISC code

```
closure not                 Linfo_table_not;

Linfo_table_not:

        dw Lupdate_not;                        // update routine
        dw Linfo_table_not +12;                // fast entry
        dw Linfo_table_not;                    // stnd entry
```

First, there's the usual argument check:

RISC code

```
        subtract RStkA, RStkABase, R1;         // calculate the number of args
        subtract R1, +4, R1;                   // are there at least two?
        branch_x<0 R1, Lupdate_PAP;            // if not, perform an update
```

Then the argument is evaluated:

RISC code

```
        load -4(RStkA), RNp;
        subtract RStkA, +4, RStkA;
        subtract RStkB, +4, RStkB;
        store RRet, +4(RStkB);

        load (RNp), R1;
        jump_link R1, RRet;
        branch Lupdate_Bool;
```

After the evaluation returns, the original return vector is retrieved and the logical negation is performed:

RISC code

```
        load +4(RStkB), RRet;                  // retrieve the return vector
        add RStkB, +4, RStkB;                  // trim the stack
        branch_x=0 R1, Lnot_1;                 // if false, return true

        move R0, R1;
        jump +4 RRet;                          // return false

Lnot_1:
        add R0, +1, R1;
        jump +4 RRet;                          // return true
```

I.1.3 Lists

Rather than introducing special syntactic support, the following STG′ declaration is used to define the `List` algebraic data type:

STG′ code

```
data List a = Cons a (List a) | Nil;
```

The following sections look at some of Haskell's `PreludeList` [Hudak et al., 1992, section A.5, pages 106–114] functions.

Nil and null

The `nil` value represents an empty list:

STG′ code

```
nil = [] \r [] -> Nil [];
```

The RISC code simply calls the nil entry from the return vector:

RISC code

```
closure nil                 Linfo_table_Nil;

Linfo_table_Nil:

        dw Lupdate_Nil;                     // update routine
        dw Linfo_table_Nil;                 // fast entry
        dw Linfo_table_Nil;                 // stnd entry

        load_high Lnil_head(R0), R1;        // load dummy values into the
        load_high Lnil_tail(R0), R2;        // head and tail to help debugging
        load -4(RRet), R3;                  // select the nil return entry
        jump R3;                            // and return
```

To illustrate how the return vector is constructed and used, consider the `null` operator:

STG′ code

```
null = [] \r [xss] -> case  xss  of { Nil -> True []; Cons x xs -> False []; };
```

The RISC code performs the usual argument checks and then forces the evaluation of its argument:

RISC code

```
closure null                Linfo_table_null;

Linfo_table_null:

        dw Lupdate_null;                       // update routine
        dw Linfo_table_null +12;               // fast entry
        dw Linfo_table_null;                   // stnd entry

        subtract RStkA, RStkABase, R1;         // calculate the number of args
        subtract R1, +4, R1;                   // is there at least one?
        branch_x<0 R1, Lupdate_PAP;            // if not, perform an update

        load -4(RStkA), RNp;                   // fetch the arg
        load (RNp), R1;
        subtract RStkA, +4, RStkA;             // re-organise the stacks
        subtract RStkB, +4, RStkB;
```

However, the return is set to a custom return vector which correctly handles the nil and non-nil lists:

RISC code

```
        store RRet, +4(RStkB);              // save the return register
        load_high Lnull_return_1(R0), RRet; // set the return register
        load_address +0(RRet), RRet;
        jump R1;
```

The return vector is specified as follows:

RISC code

```
Lnull_return_1:

        dw Lupdate_List;
        dw Lnull_return_List;
        dw Lupdate_Nil;
        dw Lnull_return_Nil;
```

The odd entries point to the associated update routines, while the even entries point to the code to handle the various cases (nil and non-nil lists). Nil-returns are handled as follows (the following two sections will look at the update entries):

RISC code

```
Lnull_return_Nil:

        load +4(RStkB), RRet;               // recover the return reg
        add RStkB, +4, RStkB;               // trim the stack
        add R0, +1, R1;                     // set the return to true
        jump +4 RRet;                       // perform a normal return
```

Non-nil returns simpley return false:

RISC code

```
Lnull_return_List:

        load +4(RStkB), RRet;               // recover the return reg
        add RStkB, +4, RStkB;               // trim the stack
        move R0, R1;                        // set the return to false
        jump +4 RRet;                       // perform a normal return
```

Updating empty lists

As for boolean values, updating a thunk with a nil list is simply a matter of resetting its info table, and then invoking the nil return from the original return vector:

RISC code

```
Lupdate_Nil:

        load_high Linfo_table_Nil(R0), R1;
        store R1, (RNp);

        load -4(RRet), R1;
        jump R1;
```

Updating lists

Updating lists on the other hand is more troublesome. First a cons cell is allocated, and the head and tail stored into it. Then the thunk is overwritten with an indirection to the new cons, before the non-nil return is invoked from the original return vector:

RISC code

```
Lupdate_List:

        add RHp, +12, RHp;                   // allocate space for a cons cell
        compare_x<y RHLimit, RHp, R3;        // ensure there's space
        branch_bit0_set R3, Lgarbage_collect; // otherwise invoke the GC

        load_high Linfo_table_List(R0), R3;
        store R3, -12(RHp);
        store R1,  -8(RHp);
        store R2,  -4(RHp);

        load_high Linfo_table_Ind(R0), R3;   // create an indirection
        load_address +0(R3), R3;             // to then new closure
        store R3, (RNp);
        subtract RHp, +12, R3;
        store R3, +4(RNp);

        load -12(RRet), R3;                  // invoke the regular return
        jump R3;
```

The info table and code for the cons cell is shown below:

RISC code

```
Linfo_table_List:

        dw Lupdate_List;                  // update routine
        dw Linfo_table_List;              // fast entry
        dw Linfo_table_List;              // stnd entry

        load +4(RNp), R1;                 // load head into R1
        load +8(RNp), R2;                 // load tail into R2
        load -12(RRet), R3;               // select the correct vector entry
        jump R3;                          // and return
```

Selecting the head of a list

STG′ code

```
head = [] \r [xss] -> case  xss  of { Cons x xs -> x  ; Nil -> error# [] ; };
```

Again, as for null, the code first forces the evaluation of the list, using a custom return vector:

RISC code

```
closure head    Linfo_table_head;

Linfo_table_head:

        dw Lupdate_head;                     // update routine
        dw Linfo_table_head +12;             // fast entry
        dw Linfo_table_head;                 // stnd entry

        subtract RStkA, RStkABase, R1;       // calculate the number of args
        subtract R1, +4, R1;                 // is there at least one?
        branch_x<0 R1, Lupdate_PAP;          // if not, perform an update

        load -4(RStkA), RNp;                 // fetch the arg
        load (RNp), R1;
        subtract RStkA, +4, RStkA;           // re-organise the stacks
        subtract RStkB, +4, RStkB;
```

RISC code

```
        store RRet, +4(RStkB);                // save the return register
        load_high Lhead_return_1(R0), RRet;   // set the return register
        load_address +0(RRet), RRet;
        jump R1;
```

The return vector is specified as follows:

RISC code

```
Lhead_return_1:

        dw Lupdate_List;
        dw Lhead_return_List;
        dw Lupdate_Nil;
        dw Lhead_return_Nil;                  // will cause an error!
```

The nil return simply throws an error, ending the current evaluation. However, the list return simply forces the evaluation of the head of the list:

RISC code

```
Lhead_return_List:

        load +4(RStkB), RRet;                 // recover the return reg
        add RStkB, +4, RStkB;                 // trim the stack
        move R1, RNp;                         // set the node pointer
        load (RNp), R1;                       // fetch the entry code
        jump R1;                              // evaluate the head of the list
```

Length

Rather than use the `foldl`-based version, the more traditional version is used:

STG′ code

```
length = [] \r [xss] -> case xss of
 { Nil        -> Int [0#] ;
   Cons x xs -> case length xs of { Int l -> let# l' = plusInt# [1#, l]
                                             in Int [l'] ; };
 };
```

The RISC implementation demonstrates the use of recursion and the fast-entry method (effectively skipping the argument check). The method starts as before, by evaluatin its argument:

RISC code

```
closure length              Linfo_table_length;
Linfo_table_length:

        dw Lupdate_length;                    // update routine
        dw Linfo_table_length +12;            // fast entry
        dw Linfo_table_length;                // stnd entry

        subtract RStkA, RStkABase, R1;        // calculate the number of args
        subtract R1, +4, R1;                  // is there at least one?
        branch_x<0 R1, Lupdate_PAP;           // if not, perform an update

        load -4(RStkA), RNp;                  // fetch the arg
        load (RNp), R1;
        subtract RStkA, +4, RStkA;            // re-organise the stacks
        subtract RStkB, +4, RStkB;
```

RISC code

```
        store RRet, +4(RStkB);                // save the return register
        load_high Llength_return_1(R0), RRet; // set the return register
        load_address +0(RRet), RRet;
        jump R1;
```

This time, however, both nil and non-nil entries of the return vector are used:

RISC code

```
Llength_return_1:

        dw Lupdate_List;
        dw Llength_return_List;
        dw Lupdate_Nil;
        dw Llength_return_Nil;
```

A nil-return simply returns a zero length:

RISC code

```
Llength_return_Nil:

        load +4(RStkB), RRet;                 // recover the return address
        add RStkB, +4, RStkB;                 // trim the stack
        move R0, R1;                          // set length = 0
        jump +4 RRet;                         // and return
```

A list return, however, retrieves the tail of the list and fast-calls the `length` method (bypassing the argument check and pre-loading the necessary arguments into the correct register):

RISC code

```
Llength_return_List:

        add RStkA, +4, RStkA;
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler
        store R2, -4(RStkA);                    // push the tail
        branch_link Linfo_table_length +12, RRet; // and calculate its length
        branch Lupdate_Int;                     // handle the update
```

Upon return, one is added to the tails length:

RISC code

```
        add R1, +1, R1;                         // increment the result
        load +4(RStkB), RRet;                   // recover the return address
        add RStkB, +4, RStkB;                   // trim the stack
        jump +4 RRet;                           // and return
```

Map

STG' code

```
map = [] \r [f xss] -> case xss of
 { Nil        -> Nil [] ;
   Cons x xs  -> let { x' = [f  x] \u [] -> f x ;
                      xs' = [f xs] \u [] -> map f xs ; } in Cons [x', xs'] ;
 };
```

RISC code

```
closure map                 Linfo_table_map;

Linfo_table_map:

        dw Lupdate_map;                   // update routine
        dw Linfo_table_map +12;           // fast entry
        dw Linfo_table_map;               // stnd entry

        subtract RStkA, RStkABase, R1;    // calculate the number of args
        subtract R1, +8, R1;              // are there at least two?
        branch_x<0 R1, Lupdate_PAP;       // if not, perform an update

        load -4(RStkA), RNp;              // fetch the second arg
        load (RNp), R1;
        subtract RStkA, +4, RStkA;        // re-organise the stacks
        subtract RStkB, +4, RStkB;
        store RRet, +4(RStkB);            // save the return register
        load_high Lmap_return_1(R0), RRet;    // set the return register
        load_address +0(RRet), RRet;
        jump R1;
```

The return vector is specified below:

RISC code

```
Lmap_return_1:

        dw Lupdate_List;
        dw Lmap_return_List;
        dw Lupdate_Nil;
        dw Lmap_return_Nil;
```

A nil return results in another nil return:

RISC code

```
Lmap_return_Nil:

        load +4(RStkB), RRet;              // recover the return address
        subtract RStkA, +4, RStkA;         // trim the stack
        add RStkB, +4, RStkB;              // trim the stack
        load -4(RRet), R1;                 // fetch the nil return
        jump R1;                           // and return
```

A non-nill return, however, results in the creation of the `x'` and `xs'` closures (represented by the map_1 and map_2 thunks respectively), which are then returned as a cons pair:

RISC code

```
Lmap_return_List:

        add RHp, +24, RHp;                  // allocate two closures
        compare_x<y RHLimit, RHp, R3;       // ensure there's space
        branch_bit0_set R3, Lgarbage_collect;  // otherwise invoke the GC

        load_high Linfo_table_map_1(R0), R3;   // set the info table
        load_address +0(R3), R3;
        store R3, -24(RHp);
        load -4(RStkA), R3;                 // recover f
        store R3, -20(RHp);                 // store f
        store R1, -16(RHp);                 // store x
```

RISC code

```
        load_high Linfo_table_map_2(R0), R4;   // set the info table
        load_address +0(R4), R4;
        store R4, -12(RHp);
        store R3, -8(RHp);                     // store f
        store R2, -4(RHp);                     // store xs

        load +4(RStkB), RRet;                  // recover the return vector
        subtract RStkA, +4, RStkA;             // trim the stack
        add RStkB, +4, RStkB;                  // trim the stack

        subtract RHp, +24, R1;                 // calculate x'
        subtract RHp, +12, R2;                 // calculate xs'
        load -12(RRet), R3;                    // select the list return
        jump R3;
```

The x' thunk, when evaluated, pushes an update frame, retrieves its free variables and invokes the function on the list element:

RISC code

```
Linfo_table_map_1:

        dw Lupdate_map_1;                      // update routine
        dw Linfo_table_map_1 +12;              // fast entry
        dw Linfo_table_map_1;                  // stnd entry

        subtract RStkB, +16, RStkB;            // decrease the B stack frame
        compare_x<y RStkB, RStkA, R1;          // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;   // overflow error handler
        store RStkABase, +16(RStkB);           // the A stack pointer
        store RStkBBase, +12(RStkB);           // the B stack pointer
        store RNp,        +8(RStkB);           // the node pointer
        store RRet,       +4(RStkB);           // the current return vector
        move RStkA, RStkABase;
        move RStkB, RStkBBase;
        move RUpdate, RRet;                    // set the return to an update

        add RStkA, +4, RStkA;
        compare_x<y RStkB, RStkA, R1;          // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;   // overflow error handler

        load +8(RNp), R1;                      // fetch x
        store R1, -4(RStkA);                   // push x
        load +4(RNp), RNp;                     // fetch f
        load (RNp), R1;
        jump R1;                               // enter f
```

