Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine *
Version 2.5

Simon L Peyton Jones
Department of Computing Science, University of Glasgow G12 8QQ
simonpj@dcs.glasgow.ac.uk

March 17, 1993

Abstract

The Spineless Tagless G-machine is an abstract machine designed to support non-strict higher-order functional languages. This presentation of the machine falls into three parts. Firstly, we give a general discussion of the design issues involved in implementing non-strict functional languages.

Next, we present the *STG language*, an austere but recognisably-functional language, which as well as a *denotational* meaning has a well-defined *operational* semantics. The STG language is the "abstract machine code" for the Spineless Tagless G-machine.

Lastly, we discuss the mapping of the STG language onto stock hardware. The success of an abstract machine model depends largely on how efficient this mapping can be made, though this topic is often relegated to a short section. Instead, we give a detailed discussion of the design issues and the choices we have made. Our principal target is the C language, treating the C compiler as a portable assembler.

Version 2.5 of this paper (minus appendix) appears in the Journal of Functional Programming 2(2) (April 1992), pp127-202. The indexing in the published version is one page out throughout, due to a CUP typesetting bug.

Changes in Version 2.0: large new section on comparing the STG machine with other designs (Section 3); profiling material (Section 11); index.

Changes in Version 2.1: proper statement of the initial state of the machine (Section 5.1); reformulation of CAF updates (Section 10.8); new format for state transition rules, separating the guards which govern the applicability of the rules ("such that") from the auxiliary definitions ("where") — Section 5.

Changes in Version 2.2: introduction of the term "lambda-form"; new subsection on lambda lifting (Section 4.5); discussion of copy-vs-share in Section 10.6; allow a variable-binding form of default alternative in algebraic `case` expressions.

Changes in Version 2.3: more explicit discussion of the translation into the STG language (Section 4.1); some re-ordering of sub-sections in Section 4; an overview of the code generation process at the start of Section 9.

Changes in Version 2.4: new-format profiling in Section 11; new section on black holes, saying how to avoid space leaks without black-holing (Section 9.3.3).

*Previously entitled "The Spineless Tagless G-machine: a second attempt".

Changes in Version 2.5: appendix added giving gory details. Apart from the appendix, this is essentially the version published in the Journal of Functional Programming. Very minor changes to main paper: fixed bug in Fig 5, and other typos.

Contents

1 Introduction

The challenges of compiling non-strict functional languages have given rise to a whole stream of research work. Generally the discussion of this work has been focussed around the design of a so-called "abstract machine", which distils the key aspects of the compilation technique without becoming swamped in the details of source language or code generation. Quite a few such abstract-machine designs have been presented in recent years; examples include the G-machine (Augustsson [1987]; Johnsson [1987]), TIM (Fairbairn & Wray [1987]), the Spineless G-machine (Burn, Peyton Jones & Robson [1988]), the Oregon G-machine chip (Kieburtz [1987]), the CASE machine (Davie & McNally [1989]), the HDG machine (Kingdon, Lester & Burn [1991]), the $\langle \nu, G \rangle$ machine (Augustsson & Johnsson [1989]), and the ABC machine (Koopman [1990]).

Early implementations, especially those based on graph reduction, were radically different from conventional compiler technology: the difference between an SK combinator implementation (Turner [1979]) and (say) a Lisp compiler is substantial. So great was this divergence that new hardware architectures were developed specifically to support the execution model (Scheevel [1986]; Stoye, Clarke & Norman [1984]). As understanding has developed, though, it has been possible to recognise features of more conventional systems emerging from the mist, and to generate efficient code for stock architectures.

In this paper we present a new abstract machine for non-strict functional languages, the Spineless Tagless G-machine, set it in the context of conventional compiler technology, and give a detailed discussion of its mapping onto stock hardware. Our design exhibits a number of unusual features:

- In all other the abstract machines mentioned above, the abstract machine code for a program consists of a sequence of abstract machine instructions. Each instruction is given a precise operational semantics using a state transition system.

  We take a different approach: the abstract machine language is itself a very small functional language, which has the usual denotational semantics. In addition, though, each language construct has a direct operational interpretation, and we give an operational semantics for the same language using a state transition system.

- Objects in the heap, both unevaluated suspensions and head normal forms, have a uniform representation with a code pointer in their first field. Many implementations examine tag fields on heap objects to decide how to treat them. With our representation we never do this; instead, a jump is made to the code pointed to by the object. This is why we call the machine "tagless".

- A pervasive feature of functional programs is the construction of data structures, and their traversal using pattern-matching. Many abstract machine designs say very little about how to do this efficiently, but we pay a lot of attention to it.

- The machine manipulates *unboxed values* directly, based on the ideas in a companion paper (Peyton Jones & Launchbury [1991]). This is essential for an efficient implementation of arithmetic, but it is usually hidden in the code generator.

- There is scope for exploiting the fruits of both strictness analysis and sharing analysis to improve execution speed.

- Lambda lifting, a common feature of almost all functional-language implementations, is not carried out. Instead, the free variables of lambda abstractions are identified, but the abstraction is left in place.

- The machine is particularly well-suited for parallel implementations, although space prevents this aspect being discussed in this paper (Peyton Jones, Clack & Salkild [1989]).

Almost all the individual ideas we describe are present in other implementations, but their combination produces a particularly fast implementation of lazy functional languages.

An earlier version of this paper (Peyton Jones & Salkild [1989]), had a similar title and introduction to this one. The underlying machine being described is mostly unchanged, but the presentation has been completely rewritten.

2 Overview

The paper divides into three parts. Part I explores the design space to show how the STG machine fits into a wider context. Part II introduces the abstract machine and gives its operational semantics, and Part III discusses how the abstract machine is mapped onto stock hardware.

2.1 Part I: the design space

The implementation of non-strict functional languages has tended to be done in a separate world to that of "real compilers". One goal of this paper is to help bridge the gap between these two cultures. To this end we identify several key aspects of a compiler (its representation of data structures, its treatment of function application, and its compilation of case-analysis on data structures), and compare the approach we take with that of others.

We hope that this exercise may be useful in its own right, as well as setting the context for the rest of the paper.

2.2 Part II: the abstract machine

The usual way of presenting an evaluation model for a functional language is to define an *abstract machine*, which executes an instruction stream. The abstract machine is given an operational semantics using a *state transition system*, and *compilation rules* are given for converting a functional program into abstract machine code. The application of these compilation rules is usually preceded by lambda lifting (Johnsson [1985]), which eliminates lambda abstractions in favour of supercombinators, functions with no free variables. A good example of this approach is the G-machine (Johnsson [1987]; Peyton Jones [1987]), whose abstract machine code is called G-code.

This approach suffers an annoying disadvantage: *the abstract machine is generally not abstract enough.* For example, the abstract G-machine uses the stack to hold many intermediate values. When G-code is to be compiled into native machine code, many stack operations can be eliminated by holding the intermediate values in registers. The code generator has to simulate the operation of the abstract stack, which is used, in effect, mainly to name intermediate values. Not only does this process complicate the code generator, but it makes G-code harder to manipulate and optimise.

To avoid this problem, one is driven to introduce explicitly-named values in the abstract machine, which is how the T-code of our earlier paper was derived (Peyton Jones & Salkild [1989]). Unfortunately, the simplicity of the abstract machine is now lost.

We take a slightly different approach here. Instead of defining a new abstract machine, we use a very small functional language, the *STG language*, as the abstract machine code. It has the usual denotational semantics, so it is in principle possible to check the transformation of the original program into the STG language is correct. But we also give it a direct operational semantics using a state transition system, which explains how we intend it to be executed. The problem of proving the entire system correct is thereby made easier than, for example, the G-machine[1], because only the equivalence of the denotational and operational semantics of a *single* language is involved. Even so, it is a substantial task, and we do not attempt it here.

2.3 Part III: mapping the abstract machine onto real hardware

Typically, much is written about the compilation of a functional program into abstract machine code, and rather little about how to map the abstract machine onto the underlying hardware. Yet the abstract machine can only be considered a success if this mapping works well; that is, the resulting code is efficient.

We believe that the Spineless Tagless G-machine comes out well in this regard, and devote considerable space to discussing the mapping process. One of the nice aspects is that a variety of mappings are possible, of increasing complexity and efficiency.

Our target machine code is the C language. This has become common of late, conferring, as it does, wide portability. We may pay some performance penalty for not generating native machine code, and plan to build other code generators which do so.

2.4 Source language and compilation route

We are interested in compiling *strongly-typed, higher-order, non-strict, purely functional languages* such as LML, or Haskell. We expect heavy use of both higher-order functions and the non-strict semantics (Hughes [1989]).

This paper is only about the back end of a compiler. Our complete compilation route involves the following steps:

[1] The task of proving a simple G-machine correct is carried out by Lester in his thesis (Lester [1989]).

1. The primary source language is Haskell (Hudak et al. [1992]), a strongly-typed, non-strict, purely-functional language. Haskell's main innovative feature is its support for systematic overloading.

2. Haskell is compiled to a small *Core language*. All Haskell's syntactic sugar is translated out, type checking is performed, and overloading is resolved. Pattern-matching is translated into simple `case` expressions, each of which performs only a single level of matching.

3. Program analyses and a variety of transformations are applied to the Core language.

4. The Core language is translated to the *STG language*, which we introduce in Section 4. This transformation is rather simple.

5. The code generator translates the STG language into *Abstract C*. The latter is just an internal data type which can simply be printed out as C code, but which can also serve as an input to a native-code generator.

Strictness analysis plays an important role in compilers for non-strict languages, enabling the compiler to determine cases where function arguments can be passed in evaluated form, which is often more efficient. Using this technology compilers for lazy languages can generate code which is sometimes as fast as or faster than C (Smetsers et al. [1991]).

Usually the results of strictness analysis are passed to the code generator, which is thereby made significantly more complicated. We take a different approach. We extend the Core and STG languages with full-fledged *unboxed values*, which makes them expressive enough to incorporate the results of strictness analysis by simple program transformations. We give a brief introduction to unboxed values in Section 4.7, but the full details, including the transformations required to exploit strictness analysis, are given in a separate paper (Peyton Jones & Launchbury [1991]).

The code generator, which is the subject of this paper, is therefore not directly involved in strictness analysis or its exploitation, so we do not discuss it further.

Part I: Exploring the design space

3 Exploring the design space

Before introducing the STG machine in detail we pause to explore the design space a little. The STG machine has its roots in lazy graph reduction. It is now folk-lore that, while graph reduction looks very different to conventional compiler technology, the best compilers based on graph reduction generate quite similar code to those for (say) Lisp. In this section we attempt to compare some aspects of the STG machine with more conventional compilers.

Any implementation is the result of a raft of inter-related design decisions, each of which is partly justified by the presence of the others. That makes it hard to find a place to start our description. We proceed by asking three key questions, which help to locate the implementation techniques for any non-strict higher-order language:

- How are function values, data values and unevaluated expressions represented (Section 3.1)?
- How is function application performed (Section 3.2)?
- How is case analysis performed on data structures (Section 3.3)?

This section gives the context and motivation for many of the implementation techniques described in Parts II and III, and suitable forward references are given.

3.1 The representation of closures

The heap contains two kinds of objects: *head normal forms* (or *values*), and as-yet unevaluated suspensions (or *thunks*). Head normal forms can be further classified into two kinds: *function values* and *data values*. A value may contain thunks inside it; for example, a list `Cons` cell might have an unevaluated head and/or tail. A value which contains no thunks inside it is called a *normal form*.

It is worth noting that, in a polymorphic language, it is not always possible to distinguish thunks whose value will turn out to be a function from thunks whose value is a data value. For example, consider the composition function:

```
compose f g x = f (g x)
```

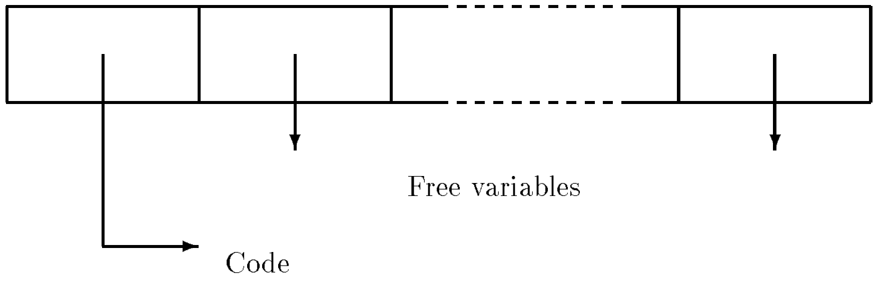
Is `(g x)` a function or not? It depends, of course, on the type of `g` and, since `compose` is polymorphic, this is not statically determined.

For reasons which will become apparent we use the term *closure* to refer to both values and thunks. In the remainder of this section we consider various ways in which closures can be represented, contrasting the STG machine with other designs.

3.1.1 Representing functions

Any implementation of a higher-order language must provide a way to represent a function value. Such a value behaves like a suspended computation: when the value is applied to its arguments, the computation is performed.

The most compact way to represent a function value is as a block of static code (shared by all dynamic instances of the value), together with the values of its free variables. (Such a value is commonly called a *closure*, though we use the term in a wider sense in this paper.) The most direct physical representation of such a closure is a pointer to a contiguous block of heap-allocated storage, consisting of a *code pointer* which points to the static code, followed by (pointers to) the values of the free variables, thus:



This is the representation adopted by many compiled Lisp systems, by SML of New Jersey, and by the Spineless Tagless G-machine. To perform the computation, a distinguished register, the *environment pointer*, is made to point to the closure, and the code is executed. We call this operation *entering a closure*. The code can access its free variables by offsets from the environment pointer, and its arguments by some standard argument-passing convention (eg in registers, on the stack, or in an activation record).

Instead of storing the values of the free variables themselves in the closure, it is possible to store a pointer to a block of free variables, or even to a chain of such blocks. These representations attempt to save storage, at the cost of slowing down access. The Orbit compiler, for example, works hard to choose the best representation for closures, including allocating them on the stack rather than in the heap whenever possible, and sharing one block of free variables between several closures (Kranz [1988]). Apart from the compiler complexity involved, considerable extra care has to be taken in the garbage collector to avoid the space leakage which can occur when a closure captures a larger set of free variables than the closure itself requires. Indeed, Appel's measurements for SML of New Jersey suggest that clever closure-representation techniques gain little, and potentially lose a lot (in space complexity), so he recommends a simple flat representation (Appel [1992, Chapter 12]).

The Three Instruction Machine (TIM) takes another interesting position. Instead of representing a closure by a single pointer, it represents a closure by a *pair* of a code pointer and a pointer to a heap-allocated *frame* (Fairbairn & Wray [1987]). The frame, which is a vector of code-pointer/frame-pointer pairs, gives the values of the free variables of the closure, and may be shared between many closures. These code-pointer/frame-pointer pairs need to be handled very carefully in a lazy system, because they cannot be duplicated without the risk of duplicating work. Proper sharing can still be ensured, but it results in a system remarkably similar to the more conventional one mentioned above (Peyton Jones & Lester [1992, Chapter

4]).

3.1.2 Representing thunks

In a non-strict language, values are passed to functions or stored in data structures in unevaluated form, and only evaluated when their value is actually required. Like function values, these unevaluated forms capture a suspended computation, and can be represented by a closure in the same way as a function value. Following the terminology of Bloss, Hudak & Young [1988], we call this particular sort of closure a *thunk*, a term which goes back to the early Algol implementations of call-by-name (Ingerman [1961]). When the value of the thunk is required, the thunk is *forced*.

A thunk can (in principle) be represented simply by a parameter-less function value, but it is inefficient to do so, because it might be evaluated repeatedly. This duplicated work is avoided by so-called *lazy* implementations as follows: when a thunk is forced for the first time, it is physically *updated* with its value.

There are three main strategies for dealing with updates in lazy implementations:

The naïve reduction model updates the graph after each reduction (Peyton Jones [1987]). (By "reduction" is meant the replacement of an instance of the left-hand side of a function definition by the corresponding instance of its right-hand side.) Apart from a few optimisations, this is the update strategy used by the G-machine (Johnsson [1984]). Its main disadvantage is that a thunk may be updated with another thunk, so the same object may be updated repeatedly, and we do not consider this model further.

The cell model. In the cell model, each closure is provided with a *status flag* to indicate whether it is evaluated or not. The code to force (that is, get the value of) a closure checks the status flag. If the closure is already evaluated, its value is extracted; otherwise the suspended computation is performed (by entering the closure), the value is written into the cell, and the status flag is flipped (Bloss, Hudak & Young [1988]).

The self-updating model, which is used by the STG machine. The cell model places the responsibility for performing the update on the code which evaluates the thunk. The self-updating model instead places this responsibility on the code inside the thunk itself. The code to force a closure simply pushes a continuation on the stack and enters the closure; if the closure is a thunk, it arranges for an update to be performed when evaluation is complete, otherwise it just returns its value. No tests need be performed.

The update overwrites the thunk with a value, *which therefore must also have a code pointer*, because subsequent forces will re-enter the thunk-turned-value. This representation is natural for function values, as we have already discussed, but is something of a surprise for data values. A list cell, for example, is represented by a code pointer together with the two pointers comprising the head and tail of the list. The code pointed to from the list cell simply returns immediately[2]. In effect, the code pointer plays the role of the flag in cell model.

[2] In Section 3.3 we explore variants of this scheme, in which the code for a list cell does rather more than simply return.

Bloss, Hudak & Young [1988] call this model "closure mode", but the implementation they suggest is very much less efficient (in both time and space) than that outlined above, because it is based on a translation into Lisp.

The latter two models each offer scope for optimisation. Consider the cell model, for example. Forcing can be optimised if the compiler can prove that the thunk is certainly already evaluated (or the reverse), because the test on the flag can be omitted (Bloss, Hudak & Young [1988]). Furthermore, if the compiler can prove that there can be no subsequent code forces on the thunk, then it can omit the code which performs the update.

A similar situation holds for the self-updating model. If the compiler can prove that a particular thunk can only be evaluated at most once (which we expect to be quite common), it can create code for the thunk which doesn't perform the update. Unlike the cell model, the self-updating model cannot take advantage of order-of-evaluation analyses.

3.1.3 A uniform representation for closures

As indicated above, the self-updating model strongly suggests that *every heap-allocated object (whether a head normal form or a thunk) is represented uniformly, by a code pointer together with zero or more fields which give the values of the free variables of the code.* The STG machine adopts this uniform representation, as can be seen in the operational semantics (Section 5), where all heap values are represented uniformly by some code together with a sequence of values. Indeed this is why the machine is called "tagless": since all objects have the same form there is no need for a tag to distinguish one kind of object from another (contrast the presentation in Peyton Jones [1987, Chapter 10]). We use the term "closure" to refer to both values and thunks because of their uniform representation.

The decision to use a uniform representation for all closures has other interesting ramifications, which we explore in this section.

Firstly, when a thunk is updated with its value, it is possible that the value will take more space than the thunk. In this case, the thunk must be updated with an *indirection* to the value (Figure 1). This causes no difficulty for the self-updating model, because indirections can be represented by a closure whose code simply enters the target closure. Such indirections can readily be removed during garbage collection (Section 7.3).

In the cell model, either a second test must be made to check for indirections, or alternatively *every* updated thunk must be an indirection. (Figure 2 shows the latter case.) Both methods impose extra overhead.

Secondly, the self-updating model also allows other exceptional cases to be taken care of without extra tests. For example,

- When a thunk is entered, its code pointer can be overwritten with a "black-hole" code pointer. If the thunk is ever re-entered before it is updated, then its value must depend on itself. It follows that the program has entered an infinite loop, and a suitable error message can be displayed. Without this mechanism stack overflow occurs, which is less helpful to the programmer.

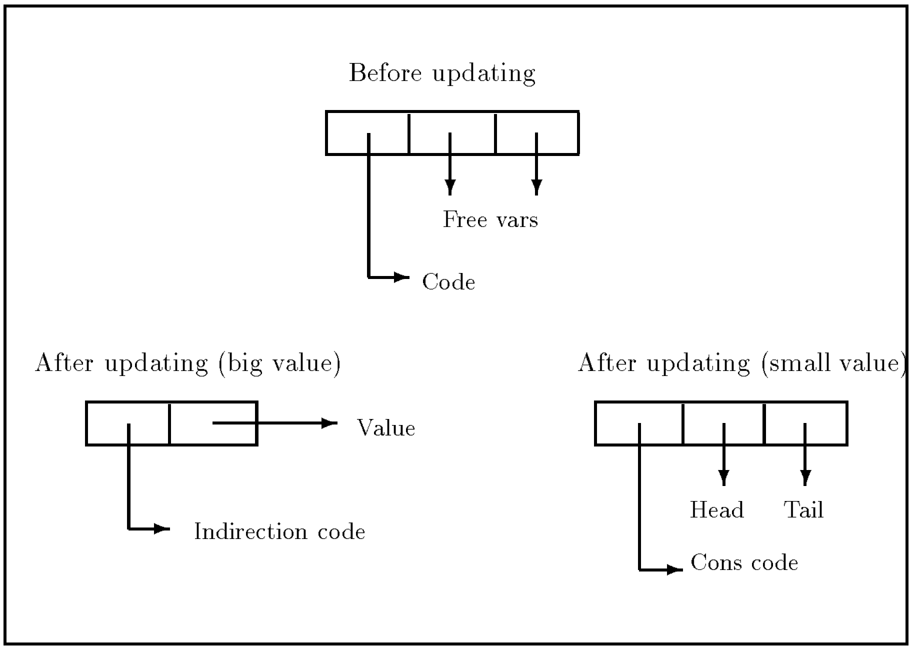


Figure 1: Updating in the self-updating model

- In a system which supports concurrent threads of execution, exactly the same method can be used to synchronise threads. When a thunk is entered, its code pointer is overwritten with a "queue-me" code pointer. If another thread tries to evaluate the thunk before the first thread has completed, the former is suspended and added to a queue of threads attached to the thunk. When the thunk is updated, the queued threads are re-enabled.

- In a system with distributed memory, pointers to remote memory units often have to be treated differently to local pointers. However, it would be very expensive to test for remote-ness whenever dereferencing a pointer! In the self-updating model, a remote pointer can be represented as a special kind of indirection, and no tests for remote pointers need be performed.