The xs' thunk, when evaluated, pushes an update frame, retrieves its free variables and tail-calls map via its fast entry point:

RISC code

```
Linfo_table_map_2:

        dw Lupdate_map_2;                      // update routine
        dw Linfo_table_map_2 +12;              // fast entry
        dw Linfo_table_map_2;                  // stnd entry
```

RISC code

```
        subtract RStkB, +16, RStkB;            // decrease the B stack frame
        compare_x<y RStkB, RStkA, R1;          // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;   // overflow error handler
        store RStkABase, +16(RStkB);           // the A stack pointer
        store RStkBBase, +12(RStkB);           // the B stack pointer
        store RNp,        +8(RStkB);           // the node pointer
        store RRet,       +4(RStkB);           // the current return vector
        move RStkA, RStkABase;
        move RStkB, RStkBBase;
        move RUpdate, RRet;                    // set the return to an update

        add RStkA, +8, RStkA;
        compare_x<y RStkB, RStkA, R1;          // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;   // overflow error handler

        load +4(RNp), R1;                      // fetch f
        store R1, -8(RStkA);                   // push f
        load +8(RNp), R1;                      // fetch xs
        store R1, -4(RStkA);                   // push xs

        branch Linfo_table_map +12;            // tail call map
```

I.2 Generating Fibonacci numbers

Unoptimised version

The main entry point implements the following code:

STG′ code

```
fib = [] \r [n] -> case const.Int.<= n one of
 { True  -> one ;
   False -> let { sum_2_fibs = ... }
            in const.Int.+ sum_2_fibs one;
 };
```

As seen before, the standard entry point checks that there are sufficient arguments and evaluates the argument:

RISC code

```
Linfo_table_Fib:

        dw Lupdate_Fib;                       // update routine
        dw Linfo_table_Fib +12;               // fast entry
        dw Linfo_table_Fib;                   // stnd entry

        subtract RStkA, RStkABase, R1;        // calculate the number of args
        subtract R1, +4, R1;                  // is there at least one?
        branch_x<0 R1, Lupdate_Fib_PAP;       // if not, perform an update
        subtract RStkB, +4, RStkB;            // save the return pointer
        add RStkA, +8, RStkA;                 // push the args
        compare_x<y RStkB, RStkA, R1;         // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;  // overflow error handler
        store RRet, +4(RStkB);                // save the return pointer
        load -12(RStkA), R1;                  // load the argument n
        store R1, -8(RStkA);                  // push the arg n
        load_high Lone(R0), R1;               // load the closure one
        load_address +0(R1), R1;              // (low bits)
        store R1, -4(RStkA);                  // push the arg one
        branch_link Linfo_table_Int_<=, RRet; // call <= n one
        branch Lupdate_Bool;                  // handle an update request
```

The simple case is handled by the code stored at `Lfib_1`, otherwise, the `sum_2_fibs` closure is allocated (represented by the `Fib_1` thunk) and the addition operator called:

RISC code

```
        branch_x<>0 R1, Lfib_1;               // return one if it's zero

        add RHp, +8, RHp;                     // heap allocate sum_fibs
        compare_x<y RHLimit, RHp, R1;         // ensure there's space
        branch_bit0_set R1, Lgarbage_collect; // otherwise invoke the GC
        load_high Linfo_table_Fib_1(R0), R1;  // load the info-table ptr
        add R1, +0, R1;                       // (low bits)
        store R1, -8(RHp);                    // store it in the closure
        load -4(RStkA), R1;                   // recover the ptr to n
        store R1, -4(RHp);                    // store it in the closure
        load +4(RStkB), RRet;                 // recover the return ptr
        add RStkB, +4, RStkB;                 // re-allocate stack space
        add RStkA, +4, RStkA;                 // (from B to A)
        load_high Lone(R0), R1;               // load the address of one
        load_address +0(R1), R1;              // (low bits)
        store R1, -8(RStkA);                  // push one
        subtract RHp, +8, R1;                 // calculate the closure's heap
        store R1, -4(RStkA);                  // address, and push it as an arg
        branch Linfo_table_Int_+;             // add them
```

The following code handles the simple case whereby the argument is less than or equal to one, and simply returns the value one:

RISC code

```
Lfib_1:

        load +4(RStkB), RRet;                 // recover the return register
        add RStkB, +4, RStkB;                 // trim the B stack
        subtract RStkA, +4, RStkA;            // trim the A stack
        add R0, +1, R1;                       // set value to one
        jump +4 RRet;                         // return
```

The code for the `sum_2_fibs` closure implements the following STG′ code:

STG′ code

```
sum_2_fibs [n] \u [] -> let { fib_n_less_2 = [n] \u [] ...;
                              fib_n_less_1 = [n] \u [] ...; }
                        in  const.Int.+ fib_n_less_1 fib_n_less_2;
```

The code pushes an update frame, and then heap allocates the two closures before tail-calling the addition operator:

RISC code

```
Linfo_table_Fib_1:

        dw Lupdate_Fib_1;                       // update routine
        dw Linfo_table_Fib_1;                   // fast entry
        dw Linfo_table_Fib_1;                   // stnd entry

        subtract RStkB, +16, RStkB;             // decrease the B stack frame
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler
        store RStkABase, +16(RStkB);            // the A stack pointer
        store RStkBBase, +12(RStkB);            // the B stack pointer
        store RNp,        +8(RStkB);            // the node pointer
        store RRet,       +4(RStkB);            // the current return vector
        move RStkA, RStkABase;                  // clear the A stack frame
        move RStkB, RStkBBase;                  // clear the B stack frame
        move RUpdate, RRet;                     // set the return to an update
```

The fib_n_less_2 and fib_n_less_1 closures are represented by the Fib_2 and Fib_3 thunks:

RISC code

```
        add RHp, +16, RHp;                      // increase the heap pointer
        compare_x<y RHLimit, RHp, R1;           // check if there's room
        branch_bit0_set R1, Lgarbage_collect;   // otherwise, invoke the GC
        load_high Linfo_table_Fib_2(R0), R1;    // load the info-table ptr
        load_address +0(R1), R1;                // (low bits)
        store R1, -16(RHp);                     // store it in the closure
        load +4(RNp), R1;                       // load the FV n
        store R1, -12(RHp);                     // ...and store it in 1
        store R1,  -4(RHp);                     // ...and 2
        load_high Linfo_table_Fib_3(R0), R1;    // load the info-table ptr
        load_address +0(R1), R1;                // (low bits)
        store R1, -8(RHp);                      // set the closures info table
        add RStkA, +8, RStkA;                   // allocate 2 arg slots
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler
        subtract RHp, +16, R1;                  // calc the first heap addr
        store R1, -8(RStkA);                    // push fib n-1
        subtract RHp, +8, R1;                   // calc the second heap addr
        store R1, -4(RStkA);                    // push fib n-2
        branch Linfo_table_Int_+;               // and add them
```

The fib_n_less_2 closure implements the following STG′ code:

STG′ code

```
fib_n_less_2 = [n] \u [] -> let { n_less_2 = [n] \u [] -> const.Int.- n two; }
                            in fib n_less_2;
```

The corresponding RISC code pushes an update frame, heap allocates n_less_2 (represented by the Fib_4 thunk) and then tail calls fib:

RISC code

```
Linfo_table_Fib_2:

        dw Lupdate_Fib_2;                       // update routine
        dw Linfo_table_Fib_2;                   // fast entry
        dw Linfo_table_Fib_2;                   // stnd entry

        subtract RStkB, +16, RStkB;             // decrease the B stack frame
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler
        store RStkABase, +16(RStkB);            // the A stack pointer
        store RStkBBase, +12(RStkB);            // the B stack pointer
        store RNp,        +8(RStkB);            // the node pointer
        store RRet,       +4(RStkB);            // the current return vector
        move RStkA, RStkABase;
        move RStkB, RStkBBase;
        move RUpdate, RRet;                     // set the return to an update

        add RHp, +8, RHp;                       // increase the heap pointer
        compare_x<y RHLimit, RHp, R1;           // check if there's room
        branch_bit0_set R1, Lgarbage_collect;   // otherwise, invoke the GC
        load_high Linfo_table_Fib_4(R0), R1;    // load the info-table ptr
        load_address +0(R1), R1;                // (low bits)
        store R1, -8(RHp);                      // store it in the closure
        load +4(RNp), R1;                       // load the FV n
        store R1,  -4(RHp);                     // ...and 2
```

RISC code

```
        add RStkA, +4, RStkA;
        compare_x<y RStkB, RStkA, R1;            // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;     // overflow error handler
        subtract RHp, +8, R1;
        store R1, -4(RStkA);                     // push n-2
        branch Linfo_table_Fib;                  // call Fib
```

The n_less_2 closure implements the following STG′ code:

STG′ code

```
n_less_2 = [n] \u [] -> const.Int.- n two;
```

It is implemented as follows:

RISC code

```
Linfo_table_Fib_4:

        dw Lupdate_Fib_4;                        // update routine
        dw Linfo_table_Fib_4;                    // fast entry
        dw Linfo_table_Fib_4;                    // stnd entry

        subtract RStkB, +16, RStkB;              // decrease the B stack frame
        compare_x<y RStkB, RStkA, R1;            // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;     // overflow error handler
        store RStkABase, +16(RStkB);             // the A stack pointer
        store RStkBBase, +12(RStkB);             // the B stack pointer
        store RNp,        +8(RStkB);             // the node pointer
        store RRet,       +4(RStkB);             // the current return vector
        move RStkA, RStkABase;
        move RStkB, RStkBBase;
        move RUpdate, RRet;                      // set the return to an update

        add RStkA, +8, RStkA;                    // increase the A stack frame
        compare_x<y RStkB, RStkA, R1;            // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;     // overflow error handler
        load +4(RNp), R1;                        // load the ptr to n
        store R1, -8(RStkA);                     // push n
        load_high Ltwo(R0), R1;                  // load the address of two
        load_address +0(R1), R1;                 // (low bits)
        store R1, -4(RStkA);                     // push two
        branch Linfo_table_Int_-;                // calculate the difference
```

The tt fib_n_less_1 closure is sufficiently similar to tt fib_n_less_1 to not warrant inclusion here.

Optimised version

STG′ code

```
fib = [] \r [n] -> case  n  of { Int n' -> fib.wrk n'; };
```

RISC code

```
Linfo_table_Fib:

        dw Lupdate_Fib;                  // update routine
        dw Linfo_table_Fib +12;          // fast entry
        dw Linfo_table_Fib;              // stnd entry
```

RISC code

```
        subtract RStkA, RStkABase, R1;  // calculate the number of args
        subtract R1, +4, R1;            // is there at least one?
        branch_x<0 R1, Lupdate_PAP;     // if not, perform an update

        load -4(RStkA), RNp;            // load the argument n
        load (RNp), R1;                 // get the entry code

        subtract RStkA, +4, RStkA;      // free up the arg. slot,
        subtract RStkB, +4, RStkB;      // but claim the space back for
        store RRet, +4(RStkB);          // saving the return pointer

        jump_link R1, RRet;             // evaluate n

        branch Lupdate_Int;             // handle an update request

        load +4(RStkB), RRet;           // recover the return vector
        add RStkB, +4, RStkB;           // and trim the stack

        branch Linfo_table_Fib';        // tail-call fib.wrk
```

STG′ code

```
fib.wrk = [] \r [n'] -> case leInt# [n', 1#] of
 { True  -> Int [1#];
   False -> let# n'_less_1 = minusInt# [n', 1#] in
            case fib.wrk n'_less_1 of { Int fib_n'_less_1 ->
            let# n'_less_2 = minusInt# [n', 2#] in
            case fib.wrk n'_less_2 of { Int fib_n'_less_2 ->
            let# sum_2_fibs' = plusInt# [fib_n'_less_1, fib_n'_less_2] in
            let# result = plusInt# [sum_2_fibs', 1#]
            in Int [result];          }; };
 };
```

RISC code

```
Linfo_table_Fib':

        dw Lupdate_Fib';      // update routine
        dw Linfo_table_Fib';  // fast entry
        dw Linfo_table_Fib';  // stnd entry

        compare_x<=y R1, +1, R2;              // test if n' <= 1
        branch_bit0_set R2, LFib'_1;          // return one if it is

        subtract R1, +1, R2;                  // calculate n_less_one'
        subtract RStkB, +8, RStkB;            // allocate space for n'
        compare_x<y RStkB, RStkA, R3;         // check for stack overflow
        branch_bit0_set R3, Lstack_overflow;  // overflow error handler
        store R1,   +4(RStkB);                // save n'
        store RRet, +8(RStkB);                // save the return vector
        move R2, R1;

        branch_link Linfo_table_Fib', RRet;   // recursive call to Fib'
        branch Lupdate_Int;                   // handle an update request

        load +4(RStkB), R2;                   // recover fib' (n' - 1)
        add R1, R2, R1;                       // sum the two values
        add R1, +1, R1;                       // and increment
        load +8(RStkB), RRet;                 // recover the return register
        add RStkB, +8, RStkB;                 // trim the stack
        jump +4 RRet;
```

RISC code

```
LFib'_1:

        add R0, +1, R1;
        jump +4 RRet;
```

I.3 Generating prime numbers – the sieve of Eratoshenes

Unoptimised version

STG′ code

```
primes = [] \r [a] -> let { primes' = [] \u [] -> ...; } in !! primes' a;
```

RISC code

```
Linfo_table_Primes:

        dw Lupdate_Primes;                    // update routine
        dw Linfo_table_Primes +12;            // fast entry
        dw Linfo_table_Primes;                // stnd entry

        subtract RStkA, RStkABase, R1;        // calculate the number of args
        subtract R1, +4, R1;                  // is there at least one?
        branch_x<0 R1, Lupdate_PAP;           // if not, perform an update

        add RHp, +8, RHp;
        compare_x<y RHLimit, RHp, R1;         // ensure there's space
        branch_bit0_set R1, Lgarbage_collect; // otherwise invoke the GC
        load_high Linfo_table_Primes_1(R0), R1;
        load_address +0(R1), R1;
        store R1, -8(RHp);                    // set the info table;

        load -4(RStkA), R1;                   // pop a
        add RStkA, +4, RStkA;
        compare_x<y RStkB, RStkA, R2;         // check for stack overflow
        branch_bit0_set R2, Lstack_overflow;  // overflow error handler
        subtract RHp, +8, R2;                 // calculate primes
        store R2, -8(RStkA);                  // push primes
        store R1, -4(RStkA);                  // push a
        branch Linfo_table_!! +12;            // tail call !!
```

STG′ code

```
primes' = [] \u [] -> let { xs = [] \u [] -> ...; } in map head xs;
```

RISC code

```
Linfo_table_Primes_1:

        dw Lupdate_Primes_1;                    // update routine
        dw Linfo_table_Primes_1 +12;            // fast entry
        dw Linfo_table_Primes_1;                // stnd entry

        subtract RStkB, +16, RStkB;             // decrease the B stack frame
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler
        store RStkABase, +16(RStkB);            // the A stack pointer
        store RStkBBase, +12(RStkB);            // the B stack pointer
        store RNp,        +8(RStkB);            // the node pointer
        store RRet,       +4(RStkB);            // the current return vector
```

RISC code

```
        move RStkA, RStkABase;
        move RStkB, RStkBBase;
        move RUpdate, RRet;                     // set the return to an update

        add RHp, +8, RHp;
        compare_x<y RHLimit, RHp, R1;           // ensure there's space
        branch_bit0_set R1, Lgarbage_collect;   // otherwise invoke the GC
        load_high Linfo_table_Primes_2(R0), R1;
        load_address +0(R1), R1;
        store R1, -8(RHp);                      // set the info table;

        add RStkA, +8, RStkA;
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler
        load_high Lhead(R0), R1;
        load_address +0(R1), R1;
        store R1, -8(RStkA);                    // push take
        subtract RHp, +8, R1;                   // calculate xs
        store R1, -4(RStkA);                    // push xs
        branch Linfo_table_map +12;             // tail call map
```

STG′ code

```
xs = [] \u [] -> let { from_2 = [] \u [] -> iterate inc two;}
                 in  iterate the_filter from_2; }
```

RISC code

```
Linfo_table_Primes_2:

        dw Lupdate_Primes_2;                    // update routine
        dw Linfo_table_Primes_2 +12;            // fast entry
        dw Linfo_table_Primes_2;                // stnd entry

        subtract RStkB, +16, RStkB;             // decrease the B stack frame
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler
        store RStkABase, +16(RStkB);            // the A stack pointer
        store RStkBBase, +12(RStkB);            // the B stack pointer
        store RNp,        +8(RStkB);            // the node pointer
        store RRet,       +4(RStkB);            // the current return vector
        move RStkA, RStkABase;
        move RStkB, RStkBBase;
        move RUpdate, RRet;                     // set the return to an update

        add RHp, +8, RHp;
        compare_x<y RHLimit, RHp, R1;           // ensure there's space
        branch_bit0_set R1, Lgarbage_collect;   // otherwise invoke the GC
        load_high Linfo_table_Primes_3(R0), R1;
        load_address +0(R1), R1;
        store R1, -8(RHp);                      // set the info table;

        add RStkA, +8, RStkA;
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler
        load_high Lthe_filter(R0), R1;
        load_address +0(R1), R1;
        store R1, -8(RStkA);                    // push the filter
        subtract RHp, +8, R1;                   // calculate from_2
        store R1, -4(RStkA);                    // push from_2
        branch Linfo_table_iterate +12;         // tail call iterate
```

STG′ code

```
from_2 = [] \u [] -> iterate inc two
```

RISC code

```
Linfo_table_Primes_3:

        dw Lupdate_Primes_3;                    // update routine
        dw Linfo_table_Primes_3 +12;            // fast entry
        dw Linfo_table_Primes_3;                // stnd entry

        subtract RStkB, +16, RStkB;             // decrease the B stack frame
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler
        store RStkABase, +16(RStkB);            // the A stack pointer
        store RStkBBase, +12(RStkB);            // the B stack pointer
        store RNp,       +8(RStkB);             // the node pointer
        store RRet,      +4(RStkB);             // the current return vector
        move RStkA, RStkABase;
        move RStkB, RStkBBase;
        move RUpdate, RRet;                     // set the return to an update

        add RStkA, +8, RStkA;
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler
        load_high Linc(R0), R1;
        load_address +0(R1), R1;
        store R1, -8(RStkA);                    // push inc
        load_high Ltwo(R0), R1;
        load_address +0(R1), R1;
        store R1, -4(RStkA);                    // push two
        branch Linfo_table_iterate +12;         // tail call iterate
```

STG′ code

```
the_filter = [] \r [nss] -> case nss of { Cons n ns ->
 let { isdivs_n =  [n] \r [x] -> isdivs n x; } in filter isdivs_n ns; };
```

RISC code

```
Linfo_table_The_Filter:


        dw Lupdate_The_Filter;                  // update routine
        dw Linfo_table_The_Filter +12;          // fast entry
        dw Linfo_table_The_Filter;              // stnd entry

        subtract RStkA, RStkABase, R1;          // calculate the number of args
        subtract R1, +4, R1;                    // is there at least one?
        branch_x<0 R1, Lupdate_PAP;             // if not, perform an update

        load -4(RStkA), RNp;                    // fetch the arg
        load (RNp), R1;
        subtract RStkA, +4, RStkA;              // re-organise the stacks
        subtract RStkB, +4, RStkB;
        store RRet, +4(RStkB);                  // save the return register
        load_high LThe_Filter_return_1(R0), RRet;
        load_address +0(RRet), RRet;            // set the return register
        jump R1;
```

RISC code

```
LThe_Filter_return_1:

        dw Lupdate_List;
        dw LThe_Filter_return_List;
        dw Lupdate_Nil;
        dw LThe_Filter_return_Nil_error;
```

RISC code

```
LThe_Filter_return_List:

        add RHp, +8, RHp;
        compare_x<y RHLimit, RHp, R3;           // ensure there's space
        branch_bit0_set R3, Lgarbage_collect;   // otherwise invoke the GC
        load_high Linfo_table_The_Filter_1(R0), R3;
        load_address +0(R3), R3;
        store R3, -8(RHp);                      // set the info table;
        store R1, -4(RHp);                      // store n

        load +4(RStkB), RRet;                   // recover the return vector
        add RStkA, +8, RStkA;                   // allocate stack space
        add RStkB, +4, RStkB;
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler

        subtract RHp, +8, R1;                   // calculate isdivs_n
        store R1, -8(RStkA);                    // push isdivs_n
        store R2, -4(RStkA);                    // push ns
        branch Linfo_table_filter +12;          // tail call filter
```

STG′ code

```
isdivs_n = [n] \r [x] -> isdivs n x
```

RISC code

```
Linfo_table_The_Filter_1:


        dw Lupdate_The_Filter_1;                // update routine
        dw Linfo_table_The_Filter_1 +12;        // fast entry
        dw Linfo_table_The_Filter_1;            // stnd entry

        subtract RStkA, RStkABase, R1;          // calculate the number of args
        subtract R1, +4, R1;                    // is there at least one?
        branch_x<0 R1, Lupdate_PAP;             // if not, perform an update
        load -4(RStkA), R1;                     // pop x
        add RStkA, +4, RStkA;
        compare_x<y RStkB, RStkA, R2;           // check for stack overflow
        branch_bit0_set R2, Lstack_overflow;    // overflow error handler
        load +4(RNp), R2;                       // fetch n
        store R2, -8(RStkA);                    // push n
        store R1, -4(RStkA);                    // push x
        branch Linfo_table_isdivs +12;          // tail call is_divs
```

STG′ code

```
isdivs = [] \r [n x] -> let { mod_x_n =  [n x] \u [] -> const.Int.mod x n; }
                        in  const.Int./= mod_x_n zero;
```

RISC code

```
Linfo_table_isdivs:

        dw Lupdate_isdivs;                      // update routine
        dw Linfo_table_isdivs +12;              // fast entry
        dw Linfo_table_isdivs;                  // stnd entry

        subtract RStkA, RStkABase, R1;          // calculate the number of args
        subtract R1, +8, R1;                    // are there at least two?
        branch_x<0 R1, Lupdate_PAP;             // if not, perform an update

        add RHp, +12, RHp;
        compare_x<y RHLimit, RHp, R1;           // ensure there's space
        branch_bit0_set R1, Lgarbage_collect;   // otherwise invoke the GC
        load_high Linfo_table_isdivs_1(R0), R1;
        load_address +0(R1), R1;
        store R1, -12(RHp);                     // set the info table;
        load -8(RStkA), R1;                     // pop n
        store R1,  -8(RHp);                     // store n
        load -4(RStkA), R1;                     // pop x
        store R1,  -4(RHp);                     // store x
        subtract RHp, +12, R1;                  // calculate mod_x_n
        store R1, -8(RStkA);                    // push mod_x_n
        load_high Lzero(R0), R1;                // load zero
        load_address +0(R1), R1;
        store R1, -4(RStkA);
        branch Linfo_table_Int_/= +12;          // tail call /=
```

STG′ code

```
mod_x_n =  [n x] \u [] -> const.Int.mod x n;
```

RISC code

```
Linfo_table_isdivs_1:

        dw Lupdate_isdivs_1;                    // update routine
        dw Linfo_table_isdivs_1 +12;            // fast entry
        dw Linfo_table_isdivs_1;                // stnd entry


        subtract RStkB, +16, RStkB;             // decrease the B stack frame
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler
        store RStkABase, +16(RStkB);            // the A stack pointer
        store RStkBBase, +12(RStkB);            // the B stack pointer
        store RNp,        +8(RStkB);            // the node pointer
        store RRet,       +4(RStkB);            // the current return vector
        move RStkA, RStkABase;
        move RStkB, RStkBBase;
        move RUpdate, RRet;                     // set the return to an update

        add RStkA, +8, RStkA;                   // allocate stack space
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler

        load +8(RNp), R1;                       // fetch x
        store R1, -8(RStkA);                    // push x
        load +4(RNp), R1;                       // fetch n
        store R1, -4(RStkA);                    // push n
        branch Linfo_table_Int_mod +12;         // tail call mod
```

STG′ code

```
succ = [] \r [x] ->
 case x of { Int x' -> let# succ_x = plusInt# [x', 1#] in Int [succ_x]; };
```

RISC code

```
Linfo_table_inc:

        dw Lupdate_Int_inc;               // update routine
        dw Linfo_table_inc +12;           // fast entry
        dw Linfo_table_inc;               // stnd entry

        subtract RStkA, RStkABase, R1;    // calculate the number of args
        subtract R1, +4, R1;              // is there at least one?
        branch_x<0 R1, Lupdate_PAP;       // if not, perform an update
        load  -4(RStkA), RNp;             // load the node pointer of arg
        subtract RStkA, +4, RStkA;        // trim the A stack
        subtract RStkB, +4, RStkB;        // trim the B stack
        store RRet, +4(RStkB);            // save the return pointer
        load (RNp), R1;                   // get the info table of arg
        jump_link R1, RRet;               // enter the closure
        branch Lupdate_Int;               // handle an update request

        add R1, +1, R1;                   // increase the value
        load +4(RStkB), RRet;             // recover the return register
        add RStkB, +4, RStkB;             // trim the B stack
        jump +4 RRet;                     // and return normally
```

Optimised version

STG′ code

```
primes = [] \r [a] -> case a of { Int a' -> primes.wrk a'; };
```

RISC code

```
Linfo_table_Primes:

        dw Lupdate_Primes;                    // update routine
        dw Linfo_table_Primes +12;            // fast entry
        dw Linfo_table_Primes;                // stnd entry

        subtract RStkA, RStkABase, R1;        // calculate the number of args
        subtract R1, +4, R1;                  // is there at least one?
        branch_x<0 R1, Lupdate_PAP;           // if not, perform an update

        load -4(RStkA), RNp;                  // fetch n
        load (RNp), R1;
        subtract RStkA, +4, RStkA;            // re-organise the stacks
        subtract RStkB, +4, RStkB;
        store RRet, +4(RStkB);                // save the return register
        jump_link R1, RRet;                   // evaluate n
        branch Lupdate_Int;                   // handle the update

        load +4(RStkB), RRet;                 // recover the return reg
        store R1, +4(RStkB);                  // push n'
        branch Lprimes.wrk + 12;              // tail call the wrapper
```

STG′ code

```
primes.wrk = [] \r [a'] -> let { from_2 =  [] \u [] ->  iterate inc two; } in
                         letstrict forced_xs = iterate the_filter from_2  in
                         letstrict forced_primes = map head forced_xs
                         in !!.wrk forced_primes a';
```