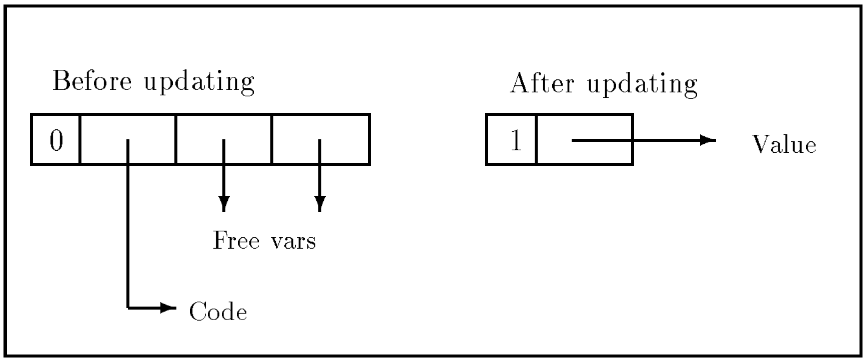


Figure 2: Updating in the cell model

3.2 Function application and the evaluation stack

Higher-order languages, which allow functions as "first-class citizens", present interesting challenges for the compiler-writer. An illuminating way of comparing compilation strategies is to ask how function application is performed.

3.2.1 Currying

The languages in which we are interested make heavy use of *curried functions*. For example, consider the following Haskell function definition:

```
f x y = x
```

`f` is attributed the type `a -> (b -> a)`. That is, `f` may be thought of as a function of one argument, which returns a function which takes the second argument. An application of `f`, say `(f 1 2)`, is short for `((f 1) 2)`. An application of `f` to one argument is perfectly acceptable; for example `(map (f 1) xs)`.

In strict languages like Lisp, Hope and SML, the definition of `f` would usually be of the form

```
f (x,y) = x
```

where `f` is attributed the type `(a,b) -> a`. (We use `(a,b)` to denote the type of pairs of elements of type `a` and `b`.) The function `f` can only be applied to a suitable pair, and cannot be applied to just one argument.

In all of these languages it is possible (and, in SML, easy) to define curried functions (otherwise they would hardly deserve the title "higher-order"), but compilers usually implement the uncurried form much more efficiently, and programmers respond accordingly. There is the inverse cultural tradition in non-strict functional languages, where the additional flexibility allowed by the curried form means that it is usually preferred by programmers, and compilers typically treat curried application as fundamental.

3.2.2 Compiling function application

Compilers from the Lisp tradition usually compile function application as follows: evaluate the function, evaluate the argument, and apply the function value to the argument. When a known function is being applied (as is often the case, especially in Lisp), the "evaluate the function" part becomes trivial. This model for function application, which we call the *eval-apply model*, is invariably used by compilers for strict languages (eg Lisp, Hope, SML and the SECD machine (Henderson [1980]; Landin [1965])). It is also used in some implementations of non-strict languages, except that of course only the function is evaluated before the application (eg the ABC machine (Koopman [1990]), and the $\langle \nu, G \rangle$-machine (Augustsson & Johnsson [1989])).

In contrast, compilers based on lazy graph reduction treat function application as follows: push the argument on an evaluation stack, and tail-call (or *enter*) the function. There is no "return" when the evaluation of the function is complete. We call this the *push-enter model*; it is used by the G-machine, TIM, and the STG machine.

The difference between the two models seems rather slight, but it has a pervasive effect. It is difficult to say in general which of the two is "better". In essentially first-order programs they generate much the same code. For programs which make extensive use of curried functions the push-enter model looks better. For example, consider the (curried) function definition

```
apply3 f x y z = f x y z
```

A Lisp-like compiler would be compelled to evaluate `(f x)`, then evaluate that function applied to `y`, and finally apply the result to `z`. A graph-reduction compiler would just push `x`, `y` and `z` onto the evaluation stack before jumping to the closure for `f`.

3.2.3 The evaluation stack

The main cost of the push-enter model of function application is that the link between a *function body* and an *activation frame* is broken. For example, consider `apply3` again. In the eval-apply model the compiler can allocate an activation frame for `apply3` which is deallocated when the value of `(f x y z)` has been computed. In the push-enter model, all that happens is that three more arguments are pushed on the evaluation stack before jumping to `f`. To put it another way, there are no identifiable moments at which a new activation frame should be allocated or reclaimed.

This pushes the push-enter evaluation model in the direction of having a contiguous evaluation stack, rather than a linked list of heap-allocated activation frames, as exemplified by the New Jersey SML compiler (Appel & Jim [1989]). The idea of heap-allocated activation frames is very appealing, because it makes it easy to implement `call/cc` (Appel & Jim [1989]), parallel threads (Cooper & Morrisett [1990]) and certain debugging mechanisms (Tolmach & Appel [1990]). But all these things can be done by allocating a contiguous stack in medium-sized chunks in the heap, at the price of a little extra complication (Hieb, Dybvig & Bruggeman [1990]; Peyton Jones & Salkild [1989]).

Indeed, performance may well be better using a contiguous stack because of the improved spatial locality, which reduces paging and cache misses. Contiguous allocation of fresh activation records is pessimal for caches, since they have to both fetch useless data (since they are not clever enough to know that it is free space which is about to be allocated) and then write back an activation frame to main memory which is quite likely to be garbage already. Unless one uses generational garbage collection, *and the youngest generation fits entirely within the cache*, using a contiguous stack is likely to have far better cache performance (Appel [1992, Chapter 15]; Wilson, Lam & Moher [1992]). Current cache sizes are still too small to contain a complete generation, but that may change. It would be very interesting to quantify these effects.

The $\langle \nu, G \rangle$-machine is another interesting design compromise (Augustsson & Johnsson [1989]). Here again, there is no contiguous evaluation stack. Instead, working space is allocated in every closure (which the $\langle \nu, G \rangle$-machine calls a frame), and the closures under evaluation are linked together much as heap-allocated activation frames are. The penalties are: space usage is worse, because all closures contain the extra space regardless of whether they are being evaluated or not (and most are not); when a function is evaluated to a partial application, the arguments must be copied from the function's frame to the application's frame; and it is not always possible statically to bound the amount of working space required (unless separate compilation is abandoned), so an exception-checking mechanism is required to deal with the cases where too little has been allocated.

3.3 Data structures

Strongly-typed functional languages such as Haskell encourage the programmer to define many algebraic data types. Even the built-in data types of the language, such as lists, booleans, tuples and (as we will see in Section 4.7) integers, may be regarded as algebraic data types. Here, for example, are representative type declarations for some of them:

```
data Boolean = False | True
data List a = Nil | Cons a (List a)
data Tuple3 a b c = MkTuple3 a b c
data Int = MkInt Int#
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

(Special syntax for lists and tuples is provided by most high-level languages, but not by the STG language.) Data values are *built* using constructors, such as `False`, `Cons`, `MkTuple3`, `Branch`, and *taken apart* using `case` expressions. For example:

```
case t of
        Leaf n       -> e1
        Branch t1 t2 -> e2
```

(In a high-level programming language, data values are usually taken apart using various pattern-matching constructs, but its is well known how to translate such constructs into `case` expressions with simple single-level patterns (Wadler [1987]). We here assume that this translation has been performed.)

These operations of construction and pattern matching are so pervasive in functional programs that they deserve particular attention. Compilers sometimes implement the built-in types (list, tuples, numbers) in special "magic" ways, and the programmer pays a performance penalty for user-defined types. We take the view that the general mechanisms used for user-defined types should be made efficient enough to use for built-in types too. (Lisp, of course, has no user-defined types, so this question does not arise.)

We have already discussed the representation of data values, as a code pointer together with zero or more contents fields. We now turn our attention to the compilation of `case` expressions. Notice that a `case` expression really does two things: it *evaluates* the expression whose value it scrutinises, and then it *selects* the appropriate alternative.

If the cell model is used, the `case` expression must first force the value to be scrutinised. Then it must inspect the value to discover which constructor it is built with, and hence which alternative of the `case` expression should be executed. It follows that each data value must contain a *tag* (usually a natural number) which distinguishes from each other the constructors of the relevant data type. So the sequence of events is:

- Force the value.

- Extract its tag.
- Take a multi-way jump based on the tag[3]
- Bind the names in the pattern of the alternative to the components of the data value.
- Execute the code for the alternative.

In the case of the self-updating model, though, there are more possibilities. Recall that in this model a closure is forced by entering it, *regardless of whether it has been forced before*. So far we have assumed that the code for a constructor always returns immediately. But other variants are possible. It could, for example, load the tag into a register before returning, so that the tag does not need to be represented explicitly at all (Section 9.4.3). Better still, instead of returning to a multi-way jump, the constructor code could return to the appropriate member of a vector of return addresses — we call this a *vectored return* (Section 9.4.3). These *return conventions* can be chosen on independently for each data type.

In effect, the self-updating model used by the STG machine takes advantage of the fact that a data value is only ever forced by a `case` expression. This property is unique to the STG machine. Other lazy implementations treat numeric data types as a special case, which are implicitly forced by the built-in arithmetic operations. In the STG machine, numeric data types are implemented as algebraic data types, and only forced using `case` (Section 4.7).

The idea can be taken one step further. Consider the expression

```
case (f x) of
        Nil       -> e1
        Cons a as -> e2
```

and suppose that `(f x)` evaluates to a `Cons`. The cell model would evaluate `(f x)`, resulting in a heap-allocated `Cons` cell, the components of which would be used in `e2`. But suppose that we use the self-updating model, and that the code for `Cons` puts the head and tail of the `Cons` cell in registers before returning (as well as loading the tag into a register, if the return is not vectored). *Then the* `Cons` *cell need never be allocated in the heap at all!* Since many functions return data values this optimisation seems quite valuable.

In summary, the cell model separates the forcing of a thunk from the case analysis and unpacking performed by a `case` expression. The self-updating model allows these operations to be woven together, which seems to offer interesting opportunities for optimisations. To be fair, these optimisations do complicate updating, as we will see when we roll up our sleeves in Part III, so the benefit is not entirely without cost (Section 10).

3.4 Summary

The single most pervasive design decision in the STG machine is that each closure (including data values) is represented uniformly, and scrutinised only by entering it. The benefits include

[3] Ireland [1992] cleverly avoids the forcing step when it is not necessary, by using the same field to encode the evaluation status flag and constructor tag. The `case` expression's multi-way jump has an extra branch for the case where an unevaluated thunk or indirection is encountered: it forces the thunk and then re-executes the multi-way jump.

- Cheap indirections are available (and are useful when performing updates). They cost nothing when they are not present, and can be eliminated easily during garbage collection.

- Other exceptional conditions (black holes, concurrency, etc) can be handled in the same way.

- A variety of return conventions for constructors are possible, including vectored returns, and returning the components of the constructor in registers. The latter means that data values may not be allocated in the heap at all.

What are the costs? The main one seems to be this: in the common case when a possible thunk turns out to be already evaluated, the self-updating model takes two jumps, one to enter the closure and one to return, while the cell model takes only one (conditional) jump. (As we have seen, though, a jump can often be saved again by using a vectored return.)

Worse, the first jump is to an unknown destination, which means that the code generator cannot keep things in registers. The cell model only incurs these context-switching costs if the thunk is unevaluated. Even so, the cell model may not always win. If there are two or more forces in a row there is the nasty possibility of saving the context, evaluating one thunk, restoring the context, discovering the second thunk is unevaluated, and saving the context for a second time. In this sort of situation it may well be just as good to save the context once and for all at the start of a string of forces, as the self-updating model must do.

There are also some underlying architectural issues. Firstly, indirect jumps are more likely to cause cache misses than (not-taken) conditional jumps. Secondly, modern RISCs are well optimised for taking *conditional* jumps (employed by the cell model), but not for taking *indirect* jumps (which are needed by the self-updating model). In principle, if the jump target address is fetched a few instructions before the jump itself, and the instruction fetch logic interprets the indirect jump directly, no pipeline bubbles need be caused. Most RISCs are not (yet) optimised for this sequence, but the Tera architecture is: it allows a branch target to be prefetched into a register, and thereby supports zero-delay indirect branches (Alverson et al. [1990]).

In short, by always entering a closure when we need its value, we pay a single, fairly modest, up-front cost but get a wide variety of other benefits at no further cost. Whether the benefits outweigh the costs is at present an open question.

Part II: The abstract machine

4 The STG language

The abstract machine code for the Spineless Tagless G-machine is a very austere purely-functional language, called the STG language, whose syntax is given in Figure 3. Virtually every functional-language compiler uses a small purely-functional language as an intermediate code (eg the "enriched lambda calculus" (Peyton Jones [1987]), FLIC (Peyton Jones [1988]), FC (Field & Harrison [1988]), Kid (Ariola & Arvind [1991])).

The distinguishing feature of the STG language is that it has a formal *operational* semantics, expressed as a state transition system, as well as the usual *denotational* semantics. Indeed it is exactly this property which justifies the title "abstract machine code". In particular, the following correspondence between the STG language and operational matters is maintained:

Construct	Operational reading
Function application	Tail call
Let expression	Heap allocation
Case expression	Evaluation
Constructor application	Return to continuation

The salient characteristics of STG code are as follows:

- *All function and constructor arguments are simple variables or constants.* This constraint corresponds to the operational reality that function arguments are prepared (either by constructing a closure or by evaluating them) prior to the call.

  It is easy to satisfy this condition when translating into the STG language, simply by adding new `let` bindings for non-trivial arguments.

- *All constructors and built-in operations are saturated.* This constraint simplifies the operational semantics of STG code. It is easily arranged by adding extra lambdas around an un-saturated constructor or built-in application, thus performing the opposite of η-reduction.

  Notice that, in a higher-order language, we cannot ensure that *every* function application is saturated (that is, gives to the function exactly the number of arguments it expects).

- *Pattern matching is performed only by* `case` *expressions,* and the patterns in `case` expressions are simple one-level patterns. More complex forms of pattern-matching can easily be translated into this form (Wadler [1987]).

  The value scrutinised by a `case` expression can be an arbitrary expression, and is not restricted to be a simple variable or constant. Nothing would be gained by such a restriction, and some performance would be lost because a closure for the expression would be unnecessarily built and then immediately evaluated.

- *There is a special form of binding.* The STG language has a special form of binding, whose general form is

$$f \texttt{ = \{} v_1, \ldots, v_n \texttt{\} \textbackslash} \pi \texttt{ \{} x_1, \ldots, x_m \texttt{\} ->} \ e$$

Program	$prog \rightarrow binds$	
Bindings	$binds \rightarrow var_1$ `=` lf_1 ; ... ; var_n `=` lf_n	$n \geq 1$
Lambda-forms	$lf \rightarrow vars_f$ `\`π $vars_a$ `->` $expr$	
Update flag	$\pi \rightarrow$ `u`	Updatable
	$\mid$ `n`	Not updatable
Expression	$expr \rightarrow$ `let` $binds$ `in` $expr$	Local definition
	$\mid$ `letrec` $binds$ `in` $expr$	Local recursion
	$\mid$ `case` $expr$ `of` $alts$	Case expression
	$\mid$ $var\ atoms$	Application
	$\mid$ $constr\ atoms$	Saturated constructor
	$\mid$ $prim\ atoms$	Saturated built-in op
	$\mid$ $literal$	
Alternatives	$alts \rightarrow aalt_1$; ... ; $aalt_n$; $default$	$n \geq 0$ (Algebraic)
	$\mid$ $palt_1$; ... ; $palt_n$; $default$	$n \geq 0$ (Primitive)
Algebraic alt	$aalt \rightarrow constr\ vars$ `->` $expr$	
Primitive alt	$palt \rightarrow literal$ `->` $expr$	
Default alt	$default \rightarrow var$ `->` $expr$	
	$\mid$ `default ->` $expr$	
Literals	$literal \rightarrow$ `0#` $\mid$ `1#` $\mid$ ...	Primitive integers
	$\mid$ ...	
Primitive ops	$prim \rightarrow$ `+#` $\mid$ `-#` $\mid$ `*#` $\mid$ `/#`	Primitive integer ops
	$\mid$ ...	
Variable lists	$vars \rightarrow$ `{`var_1 , ... , var_n`}`	$n \geq 0$
Atom lists	$atoms \rightarrow$ `{`$atom_1$, ... , $atom_n$`}`	$n \geq 0$
	$atom \rightarrow var \mid literal$	

Figure 3: Syntax of the STG language

It has two readings. From a denotational point of view, the free variables $v_1, \ldots, v_n$ and update flag π are ignored, and the definition binds f to the function $(\lambda x_1 \ldots x_m.e)$. The *arity* of the lambda-form is the number of arguments it has, namely m. We also say that f, the name bound to the lambda form, has arity m.

From an operational point of view, f is bound to a heap-allocated closure, containing a code pointer and (pointers to) the free variables $v_1, \ldots, v_n$. This closure represents the function $(\lambda x_1 \ldots x_m.e)$; when its code is executed, a special register will point to the closure thereby giving access to its free variables.

The right-hand side of a binding is called a *lambda-form*, and is the only site for a lambda abstraction. Notice, though, that the abstraction can have free variables, so no lambda lifting need be performed (Section 4.5).

The update flag on a lambda-form indicates whether its closure should be updated when it reaches its normal form (Section 4.2). We say a lambda-form (or a closure built from it) is *updatable* if its update flag is `u`, and *non-updatable* otherwise.

- *The STG language supports unboxed values.* This aspect is discussed below in Section 4.7.

A STG program is just a collection of bindings. The variables defined by this top-level set of bindings are called *globals*, while all other variables bound in the program are called *locals*. The value of an STG program is the value of the global `main`.

The concrete syntax we use is conventional: parentheses are used to disambiguate; application associates to the left and binds more tightly than any other operator; the body of a lambda abstraction extends as far to the right as possible; and, where the layout makes the meaning clear, we allow ourselves to omit semicolons between bindings and case alternatives.

The STG language is similar in some ways to “continuation-passing style” (CPS), a point we return to in Section 4.8.

4.1 Translating into the STG language

In this section we outline how to translate a functional program into the STG language. We begin with an example, the well-known function `map`. Its definition, in conventional notation (eg Haskell), is as follows:

```
map f []     = []
map f (y:ys) = (f y) : (map f ys)
```

The corresponding STG binding is this:

```
map = {} \n {f,xs} ->
          case xs of
            Nil {}      -> Nil {}
            Cons {y,ys} -> let fy = {f,y} \u {} -> f {y}
                                mfy = {f,ys} \u {} -> map {f,ys}
                           in Cons {fy,mfy}
```

Notice the flattened structure, the explicit argument lists for every call, and the free-variable lists and update flags on each lambda-form. Since `map` itself is a global constant it is not considered to be a free variable of the lambda-form for `mfy`.

In this example, every lambda-form has either no arguments or no free variables. To illustrate the two in combination, consider the following alternative definition for `map`:

```
    map1 f = mf  where  mf [] = []
                        mf (y:ys) = (f y) : (mf ys)
```

Here the recursion is over `mf`, which has free variable `f`. The corresponding STG binding is:

```
map1 = {} \n {f} ->
       letrec
         mf = {f,mf} \n {xs} ->
              case xs of
                Nil {} -> Nil {}
                Cons {y,ys} -> let fy = {f,y} \u {} -> f {y}
                                   mfy = {mf,ys} \u {} -> mf {ys}
                               in Cons {fy,mfy}
       in mf
```

Here, `mf` is an example of a lambda-form with both free variables and arguments. Notice that `mf` is a free variable of its own right-hand side (see Section 4.5).

4.1.1 The general transformation

In general, translation into the STG language involves the following transformations:

- Replace binary application by multiple application.

$$(\ldots((f\ e_1)\ e_2)\ \ldots)\ e_n \quad\Longrightarrow\quad f\ \{e_1, e_2, \ldots, e_n\}$$

  The semantics is still that of curried application, of course, but the STG machine applies a function to all the available arguments at once, rather than doing so one by one.

- Saturate all constructors and built-in operations, by η-expansion if necessary. That is

$$c\ \{e_1, \ldots, e_n\} \quad\Longrightarrow\quad \lambda y_1 \ldots y_m.\ c\ \{e_1, \ldots, e_n, y_1, \ldots, y_m\}$$

  where c is a built-in or constructor with arity $(n+m)$

- Name every non-atomic function argument, and every lambda abstraction, by introducing a `let` expression:

- Convert the right-hand side of each `let` binding into a lambda-form, by adding free-variable and update-flag information.

4.1.2 Identifying the free variables

The transformation to STG code requires the free variables of each lambda-form to be identified. The rule is as follows: a variable must appear in the free variable list of a lambda-form if

1. it is mentioned in the body of the lambda abstraction, and
2. it is not bound by the lambda, and
3. it is not bound at the top level of the program.

Thus, in the first version of `map` in the previous section, `map` does not appear in the free-variable list of `mfy` because it is a global constant. On the other hand, in the second version of `map`, `mf` is a free variable of both itself and `mfy`.

The free-variable rule handles mutual recursion without any further complications. For example, the Haskell definition

```
f x y = fbody
        where
        g1 a = ...a...g2...x...
        g2 b = ...b...g1...y...
```

would transform to the STG definition:

```
f = {} \n {x,y} -> letrec
                     g1 = {g2,x} \n {a} -> ...a...g2...x...
                     g2 = {g1,y} \n {b} -> ...b...g1...y...
                   in
                   fbody
```

The `letrec` builds a pair of closures, each of which points to the other.

The rule above says when a variable *must* appear in a free-variable list of a lambda-form. Of course, any in-scope variable *may* appear (redundantly); surprisingly, there is one situation in which such redundant free variables prove useful — see Section 4.4.

4.2 Closures and updates

In the STG language, `let(rec)` expressions bind variables to lambda-forms. Two pieces of denotationally redundant but operationally significant information are attached to a lambda-form: a list of the free variables of the lambda-form, and an *update flag*. In this section we focus on the update flag.

Updates are an expensive feature of lazy evaluation, whereby the closure representing an unevaluated expression is updated with its (head) normal form when it is evaluated (Section 3.1.2). The aim is to avoid evaluating a particular closure more than once.

In contrast to the G-machine, which performs an update after almost every reduction, the Spineless Tagless G-machine is able to decide *on a closure-by-closure basis* whether updating is required. (It shares this property with TIM and the Spineless G-machine.) The update/no-update decision is controlled by the update flag on a lambda-form: if the update flag is "`u`", the corresponding closure will be updated with its head normal form if it is ever evaluated; if it is "`n`" no update will be performed.

It is clearly safe to set the update flag of every lambda-form to `u`, thereby updating every closure. But we can do much better than this. The obvious question is: to which lambda-forms can we safely assign an update flag of "`n`", without losing the single-evaluation property? We explore this question by classifying lambda-forms into distinct classes:

Manifest functions. A manifest function is a lambda-form with a non-empty argument list. For example, `map` and `mf` are manifest functions in the examples of the previous section. Manifest functions do not require updating because they are already in head normal form.

Partial applications. A partial application lambda-form is of the form

$$vs \ \texttt{\textbackslash n} \ \{\} \ \texttt{->} \ f \ \{x_1, \ldots, x_m\}$$

where f is known to be a manifest function taking more than m arguments. Like manifest functions, partial applications are already in head normal form, and hence do not require updating. Both manifest functions and partial applications are of course function values.

Partial applications sometimes appear directly in programs, but they also arise as a result of performing updates (Section 5.6).

Only lambda-forms in precisely the form given above are classified as partial applications. For example, the lambda-form

```
{x,y} \u {} -> let z = ...
                in f {z}
```

is *not* classified as a partial application, because its body is not in the required form. There is a good reason for this: if a closure built from this lambda-form was not updated, the closure for `z` would be re-built each time it was entered.

Constructors. A constructor is a lambda-form of the form

$$vs \ \texttt{\textbackslash n} \ \{\} \ \texttt{->} \ c \ \{x_1, \ldots, x_m\}$$

where c is a constructor. (Since constructor applications are always saturated in the STG language, c is bound to have arity m.) The update flag on a constructor is always `n` (no update).

Thunks. The remaining lambda-forms, those with an empty argument list but not of the special form of a partial application or a constructor, are called *thunks*. The lambda-forms `mfy` and `fy` are examples of thunks in the previous section.

Since thunks are not in normal form, it appears at first that they should all have their update flag set to `u`. However, *if the compiler can prove that a thunk can be evaluated* *at most once* then it is safe to set its update flag to n, thereby allowing the update to be omitted.

For example, consider the following definition:

```
f = {} \n {p,xs} -> let j = {p} \n {} -> factorial p
                    in
                    case xs of
                        Nil {}      -> + {j,1}
                        Cons {y,ys} -> + {j,2}
```

Here it is clear that j will be evaluated at most once, so its closure does not need to be updated.

In summary, updates are never required for functions, partial applications and constructors; and may in addition sometimes be omitted for thunks.

The analysis phase which determines which thunks need not be updated is called *update analysis*. Not much work seems to have been done on this topic, but we are working on a simple update analyser.

4.3 Generating fewer updatable lambda-forms

The translation from the Core to STG language is largely straightforward (apart from the update analysis, which can of course be omitted by flagging all thunks as updatable). There is an important opportunity to reduce the incidence of updates, which concerns constructors and partial applications. Consider the Haskell expression

```
let xs = y1 : (y2 : (y3 : []))
in ...
```

A straightforward translation into the STG language gives

```
let xs = {y1,y2,y3} \u {} ->
         let t1 = {y2,y3} \u {} ->
                  let t2 = {y3} \u {} ->
                           let t3 = {} \n {} -> Nil {}
                           in Cons {y3,t3}
                  in Cons {y2,t2}
         in Cons {y1,t1}
in ...
```

Three updatable thunks have been built which will subsequently be updated when (and if) xs is traversed. An alternative, and usually superior, translation is this:

```
let    t3 = {} \n {} -> Nil {}
in let t2 = {y3,t3} \n {} -> Cons {y3,t3}
in let t1 = {y2,t2} \n {} -> Cons {y2,t2}
in let xs = {y1,t1} \n {} -> Cons {y1,t1}
in ...
```

No updatable thunks are built at all. The only bad thing about this translation is that if `xs` was to be discarded without being traversed then the work of constructing all four constructor closures would have been wasted. (This is a strictly bounded amount of work, however.)

Just the same alternative translation is possible when a known function is applied to too few arguments; the `let` bindings for the argument expressions can be lifted up a level so that a partial-application lambda-form remains, which does not need to be updated.

In general,

- Updates can be omitted for parameterless lambda-forms if the body is a head normal form.
- Opportunities for this improvement may be enhanced by moving `let(rec)` bindings from the lambda-form to its enclosing context. More generally, any small, bounded, computation may be moved from a lambda-form to its enclosing context to expose a head normal form, and thereby avoid an update.

4.4 Standard constructors

The final form of the example in the previous section had three lambda-forms of the form

$$\{x, xs\} \texttt{ \textbackslash n \{\} -> Cons } \{x, xs\}$$

for various x and xs. Because they all have the same shape, they can clearly all share a single code pointer. In general, a lambda-form of the form

$$\{x_1, \ldots, x_m\} \texttt{ \textbackslash n \{\} -> } c\ \{x_1, \ldots, x_m\}$$

where c is a constructor of arity m, is called a *standard constructor*. All such lambda-forms for a particular constructor c can share common code.

The free-variable list of a global definition is usually empty, since the global can only mention other globals, and is represented by a closure consisting only of a code pointer. However, consider the following global Haskell definition:

```
aList = [thing]
thing = ...
```

A straightforward translation to the STG language gives

```
aList = {} \n Cons {thing,nil}
nil = {} \n Nil {}
thing = ...
```

This is perfectly correct, but it means having special-purpose code for `aList`, which has references to `thing` and `nil` wired into it. If `aList` was a list with several items in it, each cell in the list would have a separate code sequence! An alternative translation, is

```
aList = {thing,nil} \n Cons {thing,nil}
nil = {} \n Nil {}
thing = ...
```

The lambda-form for `aList` now has free variables which are not strictly necessary, but the payoff is that the lambda-form is now a standard constructor, and can use the standard code for `Cons`. Instead of being represented by a code pointer alone, `aList` is now represented by the `Cons` code together with pointers to the globals `thing` and `nil`.

We apply this idea throughout, not just at top level, to make sure that every lambda-form whose body is a simple constructor application is a standard constructor.

For nullary constructors, such as `Nil`, it is not only possible to share its *code*, but also to share its *closure*. Thus, in the above example, the `nil` global can be shared by all occurrences of `Nil` in the program.

4.5 Lambda lifting

Lambda lifting is a process whereby all function definitions are lifted to the top level, by making their free variables into extra arguments (Johnsson [1985]; Peyton Jones & Lester [1991]). In a lambda-lifted program each lambda-form has either no free variables or no arguments. In contrast to most other abstract machines, the STG machine does not require the program to be lambda lifted; a right-hand side can have both free variables and arguments.

The operational difference between the two is fairly slight. Consider the following STG language definition:

```
f = {} \n {x1,x2,x3} -> let z = {x1,x3} \n {y} -> zbody
                        in fbody
```

in which the lambda-form for `z` has the free variables `x1` and `x3`, and argument `y`. If lambda lifting is performed, a new global function (or supercombinator) `$z` is introduced, giving:

```
$z = {} \n {x1,x3,y}  -> zbody
f  = {} \n {x1,x2,x3} -> let z = {x1,x3} \n {} -> $z {x1,x3}
                         in fbody
```

Now the program has only thunks (like `z`) and supercombinators (like `f` and `$z`). Operationally, what happens is that when `z` is entered, it pushes its two free variables, `x1` and `x3`, onto the evaluation stack and jumps to `$z`. In the original version, which the STG machine can execute directly, the free variables can be used directly from the closure itself.

In short, the local environment in which the STG machine executes consists of two parts (Section 5.2): values held in the closure just entered (its free variables), and values held on the stack (its arguments). This two-level environment reduces somewhat the movement of values from the heap to the stack, but it is not yet clear whether this is a big improvement or only a marginal one.

4.6 Full laziness

Consider the binding

```
f = {x} \n {y} -> let z = {x} \u {} -> ez
```

```
                    in ef
```

where ez and ef are arbitrary expressions. Suppose x and y are both free in ef, but only x is free in ez. Since the lambda-form for z does not have y as a free variable, this is equivalent to the pair of bindings

```
z = {x} \u {} -> ez
f = {x,z} \n {y} -> ef
```

Furthermore, the latter form may save work if f is applied many times, because z will be instantiated only once rather than once for each call of f. In general, each binding can be moved outwards until its immediately enclosing lambda abstraction binds one of the free variables of the binding. This transformation is called the *full laziness* transformation, and is described in detail by Peyton Jones & Lester [1991].

4.7 Arithmetic and unboxed values

In a non-strict functional language implementation, when a variable is bound, it is generally bound to an unevaluated closure allocated in the heap. When the value of the variable is required, the closure to which it points is evaluated, and the closure is overwritten with the resulting value. Further evaluations of the same closure will find the value immediately.

This evaluation model means that all numbers are represented by a pointer to a heap-allocated closure, or "box", which contains either information which enables the number to be computed, or (if the closure has been evaluated) the actual value of the number. We call the "actual value" an *unboxed value*; it can be manipulated directly by the instruction set of the machine.

The uniform boxed representation makes arithmetic horribly expensive. A simple addition, which takes one instruction in a conventional system, requires a sequence of instructions to: evaluate the two operands, fetch their values, add them, allocate a new box for the result, and place the result in it.

One of the innovative features of our compiler is that *unboxed values are explicitly part of the Core and STG languages.* That is, variables may be bound to unboxed values, functions may take unboxed values as arguments and return them as results, unboxed values may be stored in data structures, and so on. The main motivation for this approach is that we can then be explicit about the steps involved in (say) addition. To begin with, we declare the following data type:

```
data Int = MkInt Int#
```

This declares the data type of (boxed) integers, Int, as an algebraic data type with a single constructor, MkInt. The latter has a single argument of type Int#, the type of *unboxed* integers. So the value (MkInt 3#) represents the boxed integer 3 (3# stands for the unboxed constant 3, of type Int#).

Now, given the expression (e1 + e2), say, we can rewrite it like this:

```
case e1 of
MkInt x# -> case e2 of
```

```
MkInt y# -> case (x# +# y#) of
                t# -> MkInt t#
```

The outer two `case` expressions evaluate `e1` and `e2` respectively, while the inner `case` expresses the fact that `x#` and `y#` are added, and *then* their result `t#` is boxed with a `MkInt` constructor. (By convention, we use a trailing `#` for identifiers whose values or results are primitive. This is just for human readability: the identifiers `+` and `+#` are distinct, but the `#` is not otherwise recognised specially by the compiler.)

It turns out that this simple idea allows several optimisations which hitherto were buried in the code generator to be reformulated as program transformations. Furthermore, the idea can be generalised in a number of directions, such as allowing general algebraic data types with unboxed components (rather than just `Int`). All of this is discussed in detail in Peyton Jones & Launchbury [1991].

For the purposes of this paper, it suffices to establish the following facts:

- Data types are divided into two kinds: *algebraic data types* are introduced by explicit `data` declarations, while *primitive data types* are built into the system. Values of primitive type can be manipulated directly by machine instructions, and are always unboxed. For example, `Int` is an algebraic type, while `Int#` is primitive. For the purpose of this paper, it suffices to identify primitive types with unboxed types, though the generalisations discussed in Peyton Jones & Launchbury [1991] permit unboxed algebraic types as well.

- All literal constants are of primitive type; literals of algebraic type are expressed by giving an explicit application of a constructor.

- All arithmetic built-in operations operate over primitive values (for example `+#` above). Definitions for functions operating over non-primitive (ie algebraic) values can be expressed directly in the STG language, and hence do not need to be built in.

- Values of unboxed type need not be the same size as a pointer. For example, `Double#`, the type of double-precision floating-point numbers, occupy 64 bits while pointers usually occupy 32 bits. As a result, *polymorphic functions can take only arguments of boxed type*, because arguments must be passed to such functions in a uniform representation. (Even if unboxed values were always the same size as a pointer there would still be a difficulty for the garbage collector in distinguishing a pointer from a non-pointer.)

- A `let` or `letrec` expression cannot bind a variable of unboxed type. Such a binding is instead made using a `case` expression. The reason for this is that when a variable of unboxed type is bound, the expression to which it is bound must be evaluated immediately; the whole point about unboxed values is that they cannot be represented by as-yet-unevaluated closures.

  In other words, in the STG language, `case` expressions perform evaluation, while `let` and `letrec` build closures. This uniform semantics gives rise to uniform transformation laws; for example, a `let` expression whose bound variable is not used can always be elided.

  For the same reason, the global (top-level) bindings of an STG program cannot bind values of unboxed type.

- There are two forms of `case` expression, as the the syntax of Figure 3 describes. One takes apart a value of an algebraic data type, while the other performs case analysis on a value of primitive type.

4.8 Relationship to CPS conversion

Transformation to continuation-passing style (CPS) is a technique which has been used to good effect in several compilers for strict (call-by-value) languages (Appel [1992]; Fradet & Metayer [1991]; Kelsey [1989]; Kranz [1988]; Steele [1978]). Though the STG language is lazy, it has much the same flavour as CPS: nested constructs are flattened to an explicit sequence of simple operations, so that the flow of control is manifest, and there is a direct relationship between the remaining language constructs and individual machine operations.

To make the connection explicit, the following table shows the approximate correspondence between the constructs of the CPS language used by Appel (Appel [1992]), and those of the STG language.

<table>
<tr><th>Operation</th><th>Appel CPS form</th><th>STG form</th></tr>
<tr><td>Application</td><td>APP</td><td>Application</td></tr>
<tr><td>Local function definition</td><td>FIX</td><td rowspan="3"><code>let(rec)</code></td></tr>
<tr><td>Record construction</td><td>RECORD</td></tr>
<tr><td>Thunk construction</td><td>—</td></tr>
<tr><td>Forcing of data values</td><td>—</td><td rowspan="3"><code>case</code> on algebraic types</td></tr>
<tr><td>Selection of alternative</td><td>SWITCH</td></tr>
<tr><td>Extraction of record components</td><td>SELECT</td></tr>
<tr><td>Primitive operations</td><td>PRIMOP</td><td><code>case</code> on primitive types</td></tr>
</table>

There are a few minor differences between Appel's CPS and the STG language. Firstly, CPS-based implementations usually unbundle `case` expressions into forcing, multi-way selection, and extraction of the components of the data value. These are all bundled up together in `case` expressions which, as we have seen, can be used to advantage by the STG machine. Secondly, the STG language uses a single construct, `let(rec)`, to allocate function-valued and data-valued closures, thus allowing arbitrary mutual recursion between the two. It is not so clear how to achieve this using Appel's form of CPS.

There is a much more important difference though: the STG language is not a continuation-passing style! In CPS, every user-defined function is given an extra parameter, namely the continuation to apply to its result. For example, assuming a continuation `k`, the expression

```
(f x) + y
```

would be converted to the CPS form

```
f x (\fx. + fx y (\r. k r))
```

The call to `f` is made into a tail call, passing to `f` an extra argument, the continuation `(\fx. + fx y (\r. k r))`. This continuation says what to do after `(f x)` has been computed, namely add the result to `y` and pass that value to `k`. In contrast, the STG form of the same expression is:

```
case (f x) of
MkInt x# -> case y of
            MkInt y# -> case (x# +# y#) of
                        r# -> MkInt r#
```

The continuation to the call to `f` is passed *implicitly*; when evaluation of `f x` is complete, control returns to the second `case` expression. The second `case` evaluates `y`, which of course is not necessary in Appel's world since SML is strict. A lazy version of CPS would require the suspended computation inside a thunk to be a function taking a continuation as its argument. So the CPS form would really be

```
f x (\fx. force y (\yr. + fx yr (\r. k r)))
```

where `force` is the function which forces a thunk (its first argument) by applying it to `force`'s second argument (the continuation). This code is strikingly similar to the STG form above.

This difference in the way in which continuations are handled clearly distinguishes CPS from the STG language, but it is quite difficult to pin down all the implications of the difference. For example, the CPS version has a natural stack-less implementation, since every call is a tail call. On the other hand, it may thereby incur the cost of heap-allocating the closure for the continuation, and passing it as an argument to `f`. The STG version suggests a stack-based implementation, since the current activation frame contains the environment in which the continuation should be executed. But of course, either implementation is possible from both styles.

The STG style also seems to be more natural for curried function application. Consider the call `(f x y)`, which is left unchanged by the conversion to the STG language. If converted to CPS (assuming that the call itself has continuation `k`), this would generate something like:

```
f x (\w. w k y)
```

This is a rather expensive and clumsy compilation for an ordinary function application! We expect curried function application to be pervasive, so the STG language provides it as primitive.

Of course, this imposes an extra requirement on the code generator for the STG language: it must cope with functions applied to more or fewer arguments than they are expecting. (For example `f` might take one argument, `x`, do a lot of computation, and finally reduce to a function which takes the second argument `y`.) As Section 5 will show, though, graph reduction gives a natural way to provide this functionality.

In summary, the STG language has a similar flavour to CPS, but is a little less extreme. So far we have not discovered any opportunities for optimisation which are exposed by CPS but hidden by the STG language. (Consel & Danvy [1991] show that transforming the source program to CPS may improve the accuracy of some analyses; we have not investigated whether or not the STG language has a similar property.)

5 Operational semantics of the STG language

The STG language is the abstract machine code for the STG machine. In this section we give a direct operational semantics for the STG language using a state transition system.

A state transition semantics specifies (a) an initial state for the machine, and (b) a series of state transition rules. Each rule specifies a set of source states and the corresponding target states after the transition has taken place. The set of source states is specified implicitly, using pattern-matching and guard conditions; if a state is in the source set for a given transition rule we say that the rule *matches* the state. At most one transition rule should match any given state, and if no rule matches, the machine halts.

The state has five components:

1. the *code*, which takes one of several forms, given below;
2. the *argument stack*, *as*, which contains *values*;
3. the *return stack*, *rs*, which contains *continuations*;
4. the *update stack*, *us*, which contains *update frames*;
5. the *heap*, *h*, which contains (only) *closures*;
6. the *global environment*, σ, which gives the addresses of all closures defined at top level.

Sequences are used extensively in what follows. They are denoted using curly brackets, thus $\{a_1, \ldots, a_n\}$. The empty sequence is denoted $\{\}$; if *as* and *bs* are two sequences then $as + bs$ is their concatenation; and $a : as$ denotes the sequence obtained by adding the item a to the beginning of the sequence *as*. The length of a sequence *as* is denoted $length(as)$.

A *value* takes one of the following forms:

Addr a	A heap address
Int n	A primitive integer value

In the operational semantics, values are tagged with *Addr* and *Int* and so on to distinguish these different kinds of value. We discuss later ways to avoid actually implementing this tagging in a real implementation (Section 8). We could add further forms of value for other primitive data types, such as floating-point numbers, but they are handled exactly analogously to integers, so we omit them to reduce clutter.

We use $w, w_1, \ldots$, to range over values, and *ws* to range over sequences of values.

The *argument stack*, *as*, is just a sequence of values. The "top" of the stack is the beginning of the sequence. The return stack and update stack will be dealt with later (Sections 5.4 and 5.6 respectively).

The *heap*, *h*, is a mapping from *addresses*, ranged over by $a, a_1, \ldots$, to *closures*. Every closure is of the form

$$(vs\ \texttt{\textbackslash}\pi\ xs\ \texttt{->}\ e)\ ws$$

Intuitively, the lambda-form (vs \π xs `->` e) denotes the code of the closure, while the sequence of values ws gives the value of each of the free variables vs. (We use π to range over update flags, which can be either `u` or `n`.) This is exactly the uniform representation discussed in Section 3.1.3.

The *global environment* component of the state, σ, maps the name of each variable bound at the top level of the program to the address of its closure. These closures can all be allocated once and for all before execution begins. (Indeed, unlike the other components, σ does not change during execution.) The STG machine is unusual in binding globals to *closures* rather than to code sequences. It is important to do so, however, because a global may be updatable, so there must be a closure to update!

As we discussed earlier, it is possible to share the code for standard-constructor closures (Section 4.4). In the special case of constructors with no arguments (such as `Nil`) it is possible to share not just the code for the closure, but the closure itself. For example, all references to `Nil` can use the address of a single global closure. This is easily done by adding niladic constructors as a possible form of *atom* (Figure 3), and extending σ with the address of a suitable closure for each niladic constructor. For the sake of simplicity, we do not perform this optimisation in the operational semantics which follows.

Finally, the *code* component of the state takes one of the following four forms, each of which is accompanied by its intuitive meaning:

$Eval\ e\ \rho$	Evaluate the expression e in environment ρ and apply its value to the arguments on the argument stack. The expression e is an arbitrarily complex STG-language expression.
$Enter\ a$	Apply the closure at address a to the arguments on the argument stack.
$ReturnCon\ c\ ws$	Return the constructor c applied to values ws to the continuation on the return stack.
$ReturnInt\ k$	Return the primitive integer k to the continuation on the return stack.

The *local environment*, ρ, maps variable names to *values*. The notation $\rho[v \mapsto w]$ extends the map ρ with a mapping of the variable v to value w. This notation also extends in the obvious way to sequences of variables and values; for example $\rho[vs \mapsto ws]$.

The *val* function takes an atom (Figure 3) and delivers a value:

$$
\begin{array}{llllll}
val & \rho & \sigma & k & = & Int\ k \\
val & \rho & \sigma & v & = & \rho\ v \quad \text{if } v \in dom(\rho) \\
 & & & & = & \sigma\ v \quad \text{otherwise}
\end{array}
$$

If the atom is a literal k, *val* returns a primitive integer value. If it is a variable, *val* looks it up in ρ or σ as appropriate. *val* extends in the obvious way to sequences of variables: $val\ \rho\ \sigma\ vs$ is the sequence of values to which $val\ \rho\ \sigma$ maps the variables vs.

5.1 The initial state

We begin by specifying the initial state of the STG machine. The general form of an STG program is as follows:

$$\begin{array}{ll} g_1 & \texttt{=}\ vs_1\ \texttt{\\}\pi_1\ xs_1\ \texttt{->}\ e_1 \\ \ldots & \\ g_n & \texttt{=}\ vs_n\ \texttt{\\}\pi_n\ xs_n\ \texttt{->}\ e_n \end{array}$$

One of the g_i will be `main`. Given this program, the corresponding initial state of the machine is:

$$\begin{array}{llllll} & \textit{Arg} & \textit{Return} & \textit{Update} & & \\ \textit{Code} & \textit{stack} & \textit{stack} & \textit{stack} & \textit{Heap} & \textit{Globals} \\ & & & & & \\ \textit{Eval}\ (\texttt{main}\ \{\})\ \{\} & \{\} & \{\} & \{\} & h_{init} & \sigma \end{array}$$

$$\begin{array}{lll} \text{where}\ \sigma & = & \left[\begin{array}{l} g_1 \mapsto (\textit{Addr}\ a_1) \\ \ldots \\ g_n \mapsto (\textit{Addr}\ a_n) \end{array}\right] \\ h_{init} & = & \left[\begin{array}{l} a_1 \mapsto (vs_1\ \texttt{\\}\pi_1\ xs_1\ \texttt{->}\ e_1)\ (\sigma\ vs_1) \\ \ldots \\ a_n \mapsto (vs_n\ \texttt{\\}\pi_n\ xs_n\ \texttt{->}\ e_n)\ (\sigma\ vs_n) \end{array}\right] \end{array}$$

We write a machine state as a horizontal row of its components, sometimes with auxiliary definitions (as here) introduced by a "where" clause. In this initial state, the code component says that `main` is to be evaluated in the empty local environment; the argument, return and update stacks are empty; the initial heap, h_{init}, contains a closure for each global; and the global environment, σ, binds each global to its closure.

Notice that the values in the range of the global environment are all *addresses*. This reflects the fact that global variables are always boxed.

5.2 Applications

We begin the main operational semantics with the rule for applications.

$$(1)\quad \begin{array}{|lll|} \hline & \textit{Eval}\ (f\ xs)\ \rho & as\ \ rs\ \ us\ \ h\ \ \sigma \\ & & \\ \text{such that} & \textit{val}\ \rho\ \sigma\ f = \textit{Addr}\ a & \\ & & \\ \Longrightarrow & \textit{Enter}\ a & (\textit{val}\ \rho\ \sigma\ xs) +\!\!+\ as\ \ rs\ \ us\ \ h\ \ \sigma \\ \hline \end{array}$$

The top line of the rule gives the state before the transition, while the bottom line gives the state afterwards. We use a pattern-matching notation for the top line. In this case, the rule only matches if the code component is an *Eval* of an expression of the given form. The "such that" clause further constrains the rule to the case where f is bound to the address of a closure (and not to a primitive value).

The rule says that to perform a tail call, the values of the arguments are put on the argument stack, and the value of the function is entered. The function is expected to be a closure; the other case, when f is not an address but rather is a primitive value instead, is dealt with in Section 5.5. Notice that the local environment is discarded at this point; in general, the local environment only has a very local lifetime.

The next thing to discuss is the rule for entering a closure. We give only the rule for entering *non-updatable* closures; the rule for updatable closures is given in Section 5.6.

$$
(2)\quad
\begin{array}{|lll|}
\hline
 & \mathit{Enter}\ a\quad \mathit{as}\ \mathit{rs}\ \mathit{us}\ h[a \mapsto (\mathit{vs}\ \texttt{\textbackslash n}\ \mathit{xs}\ \texttt{->}\ e)\ \mathit{ws}_f] & \sigma \\
\text{such that} & \mathit{length}(\mathit{as}) \geq \mathit{length}(\mathit{xs}) & \\
\Longrightarrow & \mathit{Eval}\ e\ \rho\ \mathit{as}'\ \mathit{rs}\ \mathit{us}\ h & \sigma \\
\text{where} & \begin{array}{lcl} \mathit{ws}_a +\!\!+\ \mathit{as}' & = & \mathit{as} \\ \mathit{length}(\mathit{ws}_a) & = & \mathit{length}(\mathit{xs}) \\ \rho & = & [\mathit{vs} \mapsto \mathit{ws}_f,\ \mathit{xs} \mapsto \mathit{ws}_a] \end{array} & \\
\hline
\end{array}
$$

When a non-updatable closure is entered, the local environment is constructed by binding its free variables to the values, ws_f, found in the closure, and its arguments to the values, ws_a, found on the stack. Then the body of the closure is evaluated in this environment. In this rule we use a “where” clause to give values to variables used in the result state of the rule.

5.3 `let(rec)` expressions

As mentioned earlier, a `let` expression constructs one or more closures in the heap.

$$
(3)\quad
\begin{array}{|lll|}
\hline
 & \mathit{Eval} \begin{pmatrix} \texttt{let}\ \ x_1\ \texttt{=}\ \mathit{vs}_1\ \texttt{\textbackslash}\pi_1\ \mathit{xs}_1\ \texttt{->}\ e_1 \\ \qquad \ldots \\ \qquad x_n\ \texttt{=}\ \mathit{vs}_n\ \texttt{\textbackslash}\pi_n\ \mathit{xs}_n\ \texttt{->}\ e_n \\ \texttt{in}\ e \end{pmatrix} \rho\ \mathit{as}\ \mathit{rs}\ \mathit{us}\ h & \sigma \\
\Longrightarrow & \mathit{Eval}\ e\ \rho' \qquad\qquad\qquad\qquad \mathit{as}\ \mathit{rs}\ \mathit{us}\ h' & \sigma \\
\text{where} & \rho' \ = \rho[x_1 \mapsto \mathit{Addr}\ a_1, \ldots, x_n \mapsto \mathit{Addr}\ a_n] & \\
 & h' \ = h \begin{bmatrix} a_1 \mapsto (\mathit{vs}_1\ \texttt{\textbackslash}\pi_1\ \mathit{xs}_1\ \texttt{->}\ e_1)\ (\rho_{rhs}\ \mathit{vs}_1) \\ \ldots \\ a_n \mapsto (\mathit{vs}_n\ \texttt{\textbackslash}\pi_n\ \mathit{xs}_n\ \texttt{->}\ e_n)\ (\rho_{rhs}\ \mathit{vs}_n) \end{bmatrix} & \\
 & \rho_{rhs} = \rho & \\
\hline
\end{array}
$$

The rule for `letrec` is almost identical, except that ρ_{rhs} is defined to be ρ' instead of ρ.

5.4 Case expressions and data constructors

The *return stack* is used for the first time when we come to `case` expressions. Given the expression

$$\texttt{case}\ e\ \texttt{of}\ alts$$

the operational interpretation is "push a continuation onto the return stack, and evaluate e". When the evaluation of e is complete, execution will resume at the continuation, which then decides which alternative to execute. The rule for `case` follows fairly directly:

$$(4)\quad \boxed{\begin{array}{llllllll} & Eval\ (\texttt{case}\ e\ \texttt{of}\ alts)\ \rho & as & & rs & us & h & \sigma \\ \Longrightarrow & Eval\ e\ \rho & & as & (alts, \rho) : rs & us & h & \sigma \end{array}}$$

The continuation is a pair $(alts, \rho)$; the alternatives $alts$ say what to do when evaluation of e completes, while the environment ρ provides the context in which to evaluate the chosen alternative. We will have more to say about how this expensive-looking environment saving is performed later, in Section 9.4.1.

The other side of the coin is the rules for constructors and literals. Presumably e eventually evaluates to either a constructor or a literal, at which point the continuation must be popped from the return stack and executed. The rules for constructors and literals each use an intermediate state, *ReturnCon* and *ReturnInt* respectively, just as the rule for function application uses *Enter*. Primitive values are dealt with in the next section, while the rules for constructors are given next.

Evaluating a constructor application simply moves into the *ReturnCon* state:

$$(5)\quad \boxed{\begin{array}{llllll} & Eval\ (c\ xs)\ \rho & as & rs & us & h\ \sigma \\ \Longrightarrow & ReturnCon\ c\ (val\ \rho\ \sigma\ xs) & as & rs & us & h\ \sigma \end{array}}$$

The rules for *ReturnCon* return to the appropriate continuation taken from the return stack:

$$(6)\quad \boxed{\begin{array}{lllllll} & ReturnCon\ c\ ws & as\ (\ldots;\ c\ vs\ \texttt{->}\ e;\ \ldots,\ \rho) : rs & us & h & \sigma \\ \Longrightarrow & Eval\ e\ \rho[vs \mapsto ws] & as\qquad\qquad\qquad\qquad\qquad rs & us & h & \sigma \end{array}}$$

Provided that the continuation on the return stack contains a pattern $c\ vs$ whose constructor c is the same as that being evaluated, we just evaluate the right-hand side of that alternative, in the saved environment ρ augmented with bindings for the variables vs to the values of the actual arguments to c.

If there is no such alternative, the default alternative is taken. The rule for this is easy when no variable is bound in the default case:

$$(7)\quad \boxed{\begin{array}{lllll} & ReturnCon\ c\ ws & as & \left(\begin{array}{l} c_1\ vs_1\ \texttt{->}\ e_1\,; \\ \ldots; \\ c_n\ vs_n\ \texttt{->}\ e_n\,; \\ \texttt{default}\ \texttt{->}\ e_d \end{array}\ ,\ \rho\right) & : rs\ \ us\ \ h\ \ \sigma \\ \text{such that}\ \ c \neq c_i & (1 \leq i \leq n) & & & \\ \Longrightarrow & Eval\ e_d\ \rho & as & & rs\ \ us\ \ h\ \ \sigma \end{array}}$$

However, if a variable v is bound by the default, we need to heap-allocate a constructor closure to which to bind v, thus:

$$
\begin{array}{l|lll}
 & & \mathit{ReturnCon}\ c\ \mathit{ws}\ \ \mathit{as}\ \left(\begin{array}{l}c_1\ \mathit{vs}_1\ \texttt{->}\ e_1\,\texttt{;}\\ \ldots\texttt{;}\\ c_n\ \mathit{vs}_n\ \texttt{->}\ e_n\,\texttt{;}\\ v\ \texttt{->}\ e_d\end{array},\ \rho\right) : \mathit{rs}\ \ \mathit{us}\ \ h\ \ \sigma \\
 & \text{such that} & c \neq c_i \qquad (1 \leq i \leq n) \\
(8) & \Longrightarrow & \mathit{Eval}\ e_d\ \rho[v \mapsto a]\ \ \mathit{as} \qquad\qquad \mathit{rs}\ \ \mathit{us}\ \ h'\ \ \sigma \\
 & \text{where} & h' = h[a \mapsto (\mathit{vs}\ \texttt{\textbackslash n}\ \texttt{\{\}}\ \texttt{->}\ c\ \mathit{vs})\ \mathit{ws}] \\
 & & \mathit{vs} \text{ is a sequence of arbitrary distinct variables} \\
 & & \mathit{length}(\mathit{vs}) = \mathit{length}(\mathit{ws})
\end{array}
$$

This rule is a little complicated, and a simple program transformation can eliminate the variable-binding form of default from the language (for algebraic `case` expressions, anyway):

$$
\texttt{case}\ e\ \texttt{of}\ \ldots\texttt{;}\ v\ \texttt{->}\ b \Longrightarrow \begin{array}{l}\texttt{let}\ v\ \texttt{=}\ \mathit{xs}\ \texttt{\textbackslash u}\ \texttt{\{\}}\ \texttt{->}\ e\\ \texttt{in}\\ \texttt{case}\ v\ \texttt{of}\ \ldots\texttt{;}\ \texttt{default}\ \texttt{->}\ b\end{array}
$$

In implementation terms this version is a little less efficient, because a closure for v will be allocated and then updated, whereas using Rule 8 simply allocates the constructor in its final form.

Lastly, if there is no match and no default alternative, no rule matches, which is interpreted as failure.

5.5 Built in operations

In this section we give the extra rules which handle primitive values. The rule for evaluating a primitive literal, k, enters the *ReturnInt* state:

$$
(9)\ \boxed{\begin{array}{ll} & \mathit{Eval}\ k\ \rho \qquad \mathit{as}\ \ \mathit{rs}\ \ \mathit{us}\ \ h\ \ \sigma \\ \Longrightarrow & \mathit{ReturnInt}\ k\ \ \mathit{as}\ \ \mathit{rs}\ \ \mathit{us}\ \ h\ \ \sigma\end{array}}
$$

A similar rule deals with the case where a variable bound to a primitive value is entered:

$$
(10)\ \boxed{\begin{array}{ll} & \mathit{Eval}\ (f\ \{\})\ \rho[f \mapsto \mathit{Int}\ k]\ \ \mathit{as}\ \ \mathit{rs}\ \ \mathit{us}\ \ h\ \ \sigma \\ \Longrightarrow & \mathit{ReturnInt}\ k \qquad\qquad\qquad \mathit{as}\ \ \mathit{rs}\ \ \mathit{us}\ \ h\ \ \sigma\end{array}}
$$

Next come the rules for the *ReturnInt* state, which look for a continuation on the return stack. First, the case where there is an alternative which matches the literal:

$$
(11)\ \boxed{\begin{array}{ll} & \mathit{ReturnInt}\ k\ \ \mathit{as}\ \ (\ldots\texttt{;}\ k\texttt{->}e\texttt{;}\ \ldots,\ \rho) : \mathit{rs}\ \ \mathit{us}\ \ h\ \ \sigma \\ \Longrightarrow & \mathit{Eval}\ e\ \rho \qquad \mathit{as} \qquad\qquad\qquad\qquad \mathit{rs}\ \ \mathit{us}\ \ h\ \ \sigma\end{array}}
$$

Next, the cases where the default alternative is taken:

$$
(12)\quad
\begin{array}{|lll|}
\hline
 & \mathit{ReturnInt}\ k \qquad \mathit{as} \left(\begin{array}{l} k_1\ \texttt{->}\ e_1\,\texttt{;} \\ \ldots\,\texttt{;} \\ k_n\ \texttt{->}\ e_n\,\texttt{;} \\ x\ \texttt{->}\ e \end{array},\ \rho\right) : \mathit{rs}\ \ \mathit{us}\ \ h\ \ \sigma & \\
\text{such that} & k \neq k_i \qquad (1 \leq i \leq n) & \\
\Longrightarrow & \mathit{Eval}\ e\ \rho[x \mapsto \mathit{Int}\ k]\ \ \mathit{as} \qquad \mathit{rs}\ \ \mathit{us}\ \ h\ \ \sigma & \\
\hline
\end{array}
$$

$$
(13)\quad
\begin{array}{|lll|}
\hline
 & \mathit{ReturnInt}\ k\ \ \mathit{as} \left(\begin{array}{l} k_1\ \texttt{->}\ e_1\,\texttt{;} \\ \ldots\,\texttt{;} \\ k_n\ \texttt{->}\ e_n\,\texttt{;} \\ \texttt{default}\ \texttt{->}\ e \end{array},\ \rho\right) : \mathit{rs}\ \ \mathit{us}\ \ h\ \ \sigma & \\
\text{such that} & k \neq k_i \qquad (1 \leq i \leq n) & \\
\Longrightarrow & \mathit{Eval}\ e\ \rho \qquad \mathit{as} \qquad \mathit{rs}\ \ \mathit{us}\ \ h\ \ \sigma & \\
\hline
\end{array}
$$

Finally, we need a family of rules for built-in arithmetic operations which, for each binary built-in operation $\oplus$ have the form:

$$
(14)\quad
\begin{array}{|ll|}
\hline
 & \mathit{Eval}\ (\oplus\ \{x_1, x_2\})\ \rho[x_1 \mapsto \mathit{Int}\ i_1,\ x_2 \mapsto \mathit{Int}\ i_2]\ \ \mathit{as}\ \ \mathit{rs}\ \ \mathit{us}\ \ h\ \ \sigma \\
\Longrightarrow & \mathit{ReturnInt}\ (i_1 \oplus i_2) \qquad \mathit{as}\ \ \mathit{rs}\ \ \mathit{us}\ \ h\ \ \sigma \\
\hline
\end{array}
$$

5.6 Updating

In this section we cover the updating technology necessary for a graph reduction machine. Updates happen in two stages:

1. When an updatable closure is entered, it pushes an *update frame* onto the update stack, and makes the argument and return stacks empty. An update frame is a triple (as_u, rs_u, a_u), consisting of:
   - as_u, the previous argument stack;
   - rs_u, the previous return stack;
   - a_u, a pointer to the closure being entered, and which should later be updated.
2. When evaluation of the closure is complete an update is triggered. This can happen in one of two ways:
   - If the value of the closure is a data constructor or literal, an attempt will be made to pop a continuation from the return stack, which will fail because the return stack is empty. This failure triggers an update. (In the real implementation we can avoid making the test by merging the return and update stacks, and making the update into a special sort of continuation — Section 10.1.)
   - If the value of the closure is a function, the function will attempt to bind arguments which are not present on the argument stack (because they were squirreled away in the update frame). This failure to find enough arguments triggers an update.

These situations are made precise in the following rules. First, we need to add an extra rule which applies when entering an updatable closure (that is, one whose update flag is `u`). The rule is similar to the usual closure-entry rule (Rule 2):

$$
(15)\quad
\begin{array}{|lll|}
\hline
 & \mathit{Enter}\ a\ \ \mathit{as}\ \ \mathit{rs} & \mathit{us}\ \ h[a \mapsto (\mathit{vs}\ \texttt{\textbackslash u}\ \texttt{\{\}}\ \texttt{->}\ e)\ \mathit{ws}_f]\ \ \sigma \\
\Longrightarrow & \mathit{Eval}\ e\ \rho\ \{\}\ \{\}\ (\mathit{as}, \mathit{rs}, a) : \mathit{us}\ \ h & \sigma \\
\multicolumn{3}{|l|}{\text{where } \rho = [\mathit{vs} \mapsto \mathit{ws}_f]} \\
\hline
\end{array}
$$

The difference is that the argument stack, return stack, and closure being entered are formed into an update frame, which is pushed onto the update stack. (Naturally, the real implementation manipulates pointers rather than copying entire stacks — Section 10.3.) Since closures with a non-empty argument list are never updatable (Section 4.2), we only deal with this case in the rule given.

Next, we need new rules for constructors which see an empty return stack. When this happens, they update the closure pointed to by the update frame, restore the argument and return stacks from the update frame, and try again. It may be that the restored return stack contains the continuation, but it too may be empty, in which case a second update is performed, and so on until the continuation is exposed.

$$
(16)\quad
\begin{array}{|lll|}
\hline
 & \mathit{ReturnCon}\ c\ \mathit{ws}\ \ \{\}\ \ \{\}\ \ (\mathit{as}_u, \mathit{rs}_u, a_u) : \mathit{us}\ \ h & \sigma \\
\Longrightarrow & \mathit{ReturnCon}\ c\ \mathit{ws}\ \ \mathit{as}_u\ \ \mathit{rs}_u\ \ \ \ \mathit{us}\ \ h_u & \sigma \\
\text{where} & \mathit{vs} \text{ is a sequence of arbitrary distinct variables} & \\
 & \mathit{length}(\mathit{vs}) = \mathit{length}(\mathit{ws}) & \\
 & h_u = h[a_u \mapsto (\mathit{vs}\ \texttt{\textbackslash n}\ \{\}\ \texttt{->}\ c\ \mathit{vs})\ \mathit{ws}] & \\
\hline
\end{array}
$$

The closure to be updated (address a_u) is just updated with a standard-constructor closure. Only a rule for *ReturnCon* need be given. It is not possible for the *ReturnInt* state to see an empty return stack, because that would imply that a closure should be updated with a primitive value; but no closure has a primitive type (Section 4.7).

Finally, we need a rule to handle the case where there are not enough arguments on the stack to be bound by a lambda abstraction, which triggers an update. The relevant rule is:

$$
(17)\quad
\begin{array}{|lll|}
\hline
 & \mathit{Enter}\ a\ \ \ \ \mathit{as}\ \ \{\}\ \ (\mathit{as}_u, \mathit{rs}_u, a_u) : \mathit{us}\ \ h & \sigma \\
\text{such that} & h\ a = (\mathit{vs}\ \texttt{\textbackslash n}\ \mathit{xs}\ \texttt{->}\ e)\ \mathit{ws}_f & \\
 & \mathit{length}(\mathit{as}) < \mathit{length}(\mathit{xs}) & \\
\Longrightarrow & \mathit{Enter}\ a\ \ \mathit{as} \mathbin{+\!\!+} \mathit{as}_u\ \ \mathit{rs}_u\ \ \ \ \mathit{us}\ \ h_u & \sigma \\
\text{where} & \mathit{xs}_1 \mathbin{+\!\!+} \mathit{xs}_2 = \mathit{xs} & \\
 & \mathit{length}(\mathit{xs}_1) = \mathit{length}(\mathit{as}) & \\
 & h_u = h[a_u \mapsto ((\mathit{vs} \mathbin{+\!\!+} \mathit{xs}_1)\ \texttt{\textbackslash n}\ \mathit{xs}_2\ \texttt{->}\ e)\ (\mathit{ws}_f \mathbin{+\!\!+} \mathit{as})] & \\
\hline
\end{array}
$$

(The rule will only apply if the number of arguments $\#xs$ is greater than zero, so the closure being entered will be non-updatable; hence the `\n` in the first line of the rule.) The closure to be updated (address a_u) has as its value the value of the closure being entered (address a) applied to the arguments on the stack as. It is therefore updated with a closure whose code is $((vs \mathbin{+\!\!+} xs_1)$ `\n` xs_2 `->` $e)$; the body e is the same as that of a, but it has more free variables (xs_1 as well as vs) and fewer arguments (xs_2 instead of xs). After the update the *Enter* is retried.

This concludes the basic rules for updating. However, one of the constraints in a real implementation is that it cannot manufacture compiled code "on the fly", so we need to be careful about the code part of closures which are created by updating. The code required for constructors, (vs `\n` $\{\}$ `->` $c\ vs$) is OK, because we can precompile it for each constructor c.

The code for partial applications, $((vs \mathbin{+\!\!+} xs_1)$ `\n` xs_2 `->` $e)$, is more tiresome, since it suggests that we need to precompile the entire body e of every function for every possible partial application. An alternative rule for partial-application updates avoids this problem:

$$
\begin{array}{lll}
 & \begin{array}{lllll} \textit{Enter}\ a & as & \{\} & (as_u, rs_u, a_u) : us & h \quad \sigma \end{array} \\[1ex]
\text{such that} & h\ a = (vs\ \texttt{\textbackslash n}\ xs\ \texttt{->}\ e)\ ws_f \\
 & \textit{length}(as) < \textit{length}(xs) \\[1ex]
(17a)\ \Longrightarrow & \textit{Enter}\ a\ \ as \mathbin{+\!\!+} as_u\ \ rs_u \qquad\qquad us\ \ h_u\ \ \sigma \\[1ex]
\text{where} & xs_1 \mathbin{+\!\!+} xs_2 = xs \\
 & \textit{length}(xs_1) = \textit{length}(as) \\
 & f \text{ is an arbitrary variable} \\
 & h_u = h[a_u \mapsto ((f : xs_1)\ \texttt{\textbackslash n}\ \{\}\ \texttt{->}\ f\ xs_1)\ (a : as)]
\end{array}
$$

Here the closure being entered, a, is used in the new closure. The new code required, namely $((f : xs_1)$ `\n` $\{\}$ `->` $f\ xs_1)$, can be shared between all partial applications to the same number of arguments. All that is required is a family of such code-blocks, one for each possible number of arguments.

Part III: Mapping the abstract machine to stock hardware

We have now completed the abstract description of the Spineless Tagless G-machine. Whilst it has some interesting features, its real justification is that it maps very nicely onto stock hardware, with a rich set of design alternatives, some of which we have already indicated. In the rest of the paper we describe the mapping in detail.

6 Target language

Our goal is, of course, to generate good native code for a variety of stock architectures. One approach to this is to write individual code generators for each architecture, and this is likely to give the best results in the end. Unfortunately, to compete with more mature imperative languages, whose code generators have evolved and improved over many years, we would have to do a comparably good job of code generation, which is a lot of work.

Motivated by this concern, we generate code in the C language as our primary target, rather than generating native code direct. In this way we gain instant portability, because C is implemented on a wide variety of architectures, and we benefit directly from improvements in C code generation. This approach, of using C as a "high-level assembler" has gained popularity recently (Bartlett [1989]; Miranda [1991]).[4] In particular, the work of Tarditi *et al* on compiling SML to C, developed independently and concurrently with ours, addresses essentially the same problems (Tarditi, Acharya & Lee [1992]).

Rather than generating C directly, we go via an internal datatype called "Abstract C". This allows the following spectrum of alternatives for the final code generation, with increasing efficiency and decreasing portability:

- We can generate ANSI-standard C, which should be widely portable.
- We can generate C which exploits various non-standard extensions to C supported by the Gnu C compiler (Stallman [1992]).
- We can generate native machine code directly.

So far we have concentrated only on the first two alternatives.

Compiling via C is very attractive for portability reasons, but like all good things, it does not come for free. In the rest of this section we describe a few tricks which substantially improve the code we can generate using this route, usually by exploiting non-standard extensions to C provided by Gnu C.

[4] Here, "Miranda" is not the trade mark. It is the last name of a researcher at Queen Mary and Westfield College, London.

6.1 Mapping the STG machine to C

At first it appears sensible to try to map functions from the original functional program onto C functions, but we soon abandoned this approach. The mis-match between C and a non-strict higher-order functional language is too great.

Instead, the argument stacks and control stack are mapped onto explicit C arrays, bypassing the usual C parameter-passing mechanism. All "registers", such as the stack pointers, heap pointer, heap limit, and other registers introduced later, are held in global variables.

This approach results in a great deal of global-variable manipulation. The overheads can be reduced without losing portability, by caching such globals in (register-allocated) local variables during the execution of a single code-block, based on a simple usage analysis (Tarditi, Acharya & Lee [1992]). At the expense of portability, the overheads can be eliminated entirely, by telling the C compiler to keep particular globals in specified registers permanently, a (highly non-standard, architecture-specific) facility provided by the Gnu C compiler.

6.2 Compiling jumps

The main difficulty with generating C concerns labels. We use the term *code label* (or just label) to mean an identifier for a code sequence. The important characteristics of a code label are that:

- It can be used to name an arbitrary block of code.
- It can be manipulated; for example, it can be pushed onto a stack, stored in a closure, or placed in a table.
- It can be used as the destination of a jump.

We usually think of labels as being represented by code addresses. The trouble is that C has nothing which directly corresponds to code labels. There are two ways out of this dilemma, which we outline in the following subsections.

6.2.1 Using a giant `switch`

The first solution is to map labels onto integer tags, and embed the entire program in a loop with the following form:

```
int cont = 1;
while (TRUE) do
  switch (cont) {
        1:      ...code for label 1...;
        2:      ...code for label 2...;
        ... and so on ...
  }
```

Now a jump can be implemented by assigning to `cont` followed by a `break` statement. The `switch` statement will then re-execute with the new label.

The shortcomings of the technique are clear. Firstly, a layer of indirection has been imposed, because labels are not implemented directly as code pointers.

Secondly, and more seriously, separate compilation is made much more difficult. The C code for the entire program, including the run-time system, has to be gathered together into a single giant C procedure and then compiled. Not only does this stress the C compiler quite substantially, and impose heavy recompilation costs on even local changes, but it also means that a special linker has to be written to paste together the C code generated from each separately-compiled source-language module.

6.2.2 Using a tiny interpreter

Because of these problems we use an alternative method, based on a nice trick. The idea is to compile each labelled block of code to a parameter-less C function whose name is the required label. Now, C does treat functions as storable values, representing each by a pointer to its code. The only problem is how to jump to such a code block. The only mechanism C provides is to call the function, but then every jump would make C's return stack grow by one more word, causing certain stack overflow.

A C compiler which implemented a tail call as a jump would not suffer from this problem, but it would hardly be a portable solution to *require* such an optimisation for correct operation. Furthermore, C is complicated enough to make the tail-call optimisation quite hard to get right (in the presence of variadic functions, for example) and no C compiler known to us does so.

So here is the trick: each parameterless function, representing a code block, *returns* the code pointer to which it would like to jump, rather than *calling* it. The execution of the entire program is controlled by the following one-line "interpreter":

```
while (TRUE) { cont = (*cont)(); }
```

That is, `cont` is the address of the code block (that is, C function) to be executed next. The function to which it points is called, and returns the address of the next one, and so on. The loop is finally broken by a long-jump, though one could equally well test `cont` for a particular value instead, for a fairly minor cost.

Here, for example, is a code block which jumps to a label found on top of the return stack:

```
CodeLabel f() {
        CodeLabel lbl = *RetSp--;
        return( lbl );
}
```

The result is a fully-portable implementation which supports separate compilation in the usual way, with a standard linker. Labels are represented directly by code addresses.

Temporary variables, used within a single code block, are declared as local variables of the C function generated for the code block. Their scope is thereby limited, so that a good C compiler will put them in registers where possible.

It turns out that this idea is actually very old, and that we only reinvented it. Like several other clever ideas, Steele seems to have been its inventor; he called it the "UUO handler" in his Rabbit compiler for Scheme (Steele [1978]). The same idea is used by Tarditi, Acharya & Lee [1992], who use C as a target for their SML compiler.

6.3 Optimising the tiny interpreter

In the portable tiny interpreter described above, a "jump" has the following overheads:

- the epilogue generated by the C compiler for the current C function, concluding with a return instruction, which pops the return address to return to the "interpreter";
- a jump to implement the interpreter's loop;
- a subroutine call instruction, which pushes the interpreter's return address;
- the prologue generated by the C compiler for the new C function.

At the expense of portability, we can make some architecture- and compiler-specific optimisations to this jump sequence:

Eliminating register saves. For architectures with a fixed register set, most C compilers implement a callee-saves convention for all registers except a small number of work registers. There is a save sequence at the start of each function and a restore sequence at the end.

Gnu C provides a compiler flag which makes the compiler use a caller-saves convention. In conjunction with the direct-jump optimisation described below this eliminates all register save instructions.

Eliminating the frame pointer. Most C compilers generate instructions at the beginning and end of functions to set up a frame pointer register. This is redundant, because the compiler can always figure out the offsets of local variables from the stack pointer itself, but it is vital for debuggers.

Gnu C provides a compiler flag to suppress frame-pointer manipulation, at the expense of confusing the debugger.

Generating direct jumps. Instead of generating `return( lbl )` we actually generate `JUMP( lbl )`, where `JUMP` is a macro. For a portable implementation, `JUMP` expands to a `return` statement, but the implementation can be made faster by making `JUMP` expand to an in-line assembly-code instruction which really does take a jump. (Most C compilers provide an assembly-language trapdoor, which we exploit here.) Using in-line assembly code in this way has its pitfalls, especially if we simultaneously try to use local variables. Miranda gives details of the tricky things one has to do (Miranda [1991]).

6.4 Debugging

The use of a tiny interpreter turned out to have a very useful property which we had not anticipated: it is a tremendous debugging aid.

The STG machine frequently takes an indirect jump, to the code pointed to by a closure. If a bug has caused a closure to be corrupted, this indirect jump usually causes a segmentation fault or illegal instruction. The difficulty is that there usually no way of backing up to the code which performed the jump, which is the first step in identifying the source of the error.

Using the (unoptimised) tiny interpreter provides an easy solution, because it can easily record a trail of the most recent few jumps. Since every jump passes through the tiny interpreter, it faithfully records the address of the code block containing the fatal jump.

Furthermore, it is easy to add to the tiny interpreter's loop a call to a hygiene-checking routine, which checks that the machine state looks plausible. While it slows down the program considerably, we have found this hygiene-checking an invaluable aid for trapping the point at which the machine state becomes corrupt, rather than the point at which the corruption causes a crash, which is often much much later.

It is hard to overstate the usefulness of this trick, especially since it has no impact at all on the compiler and the code it generates. Only people who have spent all night trying to find the cause of the heap corruption which subsequently led to a system crash can truly appreciate it!

7 The heap

The heap is a collection of *closures* of variable size, each identified by a unique *address*. We use the term *pointer* to refer to the address of a closure.

7.1 How closures are represented

Each closure occupies a contiguous sequence of machine words, which is always laid out as shown in Figure 4.

The first word of a closure is called its *info pointer*, and points to its *info table*. Following the info pointer is a block of words each of which contains a pointer, followed by a block of words containing no pointers. (The distinction between the two is that the garbage collector must follow the former but not the latter.) There is a single, statically-allocated, info table associated with each *bind* in the program text (Figure 3). Each dynamic instance of this binding is a heap-allocated closure whose info-pointer refers to its static info table (Rule 3).

The info table contains a number of fields which will be described later, but the most important is the first field, which contains the label of the closure's *standard-entry code*. The operation of *entering* a closure is performed by:

- loading the address of the closure into the `Node` register, and

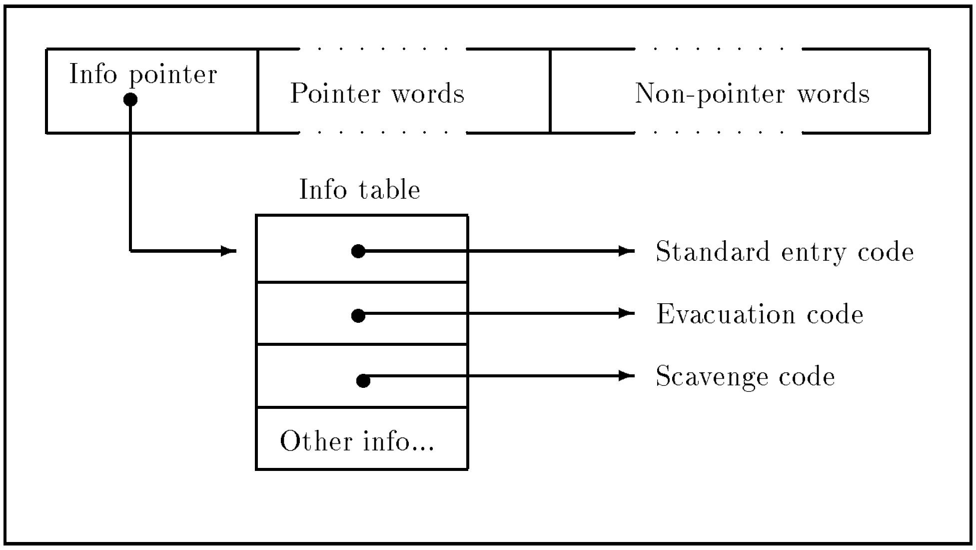


Figure 4: The layout of a closure

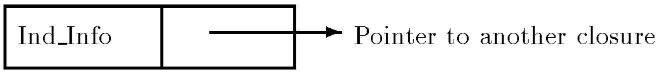
- jumping to the standard-entry code for the closure, whose label is usually fetched from the info table by indirecting from `Node`.

The standard-entry code can access the various fields of the closure by indexing from the `Node` register. The rest of the info table contains:

- Enough information to enable the garbage-collector to do its job. In fact we implement this information as two code labels, which are described further in Section 7.3.
- Debugging information for inspection by a debugger or trace generator.
- For our parallel implementation, enough information to enable the closure to be flushed into global memory. This, too, is actually implemented as a code label.

It is usual for heap-allocated objects to contain layout information, to specify their size and which of their fields contain pointers. In contrast, our closures do not contain any such information. Rather, as we shall see, size and layout information is encoded in the info table.

Indirection closures are generated by update operations, and they have a particularly efficient representation:



The standard-entry code for `Ind_Info` consists of only two instructions: one to load the indirection pointer from the closure into `Node` and a second to enter the new closure.

In retrospect, this representation is quite similar to that chosen by the Chalmers group for their G-machine implementation (Johnsson [1987]). In their system, every heap cell has a one-word "tag" which points to a table of entry points for the various operations that could be performed on the cell. Our system differs from theirs in two respects. First, and most important, rather than having a fixed collection of "tags", we generate a new info table for each *bind* in the program text, together with its associated code. This essentially eliminates the "interpretive unwind" used by the G-machine. Second, the operation of entering a closure involves an indirection to find the code label to jump to; this indirection can be avoided when generating native code directly, as Section 7.6 discusses.

7.2 Allocation

The closures for top-level globals are allocated statically at fixed addresses; we call them *static closures*. A static closure is not necessarily immutable, however, because it may be a thunk which is updated during execution. (The alert reader will spot that this policy gives rise to a garbage-collection problem, which we return to in Section 10.8.)

All other closures are allocated dynamically from the heap. As is now well understood, for good performance it is essential to allocate from a contiguous block of free space, rather than from a free list (Appel [1987]). Free space is delimited by two special registers: the `Hp` register points to one end of it, while the `HLimit` register points to the other. Allocation is done on a basic-block basis, so that only one free-space exhaustion check is made for each basic block.

7.3 Two-space garbage collection

Garbage collection is performed by a two-space stop-and-copy collector (Baker [1978]). Available memory is divided into two semi-spaces. When garbage collection is initiated, all live closures are copied from one semi-space to (one end of) the other.

This copying process involves two basic operations on closures:

- Each live closure must be *evacuated* from from-space to to-space.
- As to-space is scanned linearly, each closure must be *scavenged*; that is, each closure to which it points must be evacuated, unless it has already been evacuated, and the new to-space pointer substituted for the old from-space pointer.

The unusual feature of our system is that these two operations, evacuation and scavenging, are implemented by code pointed to from the info table of each closure. These code sequences "know" the exact structure of the closure, and therefore can operate without interpretive loops, and without any further layout information.

The evacuation code, which is called as a C function, does the following:

- It copies the closure into to-space.
- It overwrites the closure in from-space with a *forwarding pointer*, which points to the newly-allocated copy of the closure in to-space.
- It returns the new to-space address of the closure to the caller.

The scavenging code of a closure, also called as a C function, does the following. For each pointer in the closure,

- it calls the evacuation code for the closure to which it points;
- it replaces the pointer in the original closure with the to-space pointer returned from this evacuation call.

The scavenging code knows which of the argument fields contain closure addresses (and hence must be evacuated), and which are not (and hence must *not* be evacuated).

A C function does the once-per-collection work of switching spaces and accumulating statistical information, but almost all the work of garbage-collection is carried out by the evacuate and scavenge routines of the closures in the heap. As in the case of the standard-entry code of a closure, the info-table dispatch mechanism for evacuation and scavenging provides the opportunity to deal with several special cases "for free" (that is, without any further tests):

Forwarding pointers. A forwarding pointer handles the situation where a second attempt is made to evacuate the closure; an attempt to evacuate a closure which has been overwritten with a forwarding pointer simply returns the to-space address found in the forwarding pointer. There is a nice optimisation available here. Most systems distinguish a forwarding pointer by some sort of tag bit, which has to be tested just before evacuating. Instead, we make a forwarding pointer look just like any other closure: it has an info pointer and one field which points to the to-space copy. The info table for a forwarding pointer has rather simple "evacuation" code, which just returns the to-space address found in the forwarding pointer! So to evacuate a closure one simply jumps to its evacuation code, regardless of whether the closure is now a forwarding pointer or not. No forwarding-pointer test is performed.

All heap-allocated closures are at least two words long, in order to leave enough space for a forwarding pointer.

Indirections. All indirections can easily be removed during garbage collection, by another nice trick. All that is required is that the evacuation routine of an indirection jumps to the evacuation routine of the closure to which the indirection points! (The use of "jumps to" rather than "calls" is deliberate — this is a tail call!) Since indirections are thereby never moved into to-space, they don't have a scavenging routine.

Static closures. Some closures, notably those for global closures (Section 5.1), are allocated at fixed, static locations. These closures must not be moved by the garbage collector. This is easily arranged by making their evacuation code return immediately without moving the closure.

Constructor closures can exist in both static and dynamic space (Section 4.4), so in fact we need two info tables for each constructor, one for each of these cases. (The standard-entry code for the constructor can still be shared, of course.)

Small integers. A fixed-precision integer (of type `Int`) is represented by the `MkInt` constructor applied to the primitive integer value (Section 4.7). This in turn is represented by a two-word closure consisting of the `MkInt` info pointer and the primitive integer value. The evacuation code for `MkInt` sees if the value of the integer lies in a pre-determined range and, if so, uses the integer to index a table of statically-allocated `Int` closures, returning the address of this static closure. The effect is that all small integers are "commoned up" by the garbage collector, and made to point to one of a fixed collection of small-integer closures.

If the integer is not in the range of the table, the closure is evacuated to to-space as usual. There is an easy refinement: give the new copy a different info pointer which won't perform the test again next time (because it will certainly fail again).

The small-integer check could of course be made at the time an `Int` is allocated, but that means generating extra code in lots of places, whereas doing it in the garbage collector requires just one chunk of extra code. The same optimisation applies to `Char` closures and all other constructors isomorphic to `Int` or `Char`.

7.4 Other garbage collector variants

Two-space garbage collection works well until the residency of the program approaches half the real memory available, at which point the virtual memory system begins to thrash. We have implemented a dual-mode collector, which switches dynamically between a single-space compacting collector and a two-space collector to try to minimise paging, with encouraging early results (Sansom [1991]). We are developing a further extension to a generational collector, based on Appel's simple two-generation scheme (Appel [1989]).

7.5 Trading code size for speed

The info-table dispatch mechanism outlined above allows some interesting space-time tradeoffs to be made.

So far we have assumed that each kind of closure has its own evacuation and scavenging code, which "knows about" its size and layout. This requires new evacuation and scavenging routines to be compiled for each closure in the program. But since the garbage collection routines for a closure depend only on its *structure*, it is often possible to share them. For example, all closures which consist of exactly one pointer field (apart from the info pointer) can share the same evacuation and scavenging routines. Indeed our runtime system contains standard garbage-collection routines for a number of common layouts.

What if a closure must be constructed which does not match one of these standard layouts? It is possible to compile special garbage-collection code for it, but actually we adopt a compromise position which allows us to provide *all* evacuation and scavenging routines as part of the runtime system. Instead of generating code for garbage-collection routines for a "non-standard" closure, we provide "generic" evacuation and scavenging routines in the runtime system. These routines look in the closure's info table to find certain layout information, namely the number of pointer words and non-pointer words in the closure. (This is contained in the "Other info" field of Figure 4.) They then each use a loop to do their work, instead of having the loop unrolled as the special-purpose routines do. Notice that the layout information is stored in the (static) info table, so there is no extra cost in allocating the closure. The only extra execution cost is in executing the loops in the garbage collection routines.

It is for the benefit of these "generic" routines that closures are laid out with pointers preceding non-pointers. This convention means that only two numbers are required to encode the layout information. It also makes it more likely that a closure's layout will "fit" a standard layout directly supported by the runtime system. For example, all closures with two words of non-pointers and two of pointers can use the same routines; if the layout convention was more liberal, there would be a number of different possible layouts of such closures. The convention carries no runtime cost, of course.

7.6 The standard-entry code for a closure

There is one particular place where we have found that the use of C prevents an obvious code improvement. The info pointer of a closure points to a table containing a number of code labels. One of these is used much more than the others, namely the one used when the closure is entered.

It would be better to arrange that the info pointer pointed *directly* to this code, placing the rest of the info table *just before* the code. Then, entering the closure takes one fewer indirections, but the other info-table entries are still available by using negative offset from the info pointer.

This is usually quite easy to arrange when generating native code, but even the Gnu C compiler doesn't allow the programmer to specify that an array (the info table) must immediately precede the first word of the code for a function!

We abstract away from this issue by using a C macro `ENTER( c )`, where `c` contains the address of the closure to be entered. The usual definition of `ENTER` is:

```
#define ENTER(c)  JUMP(**c)
```

8 Stacks

The abstract machine contains three stacks:

- The argument stack, which contains a mixture of closure addresses and primitive values.
- The return stack, which contains continuations for `case` expressions.
- The update stack, which contains update frames.

The question is: how are these stacks to be mapped onto a concrete machine?

8.1 One stack?

The three stacks all operate in synchrony, so it would be possible to represent them all by a single concrete stack. The major reason we choose not to do so is to avoid confusing the garbage collector. The garbage collector must use all the pointers in the stack as a source of roots, and must update them to point to the new locations of the closures. Thus, it needs to know which stack locations are closure addresses and which are code addresses or primitive values.

There are a couple of ways around this problem, while retaining a single stack. One possibility is to distinguish pointers from non-pointers with a tag bit (usually the least-significant bit). This is a nuisance, because it makes arithmetic slower, and because it makes standard 32-bit floating-point numbers impossible. It is also rather against the spirit of our implementation, where all type information is static, requiring no runtime testing.

Another possibility, described in an earlier version of the Spineless Tagless G-machine (Peyton Jones & Salkild [1989]) uses static bit-masks associated with the code pointed to by return addresses on the stack to give the stack layout. This works fine, but since then we have introduced the idea of fully-fledged unboxed values, which fatally wounds this technique. Consider, for example, the program