RISC code

```
Lprimes.wrk:

        subtract RStkBBase, RStkB, R1;        // calculate the number of args
        subtract R1, +4, R1;                  // is there at least one?
        branch_x<0 R1, Lupdate_PAP;           // if not, perform an update

        add RHp, +8, RHp;
        compare_x<y RHLimit, RHp, R1;         // ensure there's space
        branch_bit0_set R1, Lgarbage_collect; // otherwise invoke the GC
        load_high Linfo_table_Primes_1(R0), R1;
        load_address +0(R1), R1;
        store R1, -8(RHp);                    // set the info table;

        add RStkA, +8, RStkA;
        subtract RStkB, +4, RStkB;
        compare_x<y RStkB, RStkA, R1;         // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;  // overflow error handler
        load_high Lthe_filter(R0), R1;
        load_address +0(R1), R1;
        store R1, -8(RStkA);                  // push the filter
        subtract RHp, +8, R1;                 // calculate from_2
        store R1, -4(RStkA);                  // push from_2
        store RRet, +4(RStkB);                // save the return vector
        load_high Lprimes_return_1(R0), RRet; // set the return register
        load_address +0(RRet), RRet;
        branch Linfo_table_iterate +12;       // tail call iterate
```

RISC code

```
Lprimes_return_1:

        dw Lupdate_List;
        dw Lprimes_return_List_1;
        dw Lupdate_Nil;
        dw Lprimes_return_Nil_1;
```

RISC code

```
Lprimes_return_Nil_1:

        add RHp, +8, RHp;
        compare_x<y RHLimit, RHp, R1;         // ensure there's space
        branch_bit0_set R1, Lgarbage_collect; // otherwise invoke the GC
        load_high Linfo_table_Nil(R0), R1;    // nil info table
        store R1, -8(RHp);
        subtract RHp, +8, R1;                 // calculate forced_xs
        branch Lprimes_join_1;
```

RISC code

```
Lprimes_return_List_1:

        add RHp, +12, RHp;
        compare_x<y RHLimit, RHp, R3;         // ensure there's space
        branch_bit0_set R3, Lgarbage_collect; // otherwise invoke the GC
        load_high Linfo_table_List(R0), R3;   // list info table
        store R3, -12(RHp);
        store R1,  -8(RHp);                   // store x
        store R2,  -4(RHp);                   // store xs
        subtract RHp, +12, R1;                // calculate forced_xs
        branch Lprimes_join_1;
```

RISC code

```
Lprimes_join_1:

        add RStkA, +8, RStkA;                 // know there are min. 2 slots
        load_high Lhead(R0), R2;
        load_address +0(R2), R2;
        store R2, -8(RStkA);                  // push head
        store R1, -4(RStkA);                  // push forced_xs
        load_high Lprimes_return_2(R0), RRet; // set the return register
        load_address +0(RRet), RRet;
        branch Linfo_table_map +12;           // tail call map
```

RISC code

```
Lprimes_return_2:

        dw Lupdate_List;
        dw Lprimes_return_List_2;
        dw Lupdate_Nil;
        dw Lprimes_return_Nil_2;
```

RISC code

```
Lprimes_return_Nil_2:

        add RHp, +8, RHp;
        compare_x<y RHLimit, RHp, R1;         // ensure there's space
        branch_bit0_set R1, Lgarbage_collect; // otherwise invoke the GC
        load_high Linfo_table_Nil(R0), R1;    // nil info table
        store R1, -8(RHp);
        subtract RHp, +8, R1;                 // calculate forced_primes
        branch Lprimes_join_2;
```

RISC code

```
Lprimes_return_List_2:

        add RHp, +12, RHp;
        compare_x<y RHLimit, RHp, R3;         // ensure there's space
        branch_bit0_set R3, Lgarbage_collect; // otherwise invoke the GC
        load_high Linfo_table_List(R0), R3;   // list info table
        store R3, -12(RHp);
        store R1,  -8(RHp);                   // store x
        store R2,  -4(RHp);                   // store xs
        subtract RHp, +12, R1;                // calculate forced_primes
        branch Lprimes_join_2;
```

RISC code

```
Lprimes_join_2:

        load +4(RStkB), RRet;                 // recover the return register

        add RStkB, +4, RStkB;                 // re-allocate stack space
        add RStkA, +4, RStkA;
        store R1, -4(RStkA);                  // push forced_primes
        branch L!!.wrk +24;                   // tail call !!
```

STG′ code

```
from_2 =  [] \u [] ->  iterate inc two;
```

RISC code

```
Linfo_table_Primes_1:

        dw Lupdate_Primes_1;                    // update routine
        dw Linfo_table_Primes_1 +12;            // fast entry
        dw Linfo_table_Primes_1;                // stnd entry

        subtract RStkB, +16, RStkB;             // decrease the B stack frame
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler
        store RStkABase, +16(RStkB);            // the A stack pointer
        store RStkBBase, +12(RStkB);            // the B stack pointer
        store RNp,       +8(RStkB);             // the node pointer
        store RRet,      +4(RStkB);             // the current return vector
        move RStkA, RStkABase;
        move RStkB, RStkBBase;
        move RUpdate, RRet;                     // set the return to an update

        add RStkA, +8, RStkA;
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler
        load_high Linc(R0), R1;
        load_address +0(R1), R1;
        store R1, -8(RStkA);                    // push inc
        load_high Ltwo(R0), R1;
        load_address +0(R1), R1;
        store R1, -4(RStkA);                    // push two
        branch Linfo_table_iterate +12;         // tail call iterate
```

STG′ code

```
the_filter = [] \r [nss] -> case nss of { Cons n ns ->
 let { isdivs_n = [n] \r [x] -> ...; } in filter isdivs_n ns; };
```

RISC code

```
Linfo_table_The_Filter:

        dw Lupdate_The_Filter;                  // update routine
        dw Linfo_table_The_Filter +12;          // fast entry
        dw Linfo_table_The_Filter;              // stnd entry

        subtract RStkA, RStkABase, R1;          // calculate the number of args
        subtract R1, +4, R1;                    // is there at least one?
        branch_x<0 R1, Lupdate_PAP;             // if not, perform an update

        load -4(RStkA), RNp;                    // fetch the arg
        load (RNp), R1;
        subtract RStkA, +4, RStkA;              // re-organise the stacks
        subtract RStkB, +4, RStkB;
        store RRet, +4(RStkB);                  // save the return register
        load_high LThe_Filter_return_1(R0), RRet;
        load_address +0(RRet), RRet;            // set the return register
        jump R1;
```

RISC code

```
LThe_Filter_return_1:

        dw Lupdate_List;
        dw LThe_Filter_return_List;
        dw Lupdate_Nil;
        dw LThe_Filter_return_Nil_error;
```

RISC code

```
LThe_Filter_return_List:

        add RHp, +8, RHp;
        compare_x<y RHLimit, RHp, R3;         // ensure there's space
        branch_bit0_set R3, Lgarbage_collect;   // otherwise invoke the GC
        load_high Linfo_table_The_Filter_1(R0), R3;
        load_address +0(R3), R3;
        store R3, -8(RHp);                      // set the info table;
        store R1, -4(RHp);                      // store n

        load +4(RStkB), RRet;                   // recover the return vector
        add RStkA, +8, RStkA;                   // allocate stack space
        add RStkB, +4, RStkB;
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler

        subtract RHp, +8, R1;                   // calculate isdivs_n
        store R1, -8(RStkA);                    // push isdivs_n
        store R2, -4(RStkA);                    // push ns
        branch Linfo_table_filter +12;          // tail call filter
```

STG′ code

```
isdivs_n = [n] \r [x] -> case n of { Int n' ->
                         case x of { Int x' -> isdivs.wrk n' x'; }; }
```

RISC code

```
Linfo_table_The_Filter_1:


        dw Lupdate_The_Filter_1;                // update routine
        dw Linfo_table_The_Filter_1 +12;        // fast entry
        dw Linfo_table_The_Filter_1;            // stnd entry

        subtract RStkA, RStkABase, R1;          // calculate the number of args
        subtract R1, +4, R1;                    // is there at least one?
        branch_x<0 R1, Lupdate_PAP;             // if not, perform an update

        load +4(RNp), RNp;                      // recover n
        subtract RStkB, +4, RStkB;
        compare_x<y RStkB, RStkA, R1;           // check for stack overflow
        branch_bit0_set R1, Lstack_overflow;    // overflow error handler
        store RRet, +4(RStkB);                  // save the return
        load (RNp), R1;
        jump_link R1, RRet;                     // evaluate n
        branch Lupdate_Int;

        load -4(RStkA), RNp;                    // recover x
        subtract RStkA, +4, RStkA;              // re-allocate stack space
        subtract RStkB, +4, RStkB;
        store R1, +4(RStkB);                    // save n'
        load (RNp), R1;
        jump_link R1, RRet;                     // evaluate x
        branch Lupdate_Int;

        load +8(RStkB), RRet;                   // recover return vector
        load +4(RStkB), R2;
        store R2, +8(RStkB);                    // push n'
        store R1, +4(RStkB);                    // push x'
        branch Lisdivs.wrk +12;                 // tail call isdivs.wrk
```

STG′ code

```
isdivs = [] \r [n x] ->
 case n of { Int n’ -> case x of { Int x’ -> isdivs.wrk n’ x’; }; };
```

RISC code

```
Linfo_table_isdivs:

        dw Lupdate_isdivs;                      // update routine
        dw Linfo_table_isdivs +12;              // fast entry
        dw Linfo_table_isdivs;                  // stnd entry

        subtract RStkA, RStkABase, R1;          // calculate the number of args
        subtract R1, +8, R1;                    // are there at least two?
        branch_x<0 R1, Lupdate_PAP;             // if not, perform an update

        load -8(RStkA), RNp;                    // pop n
        load -4(RStkA), R1;                     // pop x
        subtract RStkA, +4, RStkA;              // re-allocate stack space
        subtract RStkB, +4, RStkB;
        store RRet, +4(RStkB);                  // save the return vector
        store R1, -4(RStkA);                    // save x
        load (RNp), R1;
        jump_link R1, RRet;                     // evaluate n
        branch Lupdate_Int;

        load -4(RStkA), RNp;                    // recover x
        subtract RStkA, +4, RStkA;              // re-allocate stack space
        subtract RStkB, +4, RStkB;
        store R1, +4(RStkB);                    // save n’
        load (RNp), R1;
        jump_link R1, RRet;                     // evaluate x
        branch Lupdate_Int;

        load +8(RStkB), RRet;                   // recover return vector
        load +4(RStkB), R2;
        store R2, +8(RStkB);                    // push n’
        store R1, +4(RStkB);                    // push x’
        branch Lisdivs.wrk +12;                 // tail call isdivs.wrk
```

STG′ code

```
isdivs.wrk = [] \r [n’ x’] -> case const.Int.mod.wrk x’ n’ of { Int mod’ ->
                              case mod’ of { 0# -> False [];
                                             _  -> True  [] };
                                           };
```

RISC code

```
Lisdivs.wrk:

        subtract RStkBBase, RStkB, R1;        // calculate the number of args
        subtract R1, +8, R1;                  // are there at least two?
        branch_x<0 R1, Lupdate_PAP;           // if not, perform an update

        load +4(RStkB), R1;                   // pop x’
        load +8(RStkB), R2;                   // pop n’
        add RStkB, +8, RStkB;
        remainder R1, R2, R1;                 // calculate mod x’ n’
        branch_x=0 R1, Lisdivs.wrk_1;         // if mod == 0 return false
        add R0, +1, R1;                       // return true
        jump +4 RRet;
```

RISC code

```
Lisdivs.wrk_1:

        move R0, R1;
        jump +4 RRet;                           // return false
```

I.4 Updating algebraic constructors

The following return vector is suitable for updating polymorphic expressions, and will catch and correctly handle all forms of algebraic constructors:

RISC code

```
Lupdate_constr:

        dw Lupdate_vector_8_chained;
        dw Lupdate_vector_8;
        dw Lupdate_vector_7_chained;
        dw Lupdate_vector_7;
        dw Lupdate_vector_6_chained;
        dw Lupdate_vector_6;
        dw Lupdate_vector_5_chained;
        dw Lupdate_vector_5;
        dw Lupdate_vector_4_chained;
        dw Lupdate_vector_4;
        dw Lupdate_vector_3_chained;
        dw Lupdate_vector_3;
        dw Lupdate_vector_2_chained;
        dw Lupdate_vector_2;
        dw Lupdate_vector_1_chained;
        dw Lupdate_vector_1;

        branch Lupdate_vector_0_chained;
        load  +4(RStkB), RRet;              // recover the return ptr
        load  +8(RStkB), RNp;               // recover the node pointer
        load +12(RStkB), RStkBBase;         // recover the B stack frame
        load +16(RStkB), RStkABase;         // recover the A stack frame
        add RStkB, +16, RStkB;              // pop the update fram
        jump RRet;                          // invoke the 'update' return
```

RISC code

```
Lupdate_vector_0_chained:

        load  +8(RStkB), RMisc;              // recover the node pointer
        load_high Linfo_table_Ind(R0), RRet; // overwrite the existing closure
        load_address (RRet), RRet;           // with an indirection
        store RRet, (RMisc);                 // store the address of the node
        store RNp, +4(RMisc);                // pointer which will be updated

        load  +4(RStkB), RRet;              // recover the return ptr
        load +12(RStkB), RStkBBase;         // recover the B stack frame
        load +16(RStkB), RStkABase;         // recover the A stack frame
        add RStkB, +16, RStkB;              // pop the update frame

        jump RRet;                          // invoke the 'update' return
```

RISC code

```
Lupdate_vector_1:

        load  +4(RStkB), RRet;             // recover the return ptr
        load  +8(RStkB), RNp;              // recover the node pointer
        load +12(RStkB), RStkBBase;        // recover the B stack frame
        load +16(RStkB), RStkABase;        // recover the A stack frame
        add RStkB, +16, RStkB;             // pop the update fram
        load -8(RRet), RMisc;              // select the correct vector
        jump RMisc;                        // invoke the 'update' return
```

RISC code

```
Lupdate_vector_1_chained:

        load  +8(RStkB), RMisc;              // recover the node pointer
        load_high Linfo_table_Ind(R0), RRet; // overwrite the existing closure
        load_address (RRet), RRet;           // with an indirection
        store RRet, (RMisc);                 // store the address of the node
        store RNp, +4(RMisc);                // pointer which will be updated

        load  +4(RStkB), RRet;             // recover the return ptr
        load +12(RStkB), RStkBBase;        // recover the B stack frame
        load +16(RStkB), RStkABase;        // recover the A stack frame
        add RStkB, +16, RStkB;             // pop the update frame

        load -8(RRet), RMisc;
        jump RMisc;                        // invoke the 'update' return
```

I.5 Updating partial applications

RISC code

```
Lupdate_PAP:

        load +4(RStkBBase), RRet;               // recover the return reg
        load +8(RStkBBase), R1;                 // recover the node pointer
        load_high Linfo_table_Ind(R0), R2;      // and overwrite it with an
        load_address +0(R2), R2;                // indirection to the PAP
        store R2, (R1);
        store RHp, +4(R1);

        subtract RStkA, RStkABase, R1;          // number of A args on stack
        subtract RStkBBase, RStkB, R2;          // number of B args on stack

        add R1, R2, R3;                         // total no of args
        add R3, +16, R3;                        // factor in 4 extra words
        add RHp,  R3, RHp;                      // allocate R3 words
        compare_x<y RHLimit, RHp, R5;           // ensure there's space
        branch_bit0_set R5, Lgarbage_collect;   // otherwise invoke the GC

        subtract RHp, R3, R4;                   // calculate address of the PAP
        load_high Linfo_table_PAP(R0), R5;      // create the PAP
        load_address +0(R5), R5;
        store R5,  +0(R4);                      // set the info table
        store RNp, +4(R4);                      // save the node pointer
        store R1,  +8(R4);                      // save the no A args
        store R2, +12(R4);                      // save the no B args
```

RISC code

```
        add R4, +16, R4;                        // set R4 = heap ptr
        move RStkABase, R5;                     // set R5 = stack ptr
        load +16(RStkBBase), RStkABase;         // recover the A stack limit

        branch_x=0 R1, +6;                      // skip forward
        load (R5), R6;                          // get the first A entry
        store R6, (R4);                         // store it in the closure
        add R5, +4, R5;                         // increase stk ptr
        add R4, +4, R4;                         // increase the stk ptr
        subtract R1, +4, R1;                    // decrement the count
        branch_x>0 R1, -6;                      // re-enter the loop

        move RStkBBase, R5;                     // set R5 = stack ptr
        load +12(RStkBBase), RStkBBase;         // recover RStackBase
        add RStkB, +16, RStkB;                  // pop the update frame

        branch_x=0 R2, +7;                      // possibly skip forward

        load (R5), R7;                          // get the first B entry
        store R7, (R4);                         // store it in the closure
        store R7, +16(R5);                      // slide it down the stack
        add R4, +4, R4;                         // increase the stk ptr
        subtract R5, +4, R5;                    // increase stk ptr
        subtract R2, +4, R2;                    // decrement the count
        branch_x>0 R2, -7;                      // re-enter the loop

        load (RNp), R1;                         // re-enter the function
        jump R1;
```

The info table for a partial application is shown below:

RISC code

```
Linfo_table_PAP:

        dw Lupdate_PAP_closure;                 // update routine
        dw Linfo_table_PAP;                     // fast entry
        dw Linfo_table_PAP;                     // stnd entry

        load  +8(RNp), R1;                      // fetch the no of A args
        load +12(RNp), R2;                      // fetch the no of B args
        move RStkA, R3;                         // save RStkA
        move RStkB, R4;                         // and RStkB
        add RStkA, R1, RStkA;                   // increase the stacks
        subtract RStkB, R2, RStkB;              //
        compare_x<y RStkB, RStkA, R5;           // check for stack overflow
        branch_bit0_set R5, Lstack_overflow;    // overflow error handler

        add RNp, +16, R5;                       // set the closure ptr
        branch_x=0 R1, +6;

        load (R5), R6;                          // recover the argument
        store R6, (R3);                         // push the argument
        add R3, +4, R3;                         // advance the stk ptr
        add R5, +4, R5;                         // advance the closure ptr
        subtract R1, +4, R1;                    // reduce the no words
        branch_x>0 R1, -6;                      // and repeat
```

RISC code

```
    branch_x=0 R2, +6;

    load (R5), R6;                          // fetch the arg
    store R6, (R4);                         // push the arg
    subtract R4, +4, R4;                    // decrease the stack ptr
    add R5, +4, R5;                         // advance the closure ptr
    subtract R2, +4, R2;
    branch_x>0 R2, -6;                      // and repeat

    load +4(RNp), RNp;                      // recover the function pointer
    load (RNp), R1;                         // fetch its info table
    jump R1;                                // and call it
```

References

Aarts, E. H. L., van Leeuwen, J., and Rem, M. (Eds.). (1991a). *PARLE '91: 3rd International Conference on Parallel Architectures and Languages Europe, 10–13 June, Eidhoven, Netherlands, volume I (Parallel Architectures and Algorithms)*, number 505 in Lecture Notes in Computer Science. Springer-Verlag.

Aarts, E. H. L., van Leeuwen, J., and Rem, M. (Eds.). (1991b). *PARLE '91: 3rd International Conference on Parallel Architectures and Languages Europe, 10–13 June, Eidhoven, Netherlands, volume II (Parallel Languages)*, number 506 in Lecture Notes in Computer Science. Springer-Verlag.

Addison, C., Gee, D., Getov, V., Hey, T., Hockney, R., et al. (1993). The Genesis distributed memory benchmarks. part 1: Methodology and general relativity benchmark with results for the SUPERNUM computer. *Concurrency: Practice and Experience*, 5(1):1–22.

Aditya, S., Arvind, Hicks, J., Maessen, J.-W., Augustsson, L., and Nikhil, R. S. (1995). Semantics of pH: A parallel dialect of Haskell. Technical Report Computation Structures Group Memo 377-1, Massachusetts Institute of Technology.

Agarwal, A., Chaiken, D., D'Souza, G., Johnson, K., Kranz, D., et al. (1991). The MIT Alewife machine: A large-scale distributed-memory multiprocessor. Technical Report MIT/LCS Technical Memo 454, Massachusetts Institute of Technology.

Agarwal, A., Kubiatowicz, J., Kranz, D., Lim, B.-H., Yeung, D., et al. (1993). Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, 13(3).

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company.

Allison, L. (1983). Programming denotational semantics. *The Computer Journal*, 26(2):164–173.

Almasi, G. and Gottlieb, A. (1993). *Highly Parallel Computing* (second ed.). Benjamin/Cummings.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computer capabilities. In *AFIPS Spring Joint Computer Conference.*

Andrews, G. R. (1991). Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):49–90.

Appel, A. W. (1992). *Compiling with Continuations.* Cambridge University Press.

Appel, A. W. and Jim, T. (1990). Continuation-passing, closure-passing style. In [POPL '90, 1990].

AQUA Team (1993). *The Glorious Haskell Compilation System User's Guide.*

AQUA Team (1994). The Glasgow Haskell compiler – version 0.23. URL ftp://ftp.dcs.-glasgow.ac.uk/pub/haskell/glasgow/ghc-0.23-src.tar.gz.

Ariola, Z. and Arvind (Eds.). (1989). *FPCA '89: 4th Conference on Functional Programming Languages and Computer Architecture, 11–13 September, London, England.* ACM Press.

Augustsson, L. and Johnsson, T. (1989). Parallel graph reduction with the $\langle v, G \rangle$-Machine. In [Ariola and Arvind, 1989], (pp. 202–213).

Axford, T. (1989). *Concurrent Programming: Fundamental Techniques for Real-Time and Parallel Software Design.* Parallel Computing. John Wiley and Sons.

Axford, T. H. and Joy, M. S. (1991). List processing in parallel. Technical Report CSR-91-8, University of Birmingham, School of Computer Science, Birmingham. B15 2TT.

Backus, J. (1978). Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641.

Bailey, D. (1991). Twelve ways to fool the masses when giving performance results on parallel computers. *Supercomputing Review*, 4(8):54–55.

Bailey, D. (94). Nas parallel benchmarks. Technical Report 94-007, NAS Applied Research Branch (RNR).

Bala, V., Ferrante, J., and Carter, L. (1993). Explicit data placement (XDP): A methodology for explicit compile-time representation and optimization of data movement. In [Chen and Halstead, 1993], (pp. 139–148).

Balzer, R., Gabriel, R. P., Belz, F., Dewar, R., Fisher, D., et al. (1988). Draft report on requirements for a common prototyping system. *ACM SIGPLAN Notices*, 24(3).

Barendregt, H., van Eekelen, M., Kennaway, J., et al. (1987). Term graph rewriting. In [de Bakker, Nijman and Treleaven, 1987].

Barendregt, H. P. (1981). *The Lambda Calculus – It's Syntax and Semantics*, volume 103 of *Studies in logic and the foundations of mathematics.* North Holland Publishing Company.

Barth, P. S., Nikhil, R. S., and Arvind (1991). M-structures: Extending a parallel, non-strict functional language with state. In [Hughes, 1991].

Bartlett, J. F. (1989). Scheme-¿ c a portable scheme-to-c compiler. Technical Report DEC-WRL-89-1, Digital Equipment Corporation, Western Research Lab.

Bedichek, R. C. (1995). Talisman: Fast and accurate multicomputer simulation. In [SIGMETRICS '95, 1995], (pp. 14–24).

Beemster, M. (1994). Strictness optimization for graph reduction machines (why id might not be strict). *ACM Transactions on Programming Languages and Systems*, 16(5):1449–1466.

Beguelin, A., Dongarra, J., Geist, A., and et al, R. M. (1993). PVM 3 user's guide. Technical Report ORNL/TM-12187, Oak Ridge National Lanoratory.

Ben-Dyke, A. D. (1997). GVT algorithms. Technical report, DRA Malvern.

Ben-Dyke, A. D. and Axford, T. (1995). On the design of parallel functional intermediate languages. In [Sahni, Prasanna and Bhatkar, 1995], (pp. 673–678).

Bennett, A. J. (1993). *Parallel Graph Reduction for Shared-Memory Architectures.* PhD thesis, Department of Computing, Imperial College, London, U. K.

Bernstein, R. L. (1985). Producing good code for the case statement (short communication). *Software – Practise and Experience*, 15(10):1021–1024.

Bird, R. and Wadler, P. (1988). *Introduction to Functional Programming.* Prentice Hall International.

Birrell, A. D. (1989). An introduction to programming with threads. Technical Report SRC Research Report No. 35, Digital Equipment Corp.

Blelloch, G. E., Chatterjee, S., Hardwick, J. C., Sipelstein, J., and Zagha, M. (1993). Implementation of a portable nested data-parallel language. In [Chen and Halstead, 1993], (pp. 102–111).

Bloss, A. and Hudak, P. (1988). Path semantics. In *Mathematical Foundations of Programming Language Semantics*, (pp. 476–489).

Böhm, A. P. W. and Feo, J. T. (Eds.). (1995). *High Performance Functional Computing, 9–11 April, Denver, Colorado, USA.* Lawrence Livermore National Laboratory.

Bönniger, T., Esser, R., and Krekel, D. (1993). CM-5, KSR1, Paragon XP/S: A comparative description of massively parallel computers on the basis of a catalogue of classifying characteristics. Technical Report KFA-ZAM-IB-9320, Forschungzentrum Jülich GmbH, Germany.

Booth, C. J. M., Bruce, D. I., and Ben-Dyke, A. D. (1996). Improving APOSTLE: a progress report. Technical Report DRA/CIS(SE1)/615/55/Item 4/06/CR9601/1.0, DRA.

Booth, C. J. M., Bruce, D. I., and Ben-Dyke, A. D. (1997). On the implementation of APOSTLE. Technical Report DRA/CIS/CIS5/CR97440/1.0, DERA.

Boothe, B. (1994). Fast accurate simulation of large shared memory multiprocessor. In *Twenty-Seventh Annual Hawaii International Conference on System Sciences.*

Boquist, U. (1995). Interprocedural register allocation for lazy functional languages. In [FPCA '95, 1995].

Bratvold, T. A. (1994). *Skeleton-Based Parallelisation of Functional Programs.* PhD thesis, Heriot-Watt University.

Brewer, E. A., Dellarocas, C. N., and Weihl, W. E. (1991). PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology.

Brown, D. F., Moura, H., and Watt, D. A. (1992). Actress: an action semantics directed compiler generator. In [Kastens and Pfahler, 1992].

Buchberger, B. and Volkert, J. (Eds.). (1994). *CONPAR '94 – VAP VI: Third Joint International Conference on Vector and Parallel Processing, 6–8 September, 1994, Linz, Austria*, number 894 in Lecture Notes in Computer Science. Springer-Verlag.

Burge, W. H. (1975). *Recursive Programming Techniques.* Addison Wesley.

Burkhart, H., Korn, C. F., Gutzwiller, S., Ohnacker, P., and Waser, S. (1993). BACS: Basel algorithm classification scheme. Technical Report 93-3, University of Basel, Switzerland.

Burn, G. L. (1989). Overview of a parallel reduction machine project II. In [Odijk, Rem and Syre, 1989], (pp. 385–396).

Burn, G. L. (1991). *Lazy Functional Languages: Abstract Interpretation and Compilation.* Research Monographs in Parallel and Distributed Computing. Pitman Publishing.

Burton, F. W. (1984). Annotations to control parallelism and reduction order in the distributed evaluation of functional programs. *ACM Transactions on Programming Languages and Systems*, 6(2):159–174.

Burton, F. W. (1987). Functional programming for concurrent and distributed computing. *The Computer Journal*, 3(5):437–450.

Burton, F. W. and Rayward-Smith, V. J. (1994). Worst case scheduling for parallel functional programs. *Journal of Functional Programming*, 4(1):65–75.

Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522.

CC '82 (1982). *ACM SIGPLAN '82 Symposium on Compiler Construction, June, Boston, Massachusetts, USA*, SIGPLAN Notices 17(6).

CC '84 (1984). *ACM SIGPLAN '84 Symposium on Compiler Construction*, SIGPLAN Notices 19(6).

Chakravarty, M. M. T. (1994). A self-scheduling, non-blocking, parallel abstract machine for lazy functional languages. In [Glauert, 1994].

Chen, M. and Halstead, R. (Eds.). (1993). *PPOPP '93: 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 19–22 May, San Diego, California, USA*, SIGPLAN Notices 28(7). ACM Press.

Cheng, D. Y. (1993). A survey of parallel programming languages and tools. Technical Report RND-93-005, NASA Ames Research Center, California.

Chitnis, S. V., Satpathy, M., and Oberoi, S. (1995). Rationalized three instruction machine. *ACM SIGPLAN Notices*, 30(3):94–102.

Clack, C. and Peyton Jones, S. L. (1986). The four-stroke reduction engine. In [LFP '86, 1986], (pp. 220–232).

Clements, A. (1991). *The Principles of Computer Hardware*, volume Second Edition. Oxford University Press.

Cmelik, R. F. and Keppel, D. (1994). Shade: A fast instruction-set simulator for execution profiling. In [SIGMETRICS '94, 1994], (pp. 128–137).

Cole, M. (1989). *Algorithmic Skeletons: Structured Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing. Pitman.

Cook, C. R., Pancake, C. M., and Walpole, R. (1994). Are expectations for parallelism too high? A survey of potential parallel users. In [SC '94, 1994].

Culler, D. E., Goldstein, S. C., Schauser, K. E., and von Eicken, T. (1992). Active messages: A mechanism for integrated communication and computation. Technical Report UCB/CSD 92/675, University of California, Berkeley.

Culler, D. E., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., et al. (1993). LogP: Towards a realistic model of parallel computation. In [Chen and Halstead, 1993], (pp. 1–12).

Cypher, R., Ho, A., Konstantinidou, S., and Messina, P. (1993). Architectural requirements of parallel scientific applications with explicit communications. In [ISCA '93, 1993], (pp. 2–13).

Dahl, O. J. and Nygaard, K. (1966). SIMULA – an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678.

Damas, L. and Milner, R. (1982). Principal type-schemes for functional programs. In *ACM Symposium on Principles of Programming Languages*, (pp. 207–212).

Danelutto, M., Pelagatti, S., et al. (1992). A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems*, 8(1–3):205–220.

Darlington, J., Field, A., Harrison, P., Kelly, P., et al. (1993). Parallel programming using skeleton functions. In [PARLE '93, 1993].

Darlington, J. and Reeve, M. (1981). ALICE: A multiple-processor reduction machine for the parallel evaluation of applicative languages. In [FPCA '81, 1981], (pp. 65–76).

Davis, K. and Hughes, J. (Eds.). (1990). *Functional Programming, Glasgow 1989: Proceedings of the 1989 Glasgow Workshop on Functional Programming, 21–23 August 1989, Fraserburgh, Scotland*, Workshops in Computing. Springer-Verlag.

Davy, J. R. and Dew, P. M. (Eds.). (1995). *Proceedings of the Second Workshop on Abstract Machines Models for Highly Parallel Computers, British Computer Society, Parallel Processing Specialist Group, April, 1993, Leeds, England.* Oxford University Press.

de Bakker, J. W., Nijman, A. L., and Treleaven, P. C. (Eds.). (1987). *PARLE '87: 1st International Conference on Parallel Architectures and Languages Europe, volume II (Parallel Languages)*, number 259 in Lecture Notes in Computer Science. Springer-Verlag.

DEC (1992). *Alpha Architecture Handbook.* Digital Equipment Corporation.

Dennis, J., Aditya, S., Böhm, W., Kirkham, C., and McGraw, J. (1995). Panel: Non-determinancy in functional programming: An essential feature or a programmer's nightmare. In [Böhm and Feo, 1995], (pp. 235–238).

Dennis, J. B. (1995). Static mapping of functional programs: An example in signal processing. In [Böhm and Feo, 1995], (pp. 149–163).

Deschner, J. M. (1990). Simulating multiple processor architectures for compiled graph reduction. In [Davis and Hughes, 1990], (pp. 225–237).

Diller, A. (1994a). Animation using Miranda. In [Diller, 1994b], chapter 19, (pp. 271–278).

Diller, A. (1994b). *Z: An Introduction to Formal Methods* (second ed.). John Wiley and Sons.

Douence, R. and Fradet, P. (1995). Towards a taxonomy of functional language implementations. In *PLILP '95: 7th International Symposium on Programming Languages: Implementations, Logics and Programs, 20–22 September, Utrecht, The Netherlands*, number 982 in Lecture Notes in Computer Science. Springer-Verlag.

Duncan, R. (1990). A survey of parallel computer architectures. *IEEE Computer*, 23(2):5–16.

Ehrig, H. and et al. (Eds.). (1985). *TAPSOFT '85: the International Joint Conference on Theory and Practice of Software Development, volume II (CSE)*, number 186 in Lecture Notes in Computer Science. Springer-Verlag.

EUROPAL '90 (1990). *The First European Conference on the Practical Application of Lisp, Cambridge, U.K., 27–29 March, 1990.*

Fairbairn, J. and Wray, S. (1981). Tim: A simple, lazy abstract machine to execute supercombinators. In [FPCA '81, 1981], (pp. 34–45).

Fasel, J. H. and Keller, R. M. (Eds.). (1987). *Graph reduction, Proceedings of a Workshop, 29 September – 1 October, 1986, Santa Fe, New Mexico, USA*, number 279 in Lecture Notes in Computer Science. Springer-Verlag.

Feeley, M. and Miller, J. S. (1990). A parallel virtual machine for efficient scheme compilation. In [LFP '90, 1990].

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960.

Fowler, M. and Scott, K. (1997). *UML Distilled: Applying the standard object modelling language.* Addison-Wesley.

FPCA '81 (1981). *FPCA '81: 1st Conference on Functional Programming Languages and Computer Architecture, Boston, Massachusetts, October.*

FPCA '93 (1993). *FPCA '93: 6th Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark. ACM Press.

FPCA '95 (1995). *FPCA '95: 7th Conference on Functional Programming Languages and Computer Architecture*, San Diego, California, USA. ACM Press.

Fraser, C. W. and Hanson, D. R. (1991). A retargetable compiler for ANSI C. *ACM SIGPLAN Notices*, 26(10):29–43.

Fraser, C. W. and Hanson, D. R. (1992). Simple register spilling in a retargetable compiler. *Software – Practice and Experience*, 22(1):85–99.

Galton, A. (Ed.). (1987). *Temporal Logics and their Applications.* Academic Press Limited.

Geist, G. A., Heath, M. T., Peyton, B. W., and Worley, P. H. (1990). PICL: A portable instrumented communication library. Technical Report ORNL/TM-11130, Oak Ridge National Laboratory.

Gelernter, D. and Carriero, N. (1992). Coordination languages and their significance. *Communications of the ACM*, 35(2):96–118.

George, L. (1989). An abstract machine for parallel graph reduction. In [Ariola and Arvind, 1989], (pp. 214–229).

Gill, A. (1992). A novel approach towards peephole optimisations. In [Heldal, Holst and Wadler, 1992], (pp. 100–111).

Gill, A. and Marlow, S. (1993). *Happy Manual (version 0.8)* (second ed.)., second edition. URL http://www.dcs.gla.ac.uk/fp/software/happy.html.

Gill, A. and Peyton Jones, S. L. (1994). Building on foldr. In [Hammond and O'Donnell, 1994].

Glasgow FP '94 (1995). *Functional Programming, Glasgow 1994: Proceedings of the 1994 Glasgow Workshop on Functional Programming.* Springer-Verlag.

Glauert, J. G. W. (Ed.). (1994). *6th International Workshop on the Parallel Implementation of Functional Languages, September, Norwich, England.*

Glendinning, I., Hockney, R., Pritchard, D., et al. (1993). Performance visualisation in a portable parallel programming environment. In Kotsis, G. and Haring, G. (Eds.), *Performance Measurement and Visualization of Parallel Processing Systems*, (pp. 251–276). North-Holland.

Goldberg, B. (1988a). Buckwheat: Graph reduction on a shared memory multiprocessor. In [LFP '88, 1988], (pp. 40–51).

Goldberg, B. (1988b). Mulitprocessor execution of functional programs. *International Journal of Parallel Programming*, 17(5):425–473.

Goldberg, B. and Hudak, P. (1987). Alfalfa: Distributed graph reduction on a hypercube multiprocessor. In [Fasel and Keller, 1987], (pp. 94–113).

Goodheart, B. and Cox, J. (1994). *The Magic Garden Explained – the Internals of UNIX System V Release 4.* Prentic Hall.

Goodman, H. (1995). *From Z Specifications to Haskell Programs.* PhD thesis, Department of Computer Science, Birmingham University.

Gropp, W. and Smith, B. (1993). Users manual for the Chameleon parallel programming tools. Technical Report ANL-93/23, Argonne National Laboratory.

Gustafson, J. L. (1988). Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533.

Haines, S., Longshaw, T., and Morison, J. (1997). Event stream analysis. *Journal of Defence Science*, 2(4):480–488.

Hall, C. V. (1994). Optimised ADTS. In [Hammond and O'Donnell, 1994].

Hammond, K. (1991). *Parallel SML: A Functional Language and its Implementation in Dactl.* Research Monographs in Parallel and Distributed Computing. Pitman.

Hammond, K. (1992). The spineless tagless G-machine — not! A draft paper, URL `ftp://ftp.dcs.glasgow.ac.uk:/pub/glasgow-fp/authors/Kevin_Hammond/STG.ps.gz`.

Hammond, K. (1994). Parallel functional programming: An introduction. In [PASCO '94, 1994].

Hammond, K., Burn, G. L., and Howe, D. B. (1994). Spiking your caches. In [Hammond and O'Donnell, 1994].

Hammond, K. and Loidl, H. W. (1996). Making a packet: Cost-effective communication for a parallel graph reducer. In [IFL '96, 1996], (pp. 184–199).

Hammond, K., Loidl, H. W., and Partridge, A. (1995a). Improving granularity in parallel functional programs: A graphical winnowing system for Haskell. In [Glasgow FP '94, 1995].

Hammond, K., Loidl, H. W., and Partridge, A. (1995b). Visualising granularity in parallel programs: A graphical winnowing system for Haskell. In [Böhm and Feo, 1995].

Hammond, K., Mattson Jr., J. S., and Peyton Jones, S. L. (1994). Automatic spark strategies and granularity for a parallel functional language reducer. In [Buchberger and Volkert, 1994].

Hammond, K. and O'Donnell, J. T. (Eds.). (1994). *Functional Programming, Glasgow 1993: Proceedings of the 1993 Glasgow Workshop on Functional Programming, Ayr, Scotland, 3–5 July 1993*, Workshops in Computing. Springer-Verlag.

Hammond, K. and Peyton Jones, S. L. (1992). Profiling scheduling strategies on the GRIP parallel reducer. In [Kuchen and Loogen, 1992].

Hancock, P. (1987). A type-checker. In [Peyton Jones, 1987], chapter 9, (pp. 163–182).

Händler, W. (1982). Innovative computer architecture – how to increase parallelism but not complexity. In D. J. Evans (Ed.), *Parallel Processing Systems – An Advanced Course* (pp. 1–42). Cambridge University Press.

Hankin, C. (1994). *Lambda Calculi: A Guide for Computer Scientists.* Graduate Texts in Computer Science. Oxford University Press.

Hankin, C. L., Osmon, P. E., and Shute, M. J. (1985). COBWEB – a combinator reduction architecture. In [Jouannaud, 1985], (pp. 99–112).

Harper, R., Milner, R., and Tofte, M. (1989). The definition of Standard ML (version 3). Technical report, LFCS, Department of Computer Science, University of Edinburgh.

Harper, R., Milner, R., and Tofte, M. (1990). *The Definition of Standard ML*. MIT Press.

Hartel, P. H. (1994). Benchmarking implementations of lazy functional languages II – two years later. Technical Report CS-94-21, University of Amsterdam.

Hartel, P. H., Grieskamp, W., Hammond, K., Lee, P., Leroy, X., et al. (1996). Benchmarking implementations of functional languages with "pseudoknot", a float-intensive benchmark. *Journal of functional programming*, 6(4):621–655.

Hartel, P. H. and Langendoen, K. G. (1993). Benchmarking implementations of lazy functional languages. In [FPCA '93, 1993], (pp. 341–349).

Heath, M. T. and Finger, J. E. (1993). *ParaGraph: A Tool for Visualizing Performance of Parallel Programs.*

Hehner, E. C. R. (1994). Abstractions of time. In A. W. Roscoe (Ed.), *A Classical Mind. Essay*, Series in Computer Science chapter 12, (pp. 191–210). Prentice Hall International.

Heldal, R., Holst, C. K., and Wadler, P. (Eds.). (1992). *Functional Programming, Glasgow 1991: Proceedings of the 1991 Glasgow Workshop on Functional Programming, Portree, Isle of Skye, 12–14 August 1991*, Workshops in Computing. Springer-Verlag.

Henderson, P. (1986). Functional programming, formal specification and rapid prototyping. *IEEE Transactions on Software Engineering*, 12(2):241–249.

Henglein, F. (1993). Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289.

Hennessy, M. (1990). *The Semantics of Programming Languages – An Elementary Introduction using Structural Operational Semantics.* John Wiley and Sons.

Hewlett Packard (1994). *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* (3rd ed.)., 3rd edition. HP Part Number: 09740-90039.

Hill, J. D. (1993). Vectorizing a non-strict functional language for a data-parallel "spineless (not so) tagless G-machine". In [van Eekelen and Plasmeijer, 1993].

Hill, J. D. (1994). *Data-Parallel Lazy Functional Programming.* PhD thesis, Queen Mary and Westfield College, the University of London.

Hillis, W. D. and Tucker, L. W. (1993). The CM-5 Connection Machine: A scalable supercomputer. *Communications of the ACM*, 36(11):31–40.

Himsolt, M. (1996a). GML: Graph modelling language. Technical report, University of Passau, Germany.

Himsolt, M. (1996b). *Graphlet 4.5 User Manual.* University of Passau, Germany.

Hiromoto, R. E. (1994). On the comparative evaluation of parallel languages and systems: A functional note. In [Hammond and O'Donnell, 1994].

Hockney, R., Berry, M., and et al. (1993). Public international benchmarks for parallel computers. Technical Report TR CS-93-21, University of Tennessee. Revised in 1994.

Holyer, I. (1991). *Functional Programming with Miranda.* Pitman Publishing.

Hooman, J. (1991). A denotational real-time semantics for shared processors. In [Aarts, van Leeuwen and Rem, 1991b], (pp. 184–201).

Howe, D. (1993). The free on-line dictionary of computing. URL http://wombat.doc.ic.-ac.uk/.

Howe, D. B. and Burn, G. L. (1994). Using strictness in the STG machine. In [Hammond and O'Donnell, 1994].

Hudak, P. (1986). Denotational semantics of a para-functional programming language. *International Journal of Parallel Programming*, 15(2):103–125.

Hudak, P. (1987). Arrays, non-determinism, side-effects, and parallelism: A functional perspective (extended abstract). In [Fasel and Keller, 1987], (pp. 312–327).

Hudak, P. (1988). Exploring parafunctional programming: Separating the what from the how. *IEEE Software*, 5(1):54–61.

Hudak, P. (1989). Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411.

Hudak, P. (1991). Para-functional programming in Haskell. In [Szymański, 1991], chapter 5.

Hudak, P. and Anderson, S. (1987). Pomset interpretations of parallel functional programs. In [Kahn, 1987], (pp. 234–256).

Hudak, P. and Anderson, S. (1988). Haskell solutions to the language session problems at the 1988 salishan high-speed computing conference. Technical Report YALEU/DCS/RR-627, Yale University.

Hudak, P. and Jones, M. P. (1994). Haskell vs. Ada vs. C++ vs. Awk vs. ... an experiment in software prototyping productivity. Technical report, Yale University.

Hudak, P. and Mohr, E. (1988). Graphinators and the duality of SIMD and MIMD. In [LFP '88, 1988], (pp. 224–234).

Hudak, P., Peyton Jones, S. L., Wadler, P., et al. (1992). Report on the programming language Haskell. *ACM SIGPLAN Notices*, 27(5).

Hughes, J. (1989). Why functional programming matters. *The Computer Journal*, 32(2):98–107.

Hughes, J. (Ed.). (1991). *FPCA '91: 5th Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science.* Springer-Verlag.

Hughes, J. M. (1984). *The Design and Implementation of Programming Languages.* PhD thesis, Oxford University.

Hwang, K. and Briggs, F. A. (1985). *Computer Architecture and Parallel Processing.* Computer Science Series. McGraw-Hill International.

Hwang, S. and Rushall, D. (1992). The ν-STG machine: a parallelized spineless tagless graph reduction machine in a distributed memory architecture. In [Kuchen and Loogen, 1992].

IFL '96 (1996). *8th International Workshop on the Implementation of Functional Languages, September, Bonn/Bad-Godesberg, Germany*, number 1268 in Lecture Notes in Computer Science. Springer-Verlag.

ISCA '89 (1989). *ISCA '89: 16th Annual International Symposium on Computer Architecture.* IEEE Computer Science Press.

ISCA '90 (1990). *ISCA '90: 17th Annual International Symposium on Computer Architecture, May, Seattle, Washington, USA.* ACM Press.

ISCA '92 (1992). *ISCA '92: 19th Annual International Symposium on Computer Architecture, May, Gold Coast, Australia.* ACM Press.

ISCA '93 (1993). *ISCA '93: 20th Annual International Symposium on Computer Architecture, 16–19 May, San Diego, California, USA*, ACM Computer Architecture News 21(2). ACM Press.

Islam, N. and Campbell, R. H. (1992). Design considerations for shared memory multiprocessing message systems. *IEEE Transactions on Parallel and Distibuted Systems*, 3(6):702–711.

IWMS '92 (1992). *International Workshop on Memory Management, September, St. Malo, France*, number 637 in Lecture Notes in Computer Science. Springer-Verlag.

Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modelling.* John Wiley and Sons, Inc.

Jay, C. B. and Cockett, J. R. B. (1994). Shapely types and shape polymorphism. In [Sannella, 1994], (pp. 302–316).

Johnson, E. E. (1988). Completing an MIMD multiprocessor taxonomy. *ACM Computer Architecture News*, 16(3):44–47.

Johnsson, T. (1991). Parallel evaluation of functional programs: The $\langle v, G\rangle$-machine approach (summary). In [Aarts, van Leeuwen and Rem, 1991a], (pp. 1–5).

Jones, M. P. (1994). *Qualified Types: Theory and Practice.* PhD thesis, University of Nottingham.

Jones, M. P. and Hudak, P. (1993). Implicit and explicit parallel programming in Haskell. Technical Report YALE/DCS/RR-982, Yale University.

Jones, N. D. (Ed.). (1980). *Semantics-Directed Compiler Generation: Proceedings of a workshop*, number 94 in Lecture Notes in Computer Science, Aarhus, Denmark. Springer-Verlag.

Jouannaud, J.-P. (Ed.). (1985). *FPCA '85: 2nd Conference on Functional Programming Languages and Computer Architecture, 16–19 September, Nancy, France*, number 201 in Lecture Notes in Computer Science. Springer-Verlag.

Jouvelot, P. (1986). Designing new languages or new language manipulation systems using ML. *ACM SIGPLAN Notices*, 21(8):40–52.

Joy, M. and Axford, T. (1992). A parallel graph reduction machine. In [Kuchen and Loogen, 1992].

Kahn, G. (Ed.). (1987). *FPCA '87: 3rd Conference on Functional Programming Languages and Computer Architecture, 14–16 September, Portland, Oregon, USA*, number 274 in Lecture Notes in Computer Science. Springer-Verlag.

Kane, G. and Heinrich, J. (1992). *MIPS RISC Architecture.* Prentice Hall.

Kaser, O., Pawagi, S., Ramakrishnan, C. R., Ramakrishnan, I. V., and Sekar, R. C. (1992). Fast parallel implementation of lazy languages – the EQUALS experience. In [White, 1992], (pp. 335–344).

Kastens, U. and Pfahler, P. (Eds.). (1992). *CC '92: 4th International Conference on Compiler Construction, 5–7 October, Paderborn, Germany*, number 641 in Lecture Notes in Computer Science. Springer-Verlag.

Keller, R., Lindstrom, G., and Patil, S. (1979). A loosely-coupled applicative multiprocessing system. In *AFIPS*, (pp. 613–622).

Kelly, P. (1989). *Functional Programming for Loosely-Coupled Multiprocessors.* Research Monographs in Parallel and Distributed Computing. Pitman.

Kernighan, B. W. and Ritchie, D. M. (1978). *The C Programming Language.* Prentice Hall Inc.

Kessler, R. R. (Ed.). (1994). *LFP '94: ACM Conference on LISP and Functional Programming, 27–29 June, Orlando, Florida, USA*, LISP Pointers VII(3). ACM Press.

Kewley, J. M. and Glynn, K. (1990). Evaluation annotations for Hope+. In [Davis and Hughes, 1990], (pp. 329–337).

Kfoury, A. J., Tiuryn, J., and Urzyczyn, P. (1993). Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311.

Kingdon, H., Lester, D. R., and Burn, G. L. (1991). The HDG-machine: a highly distributed graph-reducer for a transputer network. *The Computer Journal*, 34(4).

Koeninger, R. K., Furtney, M., and Walker, M. (1992). A shared memory MPP from Cray Research. *Digital Technical Journal*, 6(2).

Kuchen, H. and Gladitz, K. (1992). Implementing bags on a shared memory MIMD-Machine. In [Kuchen and Loogen, 1992].

Kuchen, H. and Loogen, R. (Eds.). (1992). *4th International Workshop on the Parallel Implementation of Functional Languages, 28–30 September, Aachen, Germany.*

Lander, R. E. and Fisher, M. J. (1980). Parallel prefix computation. *Journal of the ACM*, 27(4):831–838.

Landin, P. J. (1966). The next 700 programming languages. *Communications of the ACM*, 9(3):157–166.

Langendoen, K. (1993). *Graph Reduction on Shared Memory Multiprocessors.* PhD thesis, Universiteit van Amsterdam.

Launchbury, J. and Sansom, P. (Eds.). (1993). *Functional Programming, Glasgow 1992: Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, 6–8 July*, Workshops in Computing. Springer-Verlag.

Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. (1989). *The Design and Implementation of the 4.3BSD UNIX Operating Sysetm.* Addison-Wesley.

Lenoski, D., Laudon, J., Joe, T., Gupta, A., Hennessy, J., et al. (1992). The DASH prototype: Implementation and performance. In [ISCA '92, 1992], (pp. 92–103).

Lester, D. R. (1989). An efficient distributed garbage collection algorithm. In [Odijk, Rem and Syre, 1989], (pp. 207–223).

LFP '86 (1986). *LFP '86: ACM Conference on LISP and Functional Programming.* ACM Press.

LFP '88 (1988). *LFP '88: ACM Conference on LISP and Functional Programming, June, Snowbird, Utah, USA.* ACM Press.

LFP '90 (1990). *LFP '90: ACM Conference on LISP and Functional Programming, June, Nice, France.* ACM Press.

Loogen, R., Kuchen, H., Indermark, K., and Damm, W. (1991). Distributed implementation of programmed graph reduction. In [Aarts, van Leeuwen and Rem, 1991a], (pp. 136–157).

Maaßen, A. (1992). Parallel programming with data structures and higher order functions. *Science of Computer Programming*, 18:1–38.

MacDonald, N. B. (1992). An overview of SIMD parallel systems: AMT DAP, Thinking Machines CM-200, and MasPar MP-1. Technical Report EPCC-TR92-18, University of Edinburgh.

MacLennan, B. J. (1987). *Principles of Programming Languages: Design, Evaluation, and Implementation* (second ed.). Holt, Rinehart and Winston.

Maheshwari, P. (1990). *Controlling Parallelism in Functional Programs Using Complexity Information.* PhD thesis, University of Manchester, Department of Computer Science.

Maranget, L. (1991). GAML: a parallel implementation of lazy ML. In [Hughes, 1991].

Markoff, J. (1994). In supercomputers, bigger and faster means trouble. *The New York Times.*

Mattson Jr., J. S. (1993a). *An Effective Speculative Evaluation Technique for Parallel Supercombinator Graph Reduction.* PhD thesis, University of California, San Diego.

Mattson Jr., J. S. (1993b). Performance of parallel schedulers for distributed graph reduction. In [van Eekelen and Plasmeijer, 1993].

May, C., Silha, E., Simpson, R., and Warren, H. (Eds.). (1994). *The PowerPC Architecture: A Specification for a New Faimly of RISC Processors* (2nd ed.). Morgan Kaufmann Publishers, Inc.

McColl, W. F. (1995). Bulk synchronous parallel computing. In [Davy and Dew, 1995], (pp. 42–58).

Michaelson, G. J. (1993). *Interpreter prototypes from formal language definitions.* PhD thesis, Heriot-Watt University.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.

Milner, R. (1993). Elements of interaction (Turing award lecture). *Communications of the ACM*, 36(1):78–89.

Mirani, R. and Hudak, P. (1995). First-class schedules and virtual maps. In [FPCA '95, 1995].

Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc.

Nikhil, R. S., Arvind, Augustsson, L., Maessen, J., Zhou, Y., et al. (1995). pH language reference manual, version 1.0 — preliminary. Technical Report Computation Structures Group Memo 369, Massachusetts Institute of Technology.

Nöcker, E. G. J. M. H., Smetsers, J. E. W., Plasmeijer, R., and van Eekelen, M. (1991). Concurrent Clean. In [Aarts, van Leeuwen and Rem, 1991b], (pp. 202–219).

Norman, M. G. and Thanisch, P. (1993). Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, 25(3). Also available as a University of Edinburgh technical report, number TR-91-14.

North, S. C. and Reppy, J. H. (1987). Concurrent garbage collection on stock hardware. In [Kahn, 1987], (pp. 113–133).

Odijk, E. A. M., Rem, M., and Syre, J.-C. (Eds.). (1989). *PARLE '89: 2nd International Conference on Parallel Architectures and Languages Europe, 12–16 June, Eidhoven, Netherlands, volume I (Parallel Architectures)*, number 365 in Lecture Notes in Computer Science. Springer-Verlag.

Oldehoeft, R. R. and Cann, D. C. (1988). Applicative parallelism on a shared-memory multiprocessor. *IEEE Software*, 5(1):62–70.

Oren, J. and Ramanathan, G. (1993). Survey of commercial parallel machines. *ACM Computer Architecture News*, 21(3):13–33.

Ostheimer, G. (1993). *Parallel Functional Programming for Message-Passing Multiprocessors.* PhD thesis, University of St. Andrews, Fife, Scotland. Published as technical report CS-93-8.

PADS '96 (1996). *PADS '96: Proceedings of the 10th Workshop on Parallel and Distributed Simulation, 22–24 May, Philadelphia, Pennsylvania, USA.*

PARLE '93 (1993). *PARLE '93: 5th International Conference on Parallel Architectures and Languages Europe, June*, Lecture Notes in Computer Science. Springer-Verlag.

Parrott, D. J. (1993). *Synthesising Parallel Functional Programs to Improve Dynamic Scheduling.* PhD thesis, University College London.

Partain, W. (1993). The `nofib` benchmarking suite. In [Launchbury and Sansom, 1993].

Partain, W. D. (1991). Normal-order reduction using scan primitives. In [Peyton Jones, Hutton and Holst, 1991], (pp. 218–226).

PASCO '94 (1994). *PASCO '94: First International Symposium on Parallel Symbolic Computation, September.* World Scientific Publishing Company.

Pelagatti, S. (1993). *A methodology for the development and the suspport of massively parallel programs.* PhD thesis, Universitá di Pisa-Genova-Udine.

Peyton Jones, S. L. (1987). *The Implementation of Functional Programming Languages.* Prentice Hall International.

Peyton Jones, S. L. (1989). Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175–186.

Peyton Jones, S. L. (1992). Implementing lazy functional languages on stock hardware: the spineless tagless G-Machine. *Journal of Functional Programming*, 2(2):127–202.

Peyton Jones, S. L., Clack, C., Salkild, J., and Hardie, M. (1987). GRIP – a high-performance architecture for parallel graph reduction. In [Kahn, 1987], (pp. 98–111).

Peyton Jones, S. L., Gordon, A., and Finne, S. (1996). Concurrent Haskell. In [Steele Jr, 1996].

Peyton Jones, S. L., Hall, C., Hammond, K., Partain, W., and Wadler, P. (1993). The Glasgow Haskell compiler: A technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT), Keele.*

Peyton Jones, S. L., Hutton, G., and Holst, C. K. (Eds.). (1991). *Functional Programming, Glasgow 1990: Proceedings of the 1990 Glasgow Workshop on Functional Programming, 13–15 August 1990, Ullapool, Scotland*, Workshops in Computing. Springer-Verlag.

Peyton Jones, S. L. and Launchbury, J. (1991). Unboxed values as first class citizens in a non-strict functional language. In [Hughes, 1991].

Peyton Jones, S. L., Launchbury, J., and Partain, W. (1994). GHC prelude: Types and operations. Part of the Glasgow Haskell distribution, available via ftp from `ftp.dcs.glasgow.ac.uk`.

Peyton Jones, S. L. and Lester, D. (1991). *Implementing Functional Languages – A Tutorial.* Computer Science. Prentice Hall International.

Peyton Jones, S. L. and Partain, W. (1994). On the effectiveness of a simple strictness analyser. In [Hammond and O'Donnell, 1994], (pp. XII–1–18).

Peyton Jones, S. L. and Salkild, J. (1989). The spineless tagless G-Machine. In [Ariola and Arvind, 1989], (pp. 184–201).

Peyton Jones, S. L. and Wadler, P. (1987). Structured types and the semantics of pattern-matching. In [Peyton Jones, 1987], chapter 4, (pp. 51–77).

Peyton Jones, S. L. and Wadler, P. (1992). A static semantics for Haskell. A draft paper, URL ftp://ftp.dcs.glasgow.ac.uk/pub/glasgow-fp/authors/-Simon_Peyton_Jones/static-semantics.dvi.gz.

Peyton Jones, S. L. and Wadler, P. (1993). Imperative functional programming. In [POPL '93, 1993], (pp. 71–84).

Pitman, K. M. (1990). Exceptional situations in Lisp. In [EUROPAL '90, 1990], (pp. 109–115).

Plasmeijer, R. and van Eekelen, M. (1993a). *Functional Programming and Parallel Graph Rewriting*. International Computer Science Series. Addison-Wesley.

Plasmeijer, R. and van Eekelen, M. C. J. D. (1993b). Process annotations and process types. In [Sleep, Plasmeijer and van Eekelen, 1993], chapter 25, (pp. 347–362).

POPL '90 (1990). *POPL '90: 17th ACM Symposium on Principles of Programming Languages, 17–19 January, San Fransisco, California, USA*. ACM press.

POPL '92 (1992). *POPL '92: 19th ACM Symposium on Principles of Programming Languages, 19–22 January, Albuquerque, New Mexico, USA*. ACM press.

POPL '93 (1993). *POPL '93: 20th ACM Symposium on Principles of Programming Languages, 11–13 January, Charleston, South Carolina, USA*. ACM press.

Raskovsky, M. and Collier, P. (1980). From standard to implementation denotational semantics. In [Jones, 1980], (pp. 94–139).

Reynolds, J. C. (1985). Three approaches to type structure. In [Ehrig and et al., 1985].

RISC Machines Ltd (ARM) (1994). *ARM710 Data Sheet*. Document Number: ARM DDI 0024E.

Robinson, J. A. (1965). A machine-orientated logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41.

Roscoe, A. W. (1997). *The Theory and Practice of Concurrency*. Prentice Hall.

Sabot, G. W. (1988). *The Paralation Model: Architecture Independent Parallel Programming*. Cambridge, Massachusetts: The MIT Press.

Sadri, F. (1987). Three recent approaches to temporal reasoning. In [Galton, 1987], chapter 4, (pp. 121–168).

Sahni, S., Prasanna, V. K., and Bhatkar, V. P. (Eds.). (1995). *Proceedings of the International Conference on High Performance Computing, December 27–30, 1995, New Delhi, India*. Tata McGraw-Hill Publishing Company Limited.

Sands, D. (1990). *Calculi for Time Analysis of Functional Programs*. PhD thesis, Department of Computing, Imperial College, University of London.

Sannella, D. (Ed.). (1994). *Programming Languages and Systems – ESOP '94: 5th European Symposium on Programming, Edinburgh, U.K., April 1994, Proceedings*, Lecture Notes in Computer Science. Springer-Verlag.

Sansom, P. M. (1992). Combining single-space and two-space compacting garbage collectors. In [Heldal, Holst and Wadler, 1992], (pp. 312–323).

Sansom, P. M. and Peyton Jones, S. L. (1993). Generational garbage collection for Haskell. In [FPCA '93, 1993].

Santos, A. (1995). *Compilation by Transformation in Non-Strict Functional Languages.* PhD thesis, University of Glasgow.

SC '94 (1994). *Supercomputing '94, 13–18 November, Washington, D.C., Maryland, USA.*

Schlesinger, J. D. and Kuehn, J. T. (1993). adl: An architecture description language version 1.0. Technical Report SRC-TR-93-104, Supercomputing Research Center, Maryland.

Schmidt, D. A. (1986). *Denotational Semantics, a Methodology for Language Development.* Allyn and Bacon.

Schmidt, D. A. (1994). *The Structure of Typed Programming Languages.* Foundations of Computing. The MIT Press.

Schreiner, W. (1994). *Parallel Functional Programming for Computer Algebra.* PhD thesis, RISC-Linz, Johannes Kepler University, Linz, Austria.

Sestoft, P. (1994). Deriving a lazy abstract machine. Technical Report ID-TR 1994-146, Department of Computer Science, Technical University of Denmark.

Sethi, R. (1982). Control flow aspects of semantics directed compiling (summary). In [CC '82, 1982], (pp. 245–260).

Seward, J. R. (1992). Towards a strictness analyser for Haskell: Putting theory into practise. Master's thesis, University of Manchester.

Shao, Z. and Appel, A. W. (1994). Space-efficient closure representations. In [Kessler, 1994].

Shao, Z. and Appel, A. W. (1995). A type-based compiler for Standard ML. In [Wall and Hanson, 1995], (pp. 116–129).

SIGMETRICS '94 (1994). *1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, 16–20 May, Nashville, Tennessee, USA*, Performance Evaluation Review 22(1).

SIGMETRICS '95 (1995). *1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, 15–19 May, Ottawa, Ontario, Canada*, Performance Evaluation Review 23(1).

SIGPLAN '96 (1996). *SIGPLAN '96: 9th Conference on Programming Language, Design and Implementation, 21–24 May, Philadelphia, USA.* ACM Press.

Singh, J. P., Weber, W.-D., and Gupta, A. (1992). SPLASH: Stanford parallel applications for shared-memory. *ACM Computer Architecture News*, 20(1):5–44.

Sipelstein, J. M. and Blelloch, G. E. (1991). Collection-orientated languages. *Proceedings of the IEEE*, 79(4):504–523.

Sites, R. L. (1992). Alpha AXP architecture. *Digital Technical Journal*, 4(4).

Skedzielewski, S. K. (1991). Sisal. In [Szymański, 1991], chapter 4.

Skillicorn, D. B. (1988). A taxonomy for computer architectures. *IEEE Computer*, 21(11):46–57.

Skillicorn, D. B. (1990). Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50.

Sleep, R., Plasmeijer, R., and van Eekelen, M. (Eds.). (1993). *Term Graph Rewriting: Theory and Practise.* John Wiley & Sons Ltd.

Smith, B. (1990). The end of architecture (keynote address). In [ISCA '90, 1990].

Snir, M., Gropp, W., Lusk, E., Geist, A., Hempel, R., Clarke, L., et al. (1994). MPI: A message-passing interface standard. Technical report, Message Passing Interface Forum.

Snyder, L. (1986). Type architecture, shared memory and the corollary of modest potential. *Annual Review of Computer Science*, 1:289–317.

Springsteel, F. and Stojmenovic, I. (1989). Parallel general prefix computations with geometric, algerbraic, and other applications. *International Journal of Parallel Programming*, 18(6):485–503.

Steele Jr, G. L. (Ed.). (1996). *POPL '96: 23rd ACM Symposium on Principles of Programming Languages, January, St Petersburg Beach, Florida, USA*, volume 23. ACM Press.

Stoy, J. E. (1977). *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* Computer Science. The MIT Press.

Stroustrup, B. (1991). *The C++ Programming Language.* Addison-Wesley.

Sun Microsystems (1988). *SPARC Architecture Manual.* Mountain View, California, USA.

Sun Microsystems (1998). *The Java Development Kit Version 1.2.* available via URL `http://java.sun.com/products/jdk`.

Szymański, B. K. (Ed.). (1991). *Parallel Functional Languages and Compilers.* ACM Press.

Tennent, R. D. (1976). The denotational semantics of programming languages. *Communications of the ACM*, 19(8):437–453.

The FAST project team (1993). FAST: Functional programming for arrays of transputers – the collected papers. Technical Report DOC 93/4 (Imperial College), CSTR 93-15 (University of Southampton), Department of Computing, Imperial College of Science, Technology and Medicine, University of London, and the Department of Electronics and Computer Science, University of Southampton, with contributions from the Department of Computer Systems, University of Amsterdam.

Tofte, M. (1990). *Compiler Generators: what they can do, what they might do, and what they will probably never do*, volume 19 of *EATCS Monograph on Theoretical Computer Science.* Springer-Verlag.

Traub, K. R. (1991). *Implementation of Non-Strict Functional Programming Languages.* Research Monographs in Parallel and Distributed Computing. Pitman Publishing.

Trinder, P. W., Hammond, K., Partridge, A. S., Peyton Jones, S. L., et al. (1996). GUM: a portable parallel implementation of Haskell. In [SIGPLAN '96, 1996].

Turner, D. A. (1979). A new implementation technique for applicative languages. *Software – Practise and Experience*, 9:31–49.

Valiant, L. (1990). A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111.

van Eekelen, M. and Plasmeijer, R. (Eds.). (1993). *5th International Workshop on the Parallel Implementation of Functional Languages, September, Nijmegen, The Netherlands.*

Vree, W. (1989). *Design Considerations For A Parallel Reduction Machine.* PhD thesis, Universiteit van Amsterdam.

Wadler, P. (1992). The essence of functional programming. In [POPL '92, 1992].

Wadsworth, C. P. (1971). *Semantics and pragmatics of the lambda calculus.* PhD thesis, Oxford University.

Wall, D. W. and Hanson, D. R. (Eds.). (1995). *SIGPLAN '95: 8th Conference on Programming Language, Design and Implementation, 18–21 June, La Jolla, California, USA*, SIGPLAN Notices 30(6). ACM Press.

Wand, M. (1984). A semantic prototyping system. In [CC '84, 1984], (pp. 213–221).

Watson, D. (1989). *High-Level Languages and Their Compilers.* Addison Wesley.

Watson, P. and Watson, I. (1987). Evaluating functional programs on the FLAGSHIP machine. In [Kahn, 1987], (pp. 80–97).

Watt, D. A. (1991). *Programming Language Syntax and Semantics.* Computer Science. Prentice Hall International.

Weber, W.-D. and Gupta, A. (1989). Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In [ISCA '89, 1989], (pp. 273–280).

White, J. L. (Ed.). (1992). *LFP '92: ACM Conference on LISP and Functional Programming, 22–24 June, San Francisco, USA*, LISP Pointers V(1). ACM Press.

Wilson, P. R. (1992). Uniprocessor garbage collection techniques. In [IWMS '92, 1992], (pp. 1–42).

Wonnacott, P. and Bruce, D. (1996). The APOSTLE simulation language: Granularity control and performance data. In [PADS '96, 1996], (pp. 114–123).

Worley, P. H. (1992). A new PICL trace file format. Technical Report ORNL/TM-12125, Oak Ridge National Laboratory.