```
pick b f g = if b then f else g
h b n = pick b (+# 1#) (-# 1#) n
```

Here, when `pick` is called, the four arguments `b`, `(+# 1#)`, `(-# 1#)`, and `n` will be on the argument stack. The last of these will presumably be primitive, since later `(+# 1#)` or `(-# 1#)` will be applied to them. Now here is the point: *if garbage collection is initiated during the evaluation of* `b`*, there is no context information available to tell that the bottom argument on the stack is primitive.*

To conclude, using a single stack seems to require runtime tagging; previous ways of avoiding this cannot cope with fully-fledged unboxed values.

8.2 Two stacks

The obvious solution, which we use, is to provide two concrete stacks, the A-stack for pointers and the B-stack for non-pointers. This was the solution adopted by the G-machine, where the non-pointer stack was called the V-stack, and a number of subsequent systems. The nomenclature we use is taken from the ABC machine (Koopman [1990]), where "A" stands for "argument" and "B" for basic value. However, our argument stack is split between the A and B stacks, and the B stack contains other things besides non-pointer arguments, as will become apparent. The detailed mapping of each of the abstract stacks to these two concrete stacks is given in subsequent sections.

The stack pointers are held in special registers `SpA` and `SpB`. Like other twin-stack implementations, we make the two stacks grow towards each other, to avoid the risk that one will overflow while the other has plenty of space left; in this paper, the A-stack grows towards lower addresses. In our sequential implementation, this stack space is allocated in a fixed-size area, separate from the heap.

9 Compiling the STG language to C

We are now at last ready to discuss the code which is generated for each of the constructs in the STG language. This section is rather long and detailed. We make no apology for this because, as remarked earlier, an abstract machine can only be considered a success if it maps well onto concrete architectures, with plenty of opportunities for optimisations.

We begin with an overview of the code generation process for an arbitrary STG expression, by considering the various syntactic forms an expression can take (Figure 3):

- *Calls to non-built-in functions* (Section 9.2). The expression $f\ a_1, \ldots, a_n$ is compiled to a sequence of statements which pushes the arguments $a_1, \ldots, a_n$ onto the appropriate stacks, adjusts the A and B stack pointers to their final values, and enters the function f. As we discuss later, this "enter" may take the form of entering the closure bound to f via its info table, or of jumping direct to the appropriate code for f.

- `let(rec)` *expressions* (Section 9.3). The `let` expression

  `let` x_1 `=` lf_1 `; ... ;` x_n `=` lf_n `in` e

  is compiled to a sequence of statements which allocates a closure in the heap for each lambda-form$lf_1, \ldots, lf_n$, followed by the code for e. `letrec` expressions are treated in the same way, the only difference being that the closures allocated thereby may be cyclic.

  If the lambda-formlf_i is not a standard constructor, the code generator also produces:

  - A separate block of code labelled x_i`_entry`, obtained by compiling the body of lf_i. (See Section 9.2.1 for why it may be useful to give this code an extra entry point.)
  - The declaration for a statically-initialised array x_i`_info`, which is the info table for x_i. The first element of the info table is (the label of) the standard-entry code x_i`_entry`.

  Both of these declarations are hoisted out to the top level, rather than appearing embedded in the middle of the code for the `let` expression. In our code generator this flattening process is performed *after* code generation, the intermediate data type (Abstract C) permitting nested declarations.

  If the lambda-form*is* a standard constructor, the shared info table for the appropriate constructor can be used, and there is no need to generate x_i`_info` and x_i`_entry`.

- *Literals and calls of built-in operators* (Section 9.5). A primitive literal k is compiled to statements which load k into a register (exactly which register depends on k's type), adjusts the A and B stack pointers to their final values, and returns to the address on top of the B stack. A call to a built-in operation works in the same way except that the operation is performed first.

  This makes it sound as if every built-in operation is associated with a return, but an easy optimisation allows sequences of built-in operations to be compiled (Section 9.5).

- `case` *expressions* (Section 9.4). The primitive `case` expression

  `case` e `of` *palts*

  is compiled to code which saves any volatile variables used by *palts* on the stacks, and pushes a return address on the B stack, followed by the code for e. An arbitrary but unique label is invented for the return address, which is used to label a separate block of code compiled from *palts*.

  The code compiled for *palts* performs case analysis on the value returned (if there are any non-default alternatives) followed by the code for each alternative expression. Like the code for lambda-forms, this entire code block is hoisted to the top level.

  The code for algebraic `case` expressions is similar, except that (the address of) a return vector is pushed instead of a return address (Section 9.4.3).

- *Top level bindings* (Section 9.1). The top-level bindings are treated a little differently to nested ones. Each declaration g_i = lf_i is compiled to the declaration of a statically-initialised array g_i`_closure`, which represents the static closure for g_i. An info table g_i`_info` and standard-entry code-block g_i`_entry` are also produced just as for nested bindings.

- *Standard constructors* (Section 9.4.2). As already mentioned, no code is generated for lambda-forms which are standard constructors, the shared info table and code for the constructor being used instead. It is therefore necessary to generate this info table and entry code for each constructor declared in the module.

To make these ideas concrete, Figure 5 gives the code compiled for `map`, whose STG code is as follows (see Section 4.1):

```
map = {} \n {f,xs} ->
            case xs of
              Nil {}      -> Nil {}
              Cons {y,ys} -> let fy = {f,y} \u {} -> f {y}
                                 mfy = {f,ys} \u {} -> map {f,ys}
                             in Cons {fy,mfy}
```

This code is written assuming that lists use a vectored return convention, and `Cons` returns its arguments in registers, matters which are explained more fully in Section 9.4.

The rest of this section explores code generation in more detail. Each subsection corresponds to the similarly-numbered subsection of Section 5 which gives !the operational semantics of the STG language.

9.1 The initial state

The machine is initialised to evaluate the global `main`, with empty argument, return and update stacks (Section 5.1). The abstract machine's initial heap is not empty, but rather contains a closure for each globally-defined variable. We implement this by allocating a static closure for each such variable (Section 7.2). Each of these closures can be referred to directly by its C label, thus effectively using the linker to implement the global environment σ.

```
StgWord map_closure[] = {map_info};
StgWord map_info[] = {map_entry, ...rest of info table...}
map_entry() {
    ...argument satisfaction check...
    JUMP( map_direct );
}
map_direct() {
    ...stack overflow check...
    SpB[1] = ret_vec1; SpB = SpB+1;     /* Push return vector */
    Node = SpA[1];    ENTER( Node );    /* Enter xs */
}

StgWord ret_vec1[] = {ret_nil1, upd_nil, ret_cons1, upd_cons};
ret_nil1() {
    SpA = SpA+2;                        /* Pop args */
    SpB = SpB-1; RetVecReg = SpB[1];    /* Grab return vector */
    JUMP( RetVecReg[0] );
}
ret_cons1() { /* Head and tail in regs RetData1 and RetData2 */
    /* Allocate fy and mfy  */
    Hp = Hp + 6;
    ...heap overflow check...
    Hp[-5] = fy_info; Hp[-4] = SpA[0]; Hp[-3] = RetData1;
    Hp[-2] = mfy_info; Hp[-1] = SpA[0]; Hp[0] = RetData2;

    /* Return the cons cell */
    RetData1 = &Hp[-5]; RetData2 = &Hp[-2];
    SpB = SpB-1; RetVecReg = SpB[1]; SpA = SpA+2;
    JUMP( RetVecReg[2] );
}

StgWord fy_info[] = {fy_entry, ...rest of info table...}
fy_entry() {
    ...push update frame...          /* This is an updatable thunk */
    ...stack overflow check...
    SpA[-1] = Node[2]; SpA = SpA-1;   /* Push y */
    Node = Node[1]; ENTER( Node );    /* Enter f */
}

StgWord mfy[] =  {mfy_entry, ...rest of info table...}
mfy_entry() {
    ...stack overflow check ...
    SpA[-1] = Node[2]; SpA[-2] = Node[1]; SpA = SpA-2; /* Push f,ys */
    JUMP( map_direct );
}
```

Figure 5: The code generated for `map`

9.2 Applications

The code generated for applications follows directly from Rule 1 in the operational semantics, and consists of two steps:

- push the arguments on the stack and adjust the stack pointer,
- enter the closure which represents the function.

We discuss these steps separately below. Before doing so, here is a small example. Consider the binding

```
apply3 = {} \n {f,x} -> f {x,x,x}
```

The following code is generated for the application `f {x,x,x}`. When this code is executed, a pointer to `f` is on top of the A stack, and under it is a pointer to `x`.

```
Node = SpA[0];          /* Grab f into Node register */
t = SpA[1];             /* Grab x into a local variable */
SpA[0] = t;             /* Push extra args */
SpA[-1] = t;
SpA = SpA - 1;          /* Adjust stack pointer */
ENTER( Node );          /* Enter f */
```

9.2.1 Entering a closure

What does it mean to "enter" a closure? After all, the operational semantics has quite a few rules dealing with the *Enter* state. The Spineless Tagless machine devolves responsibility for all these complications to the closure being entered, so that the code to enter a closure is simple and uniform. We establish the following very simple entry convention for closures: *when a closure is entered a particular register, the* `Node` *register, points to the closure*. The code for the closure can access its free variables by indexing directly from the `Node` register. All the "caller" has to do is to load a pointer to the closure into the `Node` register, and jump to the standard-entry code for the closure via its info pointer. (As previously discussed, this jump can involve either one or two indirections, depending on the particular representation chosen for closures and info tables — Section 7.6.)

It is possible to make some useful optimisations to this process, when entering a non-updatable closure. Many functions are defined at the top level of the program, or in standard libraries (`map`, for example). Entry to such functions can be made much more efficient than the standard entry mechanism just described:

- Such a function has no free variables, so there is no point in making `Node` point to its closure.
- The code label for the function is statically determined, so the jump can be a direct one, rather than indirecting via the info pointer.

- If the code generator knows how many arguments (if any) the closure is expecting (Section **??**), and if at least this number of arguments is being supplied by the call, then the jump can be made to a point (called the *direct-entry point*) just after the argument satisfaction check (see Section 9.3.2 below). Indeed, with a bit more cleverness, the stack and heap overflow checks can often be bypassed as well.

- The argument-passing convention at the direct-entry point can be different to the standard ones. In particular, arguments can be passed in registers.

  This should be beneficial, but perhaps less so than in a strict language, because functions frequently begin by evaluating one of their arguments, so the others have to be saved on the stack anyway. We have not yet implemented this idea.

Of these improvements, all but the first can be applied to locally-defined functions as well. For example, consider the expression

```
f = {} \n {x,y} -> let g = {x} \n {z} -> + {x,z}
                   in g {y}
```

The call to `g` can be made by pushing its argument `y` onto the stack, loading a pointer to the closure for `g` into `Node`, and then jumping directly to the appropriate code for `g`. Since the call is to a function whose definition is statically visible, the code generator can compile direct jumps, including bypassing the argument satisfaction check where appropriate.

We need to take a bit more care when entering an *updatable* closure. In this case we *must* jump to it via its info pointer, and never directly to its standard-entry code, because an update might have changed the info pointer! At first it seems that we must also always make `Node` point to the closure, since the standard-entry code for an updatable closure begins by pushing an update frame recording the address of the closure to be updated. But since the code to push the update frame is compiled individually for each closure, we can arrange for it to include the static address of the closure in the update frame, rather than `Node`.

No optimisations at all apply when entering the closure for a lambda-bound variable, as in the case of `apply3` above.

9.2.2 Pushing the arguments

"Pushing the arguments onto the stack" is not quite as simple as it sounds.

Firstly, the arguments may be a mixture of pointers and non-pointers so each must be pushed on the appropriate stack. The argument stack of the operational semantics is thereby split between the A and B stacks.

Secondly, in our implementation the environment ρ of the operational semantics is represented partly by locations in the stacks. This is quite conventional in many language implementations. It means, though, that the stacks must be cleared of the accumulated environment (or perhaps just part of it — see Section 9.4.1) before pushing the arguments to the call. Of course, we need to take a little care here: we must not overwrite a stack location which contains a value which is required for another argument position. There are several ways to solve this, the simplest being to move all the threatened live stack locations into registers (when generating C, local variables) before starting to overwrite them.

Here is an example:

```
f = {} \n {x,y} -> g {y,x}
```

On entry to `f`, `x` and `y` will be on the stack. It immediately calls `g` which requires the same arguments, but in the other order, so at least one register must be used during the stack rearrangement.

Such argument-shuffling is rather unusual. It is much more common for the same argument to appear in the same position, in which case no code need be generated at all. This is often the case for recursive functions which pass some arguments along unchanged.

9.3 `let(rec)` expressions

As mentioned earlier, `let` and `letrec` expressions always compile to code which allocates a closure in the heap for each definition, followed by code to evaluate the body of the let(rec). Each of these closures consists of an info pointer, and a field for each of its free variables.

For example, the expression

`let` f `=` fs `\`π xs `->` b
`in` e

compiles to code which allocates a closure for f, and then continues with code to evaluate e. For example, consider the definition of `compose`:

```
compose = {} \n {f,g,x} -> let gx = {g,x} \u {} -> g {x}
                           in f {gx}
```

The code for the body of `compose` runs as follows:

```
/* Allocate heap block */
Hp = Hp - 3;                     /* Allocate some heap */
if (Hp < HLimit)                 /* Heap exhaustion check */
        { ...trigger GC... };

/* Fill in closure for gx */
Hp[0] = &gx_info;                /* info pointer */
Hp[1] = SpA[1];                  /* g */
Hp[2] = SpA[2];                  /* x */

/* Call f */
Node = SpA[0];                   /* Grab f into Node */
SpA[2] = &Hp[0];                 /* Push gx */
SpA = SpA + 2;                   /* Adjust SpA */
ENTER( Node );
```

Here, `gx_info` is the statically-allocated info table for `gx`:

```
static int gx_info[] =
        {       &gx_entry,
                &scavenge_2,
                &evacuate_2,
                ...
        }
```

In this info table, `gx_entry` is the name of the C function which implements the standard-entry code for the closure `gx`. `scavenge_2` and `evacuate_2` are runtime system routines for performing garbage collection on closures containing two pointers (Section 7.3).

9.3.1 Allocation

The allocation of these closures is straightforward, and was discussed in Section 7.2.

References to dynamically-allocated closures within a single instruction sequence are made by offsetting from the heap pointer. (The code generator keeps track of the physical position of the heap pointer, so that correct offsets can be made even if it is moved by instructions within the basic block.)

Notice the use here of the term "single instruction sequence". In particular, this method of addressing cannot survive over the evaluation triggered by a `case` expression, because such an evaluation may take an unbounded amount of computation. Not only may this move the heap pointer unpredictably, but it may trigger garbage collection, which may rearrange the relative positions of the closures. In short, at the points in the operational semantics where the environment ρ is saved on the return stack, a pointer to each live closure must be saved on the pointer stack (Section 9.4.1).

9.3.2 The code for a closure

Much more interesting, of course, is the standard-entry code for the closure. This is the code which will get executed if the closure is ever entered. The standard-entry code for every closure begins with the following sequence:

Argument satisfaction check. This concerns updating, and is discussed in Section 10.2. It is only generated if there are one or more arguments.

Stack overflow check. If the execution of the closure can cause either stack to overflow, or (if the stacks are organised to grow towards each other) collide, execution is halted. (On a parallel machine, which works with many stacks, different action is taken.) This stack overflow check can "look ahead" into all the branches of any `case` expressions involved in the evaluation of the closure, taking the worst-case path as the overflow criterion. Of course, if there is no net stack growth, no check is performed.

Heap overflow check. A similar check is performed for heap overflow, if any heap is allocated. This was discussed in Section 7.2. The heap check cannot look ahead into `case` branches, because the evaluation implied by a `case` can perform an unbounded amount of computation.

Info pointer update. In the case of an updatable closure, its info pointer may now be overwritten with a "black hole" info pointer or, in a parallel system, a "queue me" info pointer. This is discussed in more detail below (Section 9.3.3).

Update frame construction. For updatable closures only, an update frame is pushed onto the update stack. This action causes a later update, which overwrites the closure with its head normal form. The implementation of updates, and the mapping of the update stack, are discussed in detail in Section 10.

Code is now generated for the body of the closure, with the free variables bound to appropriate offsets from the `Node` register, and the arguments to offsets from the appropriate stack pointers. (Like many other compilers, ours keeps track of where the stack pointers are pointing within the current activation record, so that at any moment it can generate the correct offset from the current stack pointer.)

9.3.3 Black holes

When an updatable closure is entered, its standard-entry code has the opportunity to overwrite the closure's info pointer with a standard "black hole" info pointer provided by the runtime system. Whilst this operation costs an instruction, it has two advantages:

- If the closure is ever re-entered before it is updated, the black hole entry code can report an error. This situation occurs in programs where a value depends on itself; for example

  ```
  letrec a = 1+a  in a
  ```

- If a closure is left undisturbed until it is finally updated with its head normal form, there is a serious risk of a *space leak*. For example, consider the STG definitions

  ```
  ns = {} \u {x} -> ..x..
  l = {} \u {ns} -> last {ns}
  ```

  where `..x..` produces some very long list. and `last` returns the last element of a list. If the thunk for `l` is left undisturbed until it is finally updated with the last element of the list `ns`, it will retain a pointer to the entire list, rather than consuming it incrementally. (This nice example is due to Jones [1991].) Overwriting the thunk for `l` with a black hole immediately it is entered solves this space leak, because a black hole retains no pointers.

It is also possible to obtain both these advantages in a slightly more subtle way, without performing the black-hole overwriting operation. Firstly, non-termination of the form detectable by black holes always results in stack overflow. The cause of the stack overflow can then easily be determined by noticing that there is more than one pointer on the update stack to the same closure. This can only happen if its value depends on itself. It is also possible that the error message obtainable from a post-mortem of the update stack could be rather more informative, because the entire collection of closures involved in the self-dependent loop can be identified and, since they all still have their original info tables attached, their source code location information could be shown too.

Secondly, we address the space-leak question. In the above example, the pointer to `ns` retained by `l` only matters at garbage-collection time. In almost all cases a thunk will be entered and updated between garbage collections, so that no space improvement is gained by overwriting with a black hole. What we would like to do is to *black-hole only those thunks which are under evaluation at garbage collection time*. Happily, they are exactly the thunks to which the update stack points! So we can safely omit the black-hole update on thunk entry, provided that instead we begin garbage collection by black-holing all the thunks pointed to from the update stack.

Since both these techniques rely on the update stack, they only apply to updatable thunks (update flag `u`). If a thunk is non-updatable (update flag `n`) it must still be black-holed by its standard-entry code. For this reason in our implementation we have two variants of the `n` update flag: `r` for *reentrant* (the closure may be entered many times, and should not be black-holed), and `s` for *single-entry* (the closure will be entered at most once, and should be black-holed). The `r` flag is used for manifest functions, constructors and partial applications, while the `s` flag is only used for thunks where update analysis has determined that an update is not required.

In a parallel system, the standard-entry code for an updatable thunk should overwrite the thunk with a "queue me" info pointer (Section 3.1.3). Unlike the black-holing of a sequential system, this operation cannot be postponed until garbage collection.

9.4 `case` expressions

Pattern matching, via `case` expressions, is utterly pervasive in lazy functional programs, all the more so in the Spineless Tagless G-machine because the boxing and unboxing operations of arithmetic are done using `case` expressions rather than by some *ad hoc* mechanism. One of the strengths of the Spineless Tagless G-Machine is that there is a rather rich design space for how pattern-matching can be implemented, including some rather efficient options.

`case` expressions (and only `case` expressions) cause evaluation to take place. In the operational semantics this is expressed by pushing a continuation onto the return stack, and evaluating the expression to be scrutinised (Rule 4). This is mirrored precisely by the code generated for `case` expressions.

In code generation for most languages the act of pushing a continuation (or return address) is immediately followed by a function call. It is worth noticing in passing that this is not the case for the STG language; there may be a significant gap between the instruction(s) which push the continuation and the instruction (if any) which actually transfers control. For example, consider the expression

```
case (case f x of ...)
of ...
```

The continuation for the outer `case` is pushed, then the continuation for the inner `case`, and then the call to `f` is made. Few programmers write this sort of code, but it arises as a result of program transformations within the compiler. The order of the two `case` expressions can be interchanged, but only at the risk of code duplication.

The main point of interest is how continuations are represented. Recall that a continuation in the operational semantics consists of two parts:

1. The alternatives of the `case` expression.

2. The environment ρ in which they should be executed.

The representation of alternatives is intimately connected with the code generated for primitive values and constructors, so we defer discussion of the first topic until the following sections. Environment-saving is independent of constructors, so we discuss it first.

9.4.1 Saving the local environment

The local environment is saved by saving in the stacks the values of all variables which are live (that is, free) in any of the alternatives. The way in which a live variable is saved depends on where it currently resides:

- It may already be in a stack, for example if it was an argument to the current closure. No code need be generated.

- It may be in a register, or it may be bound to an offset from the heap pointer. In this case it must be saved in the appropriate stack.

- It may be in the closure currently pointed to by `Node`. In this case there are two possibilities: save the variable itself, or save `Node`.

  The latter reduces the number of saves because saving `Node` effectively saves the values of several variables at once. On the other hand, an extra memory access is subsequently required to get the value of the variable.

  Another problem with saving `Node` is that the *entire* contents of the closure must then be retained by the garbage collector, even though the continuation may only use some of its fields. The space-leak avoidance mechanisms described in Section 9.3.3 cannot be applied. (It is possible to rescue the space behaviour by compiling a bitmask to indicate which fields of the closure are live, but it is complicated.)

  Our current policy is to avoid these difficulties by saving all variables individually.

When saving a variable in a stack, we can economise on stack usage by re-using stack slots belonging to variables which are now dead. There is also a useful side benefit: the structure pointed to by dead pointers in the stack cannot be reclaimed by the garbage collector, so overwriting such pointers with live ones helps to avoid space leaks. One could experiment (though we have not yet done so) with generating extra instructions to overwrite dead pointers whose slots are not to be reused, specifically in order to make their space reclaimable. We call this "stack stubbing".

9.4.2 Constructor applications

The expression scrutinised by a `case` expression must eventually evaluate either to a primitive value or a constructor application. We deal with the latter case in this section, deferring the primitive case to Section 9.5.

The code generated for a constructor application must return control to the appropriate alternative of the `case` expression, making the argument of the application available to the alternative. This is just what is done by the rules for constructors in the operational semantics (Rules 5, 6 and 7).

For example, consider the expression:

```
let
    hd = {} \n {xs} -> case xs of
                       Cons {y,ys} -> y {}
                       Nil {}      -> error {}
    single = {w,ws} \n  {} -> Cons {w, ws}
in
hd {single}
```

where `w` and `ws` are bound by some enclosing scope. The code generated for the `case` expression in `hd` pushes a continuation and enters the closure for `xs`, which is bound to `single` in this case. The code for the constructor application `Cons {w,ws}`, in the body of `single`, should return control to the appropriate alternative, returning `w` and `ws` in some agreed way. (Remember that constructor applications in the STG language are always saturated.)

There are two main aspects to consider:

- There may be several alternatives, so there is the question of how the appropriate one is selected.
- The constructor for a particular alternative may have arguments, in which case these need to be communicated to the code for the alternative.

These two issues are now discussed in turn.

9.4.3 Selecting the alternative

The simplest possible representation for the alternatives is a single code label, which we call a *return address*, pushed on the B stack. Control is returned by the constructor application to the labelled code when evaluation of the scrutinised object is completed.

If there is only one member of the algebraic data type (tuples, for example), the evaluation of the (single) alternative can proceed immediately. If there is more than one member of the type (lists, for example), the tag of the object is put into a particular register `RTag`, and a C `switch` statement is generated to perform the case analysis on `RTag`. (In a native-code generator, this multi-way jump can be compiled using a tree of conditionals or using a jump

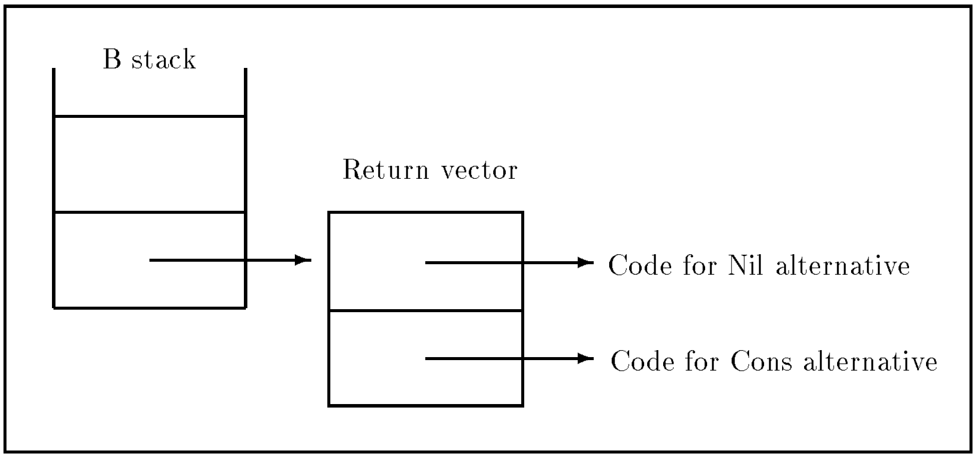


Figure 6: Vectored returns

table, depending on the sparsity of the alternatives, but using C as our target code allows to delegate this choice to the C compiler.)

This is not the only possible representation for the alternatives. Another possibility is to represent the alternatives by a pointer to a table of code labels, with one entry in the table for each constructor in the data type. Figure 6 illustrates the situation for a `case` expression which is scrutinising a list. A pointer to this table, which we call a *return vector*, is pushed on the B stack by the code for the `case` expression. Then, instead of loading `RTag`, the code for the constructor application can transfer control directly to the appropriate destination, thus saving a jump. We call this a *vectored return*, and the pointer to the return vector a *vectored return address*.

The important point to note is that *the return convention can be chosen independently on a datatype by datatype basis*. A particular `case` expression will only scrutinise objects of a particular type. In practice, we (somewhat arbitrarily) use vectored returns for data types with up to eight constructors, because this catches the vast majority of data types without risking wasting (code) space on large sparsely-used return vectors.

9.4.4 Returning the constructor arguments

There is a correspondingly simple convention available for communicating the constructor arguments to the alternative: make the `Node` register point to a constructor closure containing the appropriate values. The code for the alternative can then address its components by indexing from the `Node` register as usual.

This works fine, and is simple enough, but a much better alternative is readily available: if there are sufficiently few arguments, return them in registers! If the closure being scrutinised is already a constructor then not much is gained; indeed something may be lost, because all its components may be loaded into registers when perhaps the alternative only requires one of them. But there is a terrific gain when a thunk is scrutinised, because it may thereby avoid ever building the constructor in the heap. The most critical example of this is ordinary integer arithmetic. Consider the following example:

```
neg = {} \n {x} -> case x of
                     MkInt {x#} -> case (neg# {x#}) of
                                       y# -> MkInt {y#}
```

`neg` is the function which negates an integer. It operates by evaluating the integer `x` to extract its primitive value `x#`, negating it to give `y#`, and then returning the integer `MkInt {y#}`. Now, under the simple return convention, the boxed value `MkInt {y#}` would be constructed in the heap, `Node` would be made to point to it, and control returned to the continuation. If, instead, the component of the constructor, `y#` is returned in a register, the value need never be built in the heap, except as result of an update (Section 10). It turns out that this has a big effect on performance .

As before, the important point is that *the return convention can be chosen on a datatype by datatype basis*. Integers are not a special case. For example, list "cons" cells can be returned by putting the head and tail values into specific registers. (Independently, a vectored or non-vectored return convention can be chosen.) Even the choice of which registers are used to return values can also be made independently for each data type. For example, a floating-point number can be returned in a floating-point register.

For the reason given before, it is probably not a good idea to return constructors with many arguments entirely in registers. We therefore make a virtue of necessity (there are only a limited number of registers) and return larger constructors by allocating them in the heap and making `Node` point to them.

It turns out that the return-in-registers convention makes updates substantially harder, as we shall see, but the gain is well worth it.

9.5 Arithmetic

Suppose that the expression scrutinised by a `case` turns out to evaluate to a primitive value; that is, either a primitive literal (Rule 9), a variable whose value is primitive (Rule 10), or an arithmetic operation whose result is primitive (Rule 14). All three of these rules enter the *ReturnInt* state, which takes action depending on the alternatives stored on top of the return stack.

The return convention for primitive values is simple. The continuation on top of the return stack is always a return address, pointing directly to the continuation code. The primitive value itself is returned in a standard return register, chosen independently for each primitive data type. For example, one register can be used for integers and another for floating point values. The code generated for a primitive literal simply loads the specified value into the appropriate return register, pops the return address from the B stack and jumps to it. Similarly, the code generated for a primitive variable just loads the value of the variable into the return register and returns; and arithmetic follows in the same way.

The code generated at the return address implements the case analysis implied by the alternatives (if any), using a suitable C `switch` statement. Often there is only one alternative, which binds a variable to the value returned. This is easily done by binding the variable to the appropriate return register.

There is a very important special case, when compiling expressions of the form:

$$\texttt{case}\ v_1 \oplus v_2\ \texttt{of}\ alts$$

for built-in arithmetic operations $\oplus$. It would be pointless to push a return address, evaluate $v_1 \oplus v_2$, and return to the return address! These operations can easily be short-circuited, and it is practically essential to do so. We can express this equivalence by doing some simple transformations on the rules to give the derived rules:

$$(18)\quad \boxed{\begin{array}{lll} & Eval \left(\begin{array}{l}\texttt{case}\ \oplus\ \{x_1, x_2\}\ \texttt{of} \\ \quad k_1\ \texttt{->}\ e_1\ ; \\ \quad \ldots \\ \quad k_n\ \texttt{->}\ e_n\ ; \\ \quad x\ \texttt{->}\ e \end{array}\right) \rho \left[\begin{array}{l} x_1 \mapsto Int\ i_1 \\ x_2 \mapsto Int\ i_2 \end{array}\right] & as\ rs\ us\ h\ \sigma \\ \Longrightarrow & Eval\ e\ \rho[x \mapsto Int\ (i_1 \oplus i_2)] & as\ rs\ us\ h\ \sigma \\ \text{where} & k_j \neq i_1 \oplus i_2 \qquad (1 \leq j \leq n) & \end{array}}$$

$$(19)\quad \boxed{\begin{array}{lll} & Eval \left(\begin{array}{l}\texttt{case}\ \oplus\ \{x_1, x_2\}\ \texttt{of} \\ \quad \ldots \\ \quad k\ \texttt{->}\ e\ ; \\ \quad \ldots \end{array}\right) \rho \left[\begin{array}{l} x_1 \mapsto Int\ i_1 \\ x_2 \mapsto Int\ i_2 \end{array}\right] & as\ rs\ us\ h\ \sigma \\ \Longrightarrow & Eval\ e\ \rho & as\ rs\ us\ h\ \sigma \\ \text{where} & k = i_1 \oplus i_2 & \end{array}}$$

In particular, once this optimisation is implemented, the expression

$$\texttt{case}\ \oplus\ \{x_1, x_2\}\ \texttt{of}\ x\ \texttt{->}\ e$$

compiles to the simple C statement:

```
x = x1 ⊕ x2
```

where `x`, `x1` and `x2` are the C local variables used to hold the values of x, x_1 and x_2.

10 Adding updates

So far everything has been quite tidy: tree reduction is nice and easy. Sadly, graph reduction is harder, and updates are quite complicated. This is much the trickiest part of the Spineless Tagless G-machine. Still, we begin bravely enough.

10.1 Representing update frames

Recall that when a closure is entered it has the opportunity to push an update frame onto the update stack. An update frame consists of

- A pointer to the closure to be updated.
- The saved argument and return stacks.

After an update frame is pushed, execution continues with empty argument and return stacks.

Of course, we don't actually copy the argument and return stacks onto a separate update stack! Instead, we dedicate two registers, called the *stack base registers*, to point just below the bottom-most word of the A and B stacks respectively. The argument and return stacks can now be "saved" and then "made empty" merely by saving the stack base registers in the update frame, and making them point to the current top of the A and B stacks.

Where is the update stack kept? It could be represented by a separate stack all of its own, but we have chosen to merge it with the B stack. The minor reason for this is to avoid yet another stack. The major reason is that it makes available an important optimisation which we discuss below (Section 10.3).

To conclude, the operation of pushing an update frame (Rule 15) is done by:

- Pushing an update frame onto the B stack.
- Setting the stack base registers to point to the top of their respective stacks.

(The alert reader will have spotted that a pointer (to the closure to be updated) has thereby ended up on the B stack. We discuss this in Section 10.7.)

An update is triggered in one of two ways: either a function finds too few arguments on the stack, or a constructor application finds an empty return stack. These two situations are discussed in the following sections.

10.2 Partial applications

When a closure is entered which finds too few arguments on the stack, an update is triggered. This is described by Rules 17 and 17a. The check for too few arguments is called the *argument satisfaction check*, and occurs at the start of the code for every closure which takes one or more arguments (cf Section 9.3.2).

A minor complication is that the arguments are split between the A and B stacks, but this presents little difficulty. If the *last* argument is available then certainly all the others will be, so the check is performed only on the stack which contains the last argument.

The argument satisfaction check is performed by subtracting the stack base pointer of the appropriate stack from the corresponding stack pointer, giving a difference in words. This is compared with the (statically calculated) number of words required for all the arguments which are passed on that stack. If too few words are present, a jump is taken to a runtime system routine, `UpdatePAP`, which performs the update. Once the update has been done, `UpdatePAP` concludes by re-entering the closure, which `Node` should be pointing to. The argument satisfaction check is thereby performed again, as Rule 17 requires, in case a further update is needed.

A special case is required for top-level closures, because the code entering the closure may not have made `Node` point to it (Section 9.2.1). In this case, just before jumping to `UpdatePAP`, the argument-satisfaction-check code loads a pointer to the closure into `Node`. (Recall that top-level closures are statically allocated, so their address is fixed.)

What does `UpdatePAP` do? It follows Rule 17a:

1. First, it builds in the heap a closure representing the partial application, whose structure is given below.
2. Next, it overwrites the closure to be updated (obtained from the update frame) with an indirection to the newly-constructed closure.
3. It restores the values of the stack-base registers from their values saved in the update frame.
4. It removes the update frame from the B stack, sliding down the portion of the stack (if any) which is above it.
5. Finally, it re-enters the closure pointed to by `Node`.

What does the partial-application closure look like? In the most general case it contains:

- The info pointer `PAP_Info`.
- The total size of the closure, and the number of pointers in it. As well as being used by the storage manager, this information is required by the standard-entry code of `PAP_Info` (see below).
- The pointer to the function closure, which is in `Node`.
- The contents of the A stack between the top of stack and its stack base pointer.
- The contents of the B stack between the top of stack and its stack base pointer.

If this partial-application closure is entered, the standard-entry code of `PAP_Info` pushes the saved stack contents onto their respective stacks (using the size information to determine how many words to move to which stack), and then enters the function closure saved in the partial application closure. Its garbage-collection routines use the size information stored in the closure to guide their work.

So much for the general case. A couple of optimisations are readily available. Firstly, a collection of specialised `PAP_Info` pointers can be provided for various combinations of numbers of pointer and non-pointer words. For example, `PAP_Info_1_0` is used when there is one pointer word and no non-pointers. The advantages of such specialised info pointers are: there is no need to store the field sizes in the closure; and the entry and garbage-collection code is faster because it has no interpretive loop. There is, of course, a small execution-time cost in `UpdatePAP` to decide whether a special case applies.

Secondly, if the new closure is small enough it can be built directly on top of the closure to be updated.

10.3 Constructors

The other way in which an update can be triggered is when a constructor finds an empty return stack. It looks as though the code for a constructor application has to test for an empty return stack; indeed this is just what is implied by Rule 16. This looks expensive, because constructors are so common. Furthermore, the return stack is almost always non-empty, so the test is in vain. Data structures are often built once and then repeatedly traversed. Each time pattern matching is performed on a data structure, a continuation is pushed on the return stack, and the closure representing the data structure is entered. Since it is already evaluated, it returns immediately (perhaps using a vectored return), but it must first perform the return-stack test.

So now comes the tricky part. Since update frames and continuations are both stored on the B stack, if a return address is not on top of the stack, then an update frame must be. *If we make the top word of each update frame into a code label,* `UpdateConstr`*, the constructor could just return without making any test.* In the common case where there is no update frame this does just what we want. If an update is required there will be an update frame on top of the B stack, so the "return" will land in the `UpdateConstr` code, which can perform the update and then return again, perhaps to another update frame, or perhaps to the "real" continuation.

Before we examine the complications, it is worth looking at the crude costs and benefits. The cost is one extra instruction for every update frame pushed. The benefit is the omission of a couple of instructions (one of them a conditional jump) from every constructor evaluation. If data structures are traversed repeatedly, constructor evaluations will occur substantially more often than updates. The benefits look significant.

The trouble is that this trick interacts awkwardly with the various return conventions for constructors discussed in Sections 9.4.3 and 9.4.4. With the simple return conventions, everything works fine. The `case` alternatives are always represented by a simple code label on the B stack, and the result is returned by making `Node` point to the constructor closure. All `UpdateConstr` need do is to overwrite the closure to be updated with an indirection to this constructor closure, restore the stack base registers, and return again.

10.4 Vectored returns

Life gets more complicated when we add vectored returns. There is no problem with providing a vectored form of `UpdateConstr`; each entry in its return vector points to code which performs the update and then returns in its turn in a vectored fashion. The difficulty is that *when the update frame is created, the return convention is not known.* This is because the type of the expression may be polymorphic. Consider, for example, the `compose` function, which looks like this in the STG language:

```
compose = {} \n {f,g,x} -> let gx = {g,x} \u {} -> g {x}
                           in f {gx}
```

The code for the closure `gx` does not know its type. Hence, when it pushes the update frame it cannot know whether a vectored return is to be expected or not. In short, `UpdateConstr`

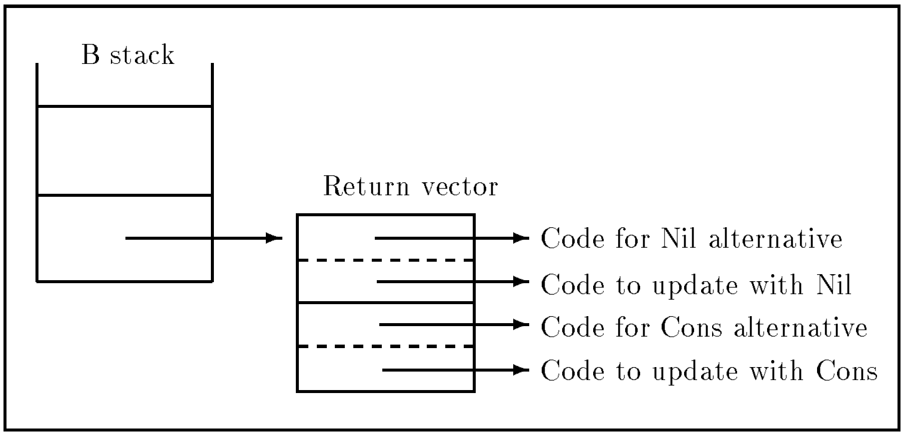


Figure 7: Vectored updates

must be able to cope with either a vectored or a non-vectored return.

If we were generating machine code this would present little problem. We just adopt the convention that the pointer to a vector table points just after the end of the table, so that the table is accessed by indexing backwards from the pointer. Now `UpdateConstr` labels ordinary code immediately preceded by its vector table. Sadly, C does not allow us to specify the relative placement of data and code in this way, so instead we have to adopt the convention that non-vectored returns behave just like vectored returns through a vector table with one entry (cf Section 7.6). This imposes an extra indirection on non-vectored returns.

10.5 Returning values in registers

Unfortunately, matters get worse when we consider the idea of returning constructor values in registers. Now `UpdateConstr` has no way to figure out how to perform the update, because it has no way to tell what return convention is being used. Can the closure which pushed the update frame push a version of `UpdateConstr` appropriate for the data type? As just discussed, the answer is no, because of polymorphism.

This looks like a rather serious problem. There is a way round it, but it is rather tricky.

The idea is this: the update frame may not know the type of the value being returned, *but the* `case` *expression which caused the evaluation in the first place certainly does.* So `UpdateConstr` does not perform an update at all; it merely records that an update is required, by placing a pointer to the closure to be updated in a special register `UpdatePtr`. It is up to the `case` expression continuation to perform the update. How does the continuation "know" whether an update is pending? Simple: each entry in the `case`-expression's return vector is expanded to a pair of code labels (Figure 7). The first of these is just as before (ie the code for the `case`-expression alternative); the other performs an update on the closure pointed to by `UpdatePtr`, and then jumps to the first. We call these the *normal return code* and *update return code* respectively. All `UpdateConstr` has to do to precipitate the update is to return to the update return code rather than the normal return code, which it can do merely by increasing its offset into the return vector by one.

The costs are surprising slight. There is a static space cost, as each vector table now doubles in size. The extra code to perform the update can be generated once only for each constructor, and then pointed to from all the return vectors for its data type. This per-constructor update code can still find its way to the appropriate `case` alternative provided the pointer to the return vector is kept handy in a register.

One objection remains, which looks serious: suppose there are several update frames on top of each other before the "real" continuation is reached? This can arise in programs like the following:

```
let x1 = ...
in
let x2 = {x1} \u {} -> x1
in
let x3 = {x2} \u {} -> x2
in
...x3...
```

When `x3` is entered, it will push an update frame and then enter `x2`, which will push another update frame and enter `x1`. When `x1` reaches head normal form it will find two update frames on top of the stack, reflecting the fact that both the closure for `x2` and that for `x3` must be updated with `x1`'s value. This looks like a rather special case, but it does arise in practice: any closure which may return the value of another closure has the same property. For example, the definition of `x3` could be:

```
x3 = {x2,z} \u {} case ... of
                    Nil {} -> x2
                    Cons {p,ps} -> p
```

The problem with multiple update frames is that the `UpdatePtr` register can only point to one closure! Fortunately there is an easy solution. Recall that all return vectors *including that for* `UpdateConstr` consist of paired entries, and that `UpdateConstr` returns to the *update return code* rather than the *normal return code* of the pair. The update return code of the `UpdateConstr` vector therefore knows that it is not the first update frame, and so `UpdatePtr` is already in use. One possibility would be to chain together all the closures to be updated, but there is a simpler way: the second (and subsequent) update frames just update their closures with an indirection to the one pointed to by `UpdatePtr`. When the latter is finally updated all the updating has been successfully completed. This can result in chains of at most two indirections; and remember that indirections are all eliminated by the garbage collector.

Finally, what of non-vectored returns? We still need a pair of code addresses to return to, as in the vectored case. If the results are returned in a heap-allocated closure pointed to by `Node` there is no problem: the update return code just performs the update and jumps to the normal return code. If results are being returned in registers, then the update return code needs to perform case analysis on `RTag` to figure out how to perform the update. As before, there need be only one copy of this update return code.

10.6 Update in place

The update technology just described has another very important benefit: *it allows the updated closure to be overwritten directly with the result (if it is small enough), rather than being overwritten with an indirection to the result.*

Up to now, our uniform return convention has meant that closures are only ever overwritten by indirections, even though it is often the case that it is in principle possible to overwrite it directly with the result. Not only does this introduce extra indirections but, more seriously, it gives rise to a lot of extra memory allocation. Kieburtz and Agapiev specifically identify and quantify this shortcoming (Kieburtz & Agapiev [1988]).

If the dual-return mechanism of the previous section is used, however, then this shortcoming can easily be overcome. The code performing the update knows exactly how the closure is laid out so, if it is small enough, it can directly overwrite the closure to be updated. For example, here is the code to perform updates for a list `Cons` cell:

```
ConsUpd() {
        UpdatePtr[0] = Cons_Info;
        UpdatePtr[1] = Head;
        UpdatePtr[2] = Tail;
        JUMP( ReturnVector[2] );
}
```

Here we are assuming that the `Cons` constructor returns its head in a register `Head` and tail in `Tail`. `Cons_Info` is the info table for the `Cons` constructor. The address of the return vector is assumed to be in a register `ReturnVector`. The offset of 2 picks the first code address of the pair for `Cons`.

For larger constructors, the new object cannot be built directly on top of the old one, so a new object must be built in the heap and the old one updated with an indirection to it. The usual heap-exhaustion check must be made, and garbage collection triggered if no space remains. The update routine must then be careful to save any pointers being returned in registers into a place where the garbage collector will find them.

How does the updating code know if the closure to be updated is large enough? There are two main possibilities:

- We can establish a global convention for the minimum size of updatable closures; making them all large enough (by padding if necessary) to contain a list cons cell seems a plausible guess. There is scope for another small optimisation here: if a closure is being allocated whose type is (say) `Int`, then it cannot possibly be updated by anything other than an integer, so it does not need to be padded out to cons-cell size.
- The updating code can look in the closure's info table to find its size, and either update in place or use an indirection, depending on what it finds. This costs more time than the unconditional scheme, but has the merit that it will succeed in updating in place more often. This is another aspect of the design which we plan to quantify.

Update in place is not always desirable. Suppose that the value of the thunk turned out to be an *already-existing* constructor which returns its components in registers. Then update-in-place will overwrite the thunk with a *copy* of the constructor. No work is duplicated thereby, but there is a potential loss of space if both copies stay live for a while. At worst, a great many copies of the same object could be built in this way, substantially increasing the space usage of the program. The point is this: once copied, there is no cheap mechanism for "commoning up" the original with the copy.

Our preliminary measurements suggest that up to 10% of all updates copy an already-existing constructor, though the figure can occasionally be much higher (for the program reported by Runciman & Wakeling [1992] it is 47%). We plan to make more careful measurements to see how important this effect is. If it turns out to be significant we will implement a simple extension of the dual-return-address scheme outlined in Section 10.5, whereby each element of the return vector is a *triple* of return addresses. We omit the details, but the scheme has the effect of always updating a thunk with an indirection whenever the value of the thunk is an already-existing heap object.

10.7 Update frames and garbage collection

Update frames, which include a pointer to the closure to be updated, are kept on the B stack. At first this looks rather awkward, because the garbage collector expects all pointers to be on the pointer stack, but it actually turns out to be quite convenient, because of the following observation: *if the only pointer to a closure is from an update frame, then the closure can be reclaimed, and the update frame discarded.*

We can take advantage of this during garbage collection in the following way. First perform garbage collection as usual, but without using the pointers from update frames as roots. Now, look at each update frame and see if it points to a closure which has been marked as live. If so, and a copying collector is being used, adjust the pointer to point to the new copy of the closure. If not, squeeze the update frame out of the B stack altogether.

It is easy to find all the update frames, because the stack base register for the B stack always points to the topmost word of the topmost update frame; and the saved stack base register for the B stack points to the next update frame, and so on. This gives a top-to-bottom traversal, but it turns out that a bottom-to-top traversal makes the "squeezing-out" process much more efficient, for two reasons:

- Since the squeeze must move data towards the bottom of the stack (otherwise the stack would creep up in memory!), working from bottom to top means that each word of the B stack is moved only once.
- When an update frame is removed, the stack-base pointers for the next update frame above it need to be adjusted. This is easy to do when working bottom to top.

Happily, it is easy to make a top-to-bottom traversal, reversing all the pointers, and then make the bottom-to-top traversal to do the work.

The result of all this is that the garbage collector reclaims redundant update frames. The main benefit is the saving in updates performed. This optimisation was performed, but not documented, in Fairbairn and Wray's original TIM implementation.

10.8 Global updatable closures

As previously discussed (Section 9.1), each globally-defined variable is bound to a statically-allocated closure. Since such closures have no free variables (except of course other statically-allocated closures), there is no need to treat them as a source of roots during garbage collection.

But some of these global closures may have no arguments, and hence be updatable: we call such such argument-less top-level closures *constant applicative forms* or CAFs. For example, `ints` is a CAF whose value is the infinite list of integers:

```
ints = {} \u {} from {zero}
zero = {} \n {} MkInt {0#}
```

where `from` is a function returning the infinite list of integers starting from its argument.

There are two difficulties:

1. If such a CAF is updated, there will be pointers from the static space into the dynamic heap. The question is: how is the garbage collector to find all such pointers?
2. With the garbage-collection techniques described in Section 7.3 closures in static space need different garbage-collection code from those in the dynamically-allocated heap. The two are readily distinguishable (by address) but it is unfortunate if *every* update is slowed down by a test when the vast majority of updates are to dynamic closures.

There is more than one way to solve this problem, but the one we have adopted is as follows. The idea is to arrange that:

1. CAFs which are being evaluated, or whose evaluation is complete, are linked together onto the *CAF list*, which is known to the garbage collector. This solves the first of the above problems.
2. All update frames point to closures in the dynamic heap, thus solving the second problem.

We achieve these goals by adding a little extra code to the start of the standard-entry code for a CAF (Figure 8). The extra code does the following: it allocates a black hole closure in the heap whose purpose is to receive the subsequent update; it pushes an update frame pointing to this black hole; and it overwrites the static CAF closure with a three-word `CAFlist` cell, pointing to the black hole in the heap, and linked onto the CAF list.

In the example shown in Figure 8, the CAFs `p`, `q` and `r` have all been entered, and hence are linked onto the CAF list (we use `CL` to abbreviate the info pointer for a `CAFList` cell). The evaluation of `p` is not yet complete, so it still points to a black hole in the dynamic heap (info pointer `BH`). The evaluation of `q` has been completed, and the black hole has been updated with an indirection (info pointer `I`) to its value. `s` has not been entered, so it consists of a info pointer (`S`) only.

A `CAFList` cell looks like any other closure. If entered, it simply enters the heap-allocated closure to which it points, behaving just like an indirection. The garbage collector knows

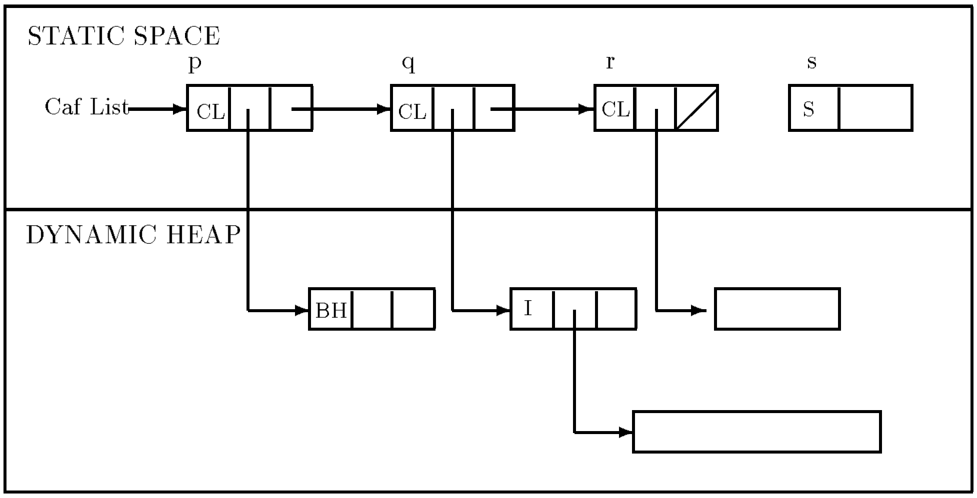


Figure 8: Global updates

about the CAF list, and walks it iteratively, evacuating the heap-allocated closure to which each cell points, and updating the cell appropriately. This is slightly pessimistic, since it holds onto the value of every CAF even though the program may never reference it again, but there is no avoiding this unless the garbage collector traverses the code as well.

11 Status and profiling results

We have built a compiler for Haskell whose back end is based on the STG machine, just as described above. The whole implementation has been constructed rather carefully so that it may be used as a "motherboard" into which other implementors may "plug in" their optimisation passes. All the source code is available by anonymous FTP by contacting `haskell-request@dcs.glasgow.ac.uk`.

Apart from the STG machine technology described in the current paper, the major innovations of the compiler are:

- The systematic use of unboxed values to implement built-in data types.
- A new approach to input/output based on monads, which allows the entire I/O system to be written in Haskell (Peyton Jones & Wadler [1993]). This is done via a general-purpose mechanism which allows arbitrary calls to be made from Haskell to C. As a result the I/O system can be readily extended without modifying the compiler or its runtime system.
- The Core language, which serves as the main intermediate data type in the compiler, is actually based on the second-order lambda calculus, complete with type abstraction and application. This permits us to maintain complete type information in the presence of extensive program transformation, as well as accommodating other front ends whose type system is more expressive than Haskell's.

The implementation covers almost the whole language, but virtually no optimisations have yet been implemented. As a result, we have not compared its absolute performance with other compilers. (This omission is an important shortcoming of this paper, which will be rectified by a follow-up paper.)

We have begun to gather simple dynamic statistics, however. Figure 9 shows some output taken from a run of a simple type-inference program. This program takes some 600 lines of Haskell source code (apart from functions used from the Prelude), and the sample run allocated about 10 megabytes of heap. The profile has the following main headings:

Allocations, split into various categories. Most allocation is for thunks. The proportion of data-value allocation seems surprisingly low, because most data values are built by updating a thunk, rather than by performing new allocation in a `let(rec)`. This program uses monads heavily, so quite a lot of function-valued closures are allocated.

Stack high-water marks are self explanatory.

Enters, with a classification of what kind of closure is being entered. In this case, a rather low proportion of function calls (34%) bypass the argument satisfaction check, again due to the very higher-order nature of the program.

Returns, which give information about the data-value returns which took place. In this run, almost all were vectored and in registers. The third classification tells how many returns were from entering an already-evaluated constructor (some 50% in this case). In the cell model, the enter/return sequence would not be performed for these cases.

Update frames classifies various forms of update frame, which is rather uninformative in this case.

Updates. The fifth line counts the number of times two or more update frames were stacked directly on top of one another. The last line counts the number of updates in which an already-existing value was copied by an updates (Section 10.6).

Acknowledgements

This paper has been a long time in gestation. I would like to thank those who have been kind enough to give me help and guidance in improving it. Andrew Appel made detailed comments about the relationship of this work to his. Geoff Burn, Cordy Hall, Denis Howe, Rishiyur Nikhil, Chris Okasaki, Julian Seward, and the two anonymous referees gave very useful feedback. The implementation of the compiler itself owes much to the work of Cordy Hall and Will Partain.

Sadly, my friend and colleague Jon Salkild, who co-authored the first STG paper, died most suddenly in 1991.

```
ALLOCATIONS: 73920 (231212 words total:
    70387 admin, 136489 goods, 24336 slop)
avg #words of: admin goods slop
  14426 ( 19.5%) function values 1.0 1.7 0.1
  45131 ( 61.1%) thunks 1.0 1.7 0.5
   9609 ( 13.0%) data values 1.0 1.7 0.0
      0 (  0.0%) big tuples
      0 (  0.0%) partial applications
    407 (  0.6%) black-hole closures 3.0 0.0 0.0
   4341 (  5.9%) partial-application updates 0.0 4.4 0.0
      6 (  0.0%) data-value updates 0.0 4.5 0.0

Total storage-manager allocations: 49594 (235559 words)

STACK USAGE:
A stack max. depth: 222 words
B stack max. depth: 446 words

ENTERS: 142737, of which 41914 (29.4%) direct to the entry code
  [the rest indirected via Node's info ptr]
  35626 ( 25.0%) thunks
  24713 ( 17.3%) data values
  64346 ( 45.1%) function values
  [of which 21604 (33.6%) bypassed arg-satisfaction chk]
   4788 (  3.4%) partial applications
  13264 (  9.3%) indirections

RETURNS: 49212
  48991 ( 99.6%) in registers [the rest in the heap]
  49210 (100.0%) vectored [the rest unvectored]
  24499 ( 49.8%) from entering a new constructor
  [the rest from entering an existing constructor]

UPDATE FRAMES: 35626 (0 omitted from thunks)
  35626 (100.0%) standard frames
      0 (  0.0%) constructor frames
      0 (  0.0%) black-hole frames

UPDATES: 35626
  25788 ( 72.4%) data values
  [25781 in place, 6 allocated new space, 1 with Node]
   4349 ( 12.2%) partial applications
  [8 in place, 4341 allocated new space]
   5489 ( 15.4%) updates followed immediately
   2594 ( 10.1%) in-place updates copied
```

Figure 9: Example output of dynamic profiling information

Bibliography

R Alverson, D Callahan, D Cummings, B Koblenz, A porterfield & B Smith [June 1990], "The Tera computer system," in *Proc International Conference on Supercomputing, Amsterdam*.

AW Appel [1987], "Garbage collection can be faster than stack allocation," *Info Proc Lett* 25, 275–279.

AW Appel [1992], *Compiling with continuations*, Cambridge University Press.

AW Appel [Feb 1989], "Simple generational garbage collection and fast allocation," *Software – Practice and Experience* 19, 171–183.

AW Appel & T Jim [Jan 1989], "Continuation-passing, closure-passing style," in *Proc ACM Conference on Principles of Programming Languages*, ACM, 293–302.

ZM Ariola & Arvind [June 1991], "A syntactic approach to program transformations," in *Symposium on Partial Evaluation and Semantics-Based Program Manipulation, Yale*.

L Augustsson [1987], "Compiling lazy functional languages, part II," PhD thesis, Dept Comp Sci, Chalmers University, Sweden.

L Augustsson & T Johnsson [Sept 1989], "Parallel graph reduction with the $< \nu, G >$-machine," in *Proc IFIP Conference on Functional Programming Languages and Computer Architecture, London*, ACM.

Henry Baker [Apr 1978], "List processing in real time on a serial computer," *CACM* 21, 280–294.

JF Bartlett [Jan 1989], "SCHEME to C: a portable Scheme-to-C compiler," DEC WRL RR 89/1.

A Bloss, P Hudak & J Young [1988], "Code optimizations for lazy evaluation," *Lisp and Symbolic Computation* 1, 147–164.

Geoff Burn, SL Peyton Jones & John Robson [July 1988], "The Spineless G-machine," in *Proc ACM Conference on Lisp and Functional Programming, Snowbird*, 244–258.

C Consel & O Danvy [Sept 1991], "For a better support of static data flow," in *Functional Programming Languages and Computer Architecture, Boston*, Hughes, ed., LNCS 523, Springer Verlag, 496–519.

EC Cooper & JG Morrisett [Dec 1990], "Adding threads to Standard ML," CMU-CS-90-186, Dept Comp Sci, Carnegie Mellon Univ.

AJT Davie & DJ McNally [1989], "CASE - a lazy version of an SECD machine with a flat environment," in *Proc IEEE TENCON, Bombay*.

Jon Fairbairn & Stuart Wray [Sept 1987], "TIM - a simple lazy abstract machine to execute supercombinators," in *Proc IFIP conference on Functional Programming Languages and Computer Architecture, Portland*, G Kahn, ed., Springer Verlag LNCS 274, 34–45.

AJ Field & PG Harrison [1988], in *Functional programming*, Addison Wesley.

P Fradet & D Le Metayer [Jan 1991], "Compilation of functional languages by program transformation," *ACM Transactions on Programming Languages and Systems* 13.

P Henderson [1980], *Functional programming: application and implementation*, Prentice Hall.

R Hieb, RK Dybvig & C Bruggeman [June 1990], "Representing control in the presence of first-class continuations," in *Proc Conference on Programming Language Design and Implementation (PLDI 90)*.

P Hudak, SL Peyton Jones, PL Wadler, Arvind, B Boutel, J Fairbairn, J Fasel, M Guzman, K Hammond, J Hughes, T Johnsson, R Kieburtz, RS Nikhil, W Partain & J Peterson [May 1992], "Report on the functional programming language Haskell, Version 1.2," *SIGPLAN Notices* 27.

John Hughes [Apr 1989], "Why functional programming matters," *The Computer Journal* 32, 98–107.

PZ Ingerman [1961], "Thunks," *Comm ACM* 4, 55–58.

E Ireland [Jan 1992], "The Lazy Functional Abstract Machine," in *Proc 15th Australian Computer Science Conference, Hobart*, World Scientific Publishing.

Thomas Johnsson [1985], "Lambda lifting: transforming programs to recursive equations," in *Proc IFIP Conference on Functional Programming and Computer Architecture*, Jouannaud, ed., LNCS 201, Springer Verlag, 190–205.

Thomas Johnsson [1987], "Compiling lazy functional languages," PhD thesis, PMG, Chalmers University, Goteborg, Sweden.

Thomas Johnsson [June 1984], "Efficient compilation of lazy evaluation," in *Proc SIGPLAN Symposium on Compiler Construction, Montreal*.

R Jones [March 1991], "Tail recursion without space leaks," Department of Computer Science, University of Kent.

R Kelsey [May 1989], "Compilation by program transformation," YALEU/DCS/RR-702, PhD thesis, Department of Computer Science, Yale University.

RB Kieburtz [Oct 1987], "A RISC architecture for symbolic computation," in *Proc ASPLOS II*.

RB Kieburtz & B Agapiev [Sept 1988], "Optimising the evaluation of suspensions," in *Proc workshop on implementation of lazy functional languages, Aspenas*.

H Kingdon, D Lester & GL Burn [1991], "The HDG-machine: a highly distributed graph reducer for a transputer network," *Computer Journal* 34, 290–302.

PWM Koopman [1990], "Functional programs as executable specifications," PhD thesis, University of Nijmegen.

DA Kranz [May 1988], "ORBIT - an optimising compiler for Scheme," PhD thesis, Department of Computer Science, Yale University.

PJ Landin [March 1965], "A correspondence between Algol 60 and Church's lambda calculus," *Comm ACM* 8, 158–165.

D Lester [Apr 1989], "Combinator graph reduction: a congruence and its applications," PhD Thesis, Programming Research Group, Oxford.

Erik Meijer [Sept 1988], "Generalised expression evaluation," in *Proc workshop on implementation of lazy functional languages, Aspenas*.

E Miranda [Apr 1991], "How to do machine-independent fast threaded code," Dept of Computer Science, Queen Mary and Westfield College, London.

SL Peyton Jones [1987], *The Implementation of Functional Programming Languages*, Prentice Hall.

SL Peyton Jones [1988], "FLIC - a functional language intermediate code," *SIGPLAN Notices* 23.

SL Peyton Jones, C Clack & J Salkild [June 1989], "High-performance parallel graph reduction," in *Proc Parallel Architectures and Languages Europe (PARLE)*, E Odijk, M Rem & J-C Syre, eds., LNCS 365, Springer Verlag, 193–206.

SL Peyton Jones & J Launchbury [Sept 1991], "Unboxed values as first class citizens," in *Functional Programming Languages and Computer Architecture, Boston*, Hughes, ed., LNCS 523, Springer Verlag.

SL Peyton Jones & D Lester [May 1991], "A modular fully-lazy lambda lifter in HASKELL," *Software – Practice and Experience* 21.

SL Peyton Jones & DR Lester [1992], *Implementing functional languages: a tutorial*, Prentice Hall.

SL Peyton Jones & Jon Salkild [Sept 1989], "The Spineless Tagless G-machine," in *Functional Programming Languages and Computer Architecture*, D MacQueen, ed., Addison Wesley.

SL Peyton Jones & PL Wadler [Jan 1993], "Imperative functional programming," in *20th ACM Symposium on Principles of Programming Languages, Charleston*, ACM.

C Runciman & D Wakeling [Apr 1992], "Heap profiling of lazy functional programs," Tech Report 172, Department of Computer Science, University of York.

P Sansom [Aug 1991], "Combining copying and compacting garbage collection," in *Proc Fourth Annual Glasgow Workshop on Functional Programming*, Springer Verlag Workshops in Computer Science.

Mark Scheevel [Aug 1986], "NORMA - a graph reduction processor," *Proc ACM Conference on Lisp and Functional Programming*.

S Smetsers, E Nocker, J van Groningen & R Plasmeijer [Sept 1991], "Generating efficient code for lazy functional languages," in *Functional Programming Languages and Computer Architecture, Boston*, Hughes, ed., LNCS 523, Springer Verlag.

RM Stallman [Feb 1992], "Using and porting Gnu CC, Version 2.0," Free Software Foundation Inc.

GL Steele [1978], "Rabbit: a compiler for Scheme," AI-TR-474, MIT Lab for Computer Science.

William Stoye, Thomas Clarke & Arthur Norman [August 1984], "Some practical methods for rapid combinator reduction," in *Proc 1984 ACM symposium on Lisp and functional programming*, 159–166.

D Tarditi, A Acharya & P Lee [July 1992], "No assembly required: compiling Standard ML to C," School of Computer Science, Carnegie Mellon University.

AP Tolmach & AW Appel [June 1990], "Debugging Standard ML without reverse engineering," in *Proc ACM Conference on Lisp and Functional Programming, Nice*, ACM.

DA Turner [1979], "A new implementation technique for applicative languages," *Software Practice and Experience* 9, 31–49.

PL Wadler [1987], "Efficient compilation of pattern matching," in *The implementation of functional programming languages*, SL Peyton Jones, ed., Prentice Hall, 78–103.

PR Wilson, MS Lam & TG Moher [Jan 1992], "Caching considerations for generational garbage collection," Department of Computer Science, University of Texas.

A The gory details

This appendix contains gory implementation-specific notes for the Glasgow Haskell compiler.

ToDo: add some stuff about stack stubbing.

A.1 Update flags

There are really three kinds of update flag:

- *Updatable.* Update me with my normal form.
- *Single-entry.* Don't update me with my normal form, but you can overwrite me with a black hole to prevent a space leak. I promise I will be entered at most once.
- *Reentrant.* Don't update me with my normal form, or with a black hole. I may be entered (and re-evaluated) more than once. Manifest functions, constructors, and partial applications are always reentrant. Note that the latter two have zero arguments, so zero-arg things may be reentrant.

A.2 Black holes

There are three reasons for overwriting a thunk with a black hole immediately it is entered:

- To prevent space leaks.
- To give a better error message when there's an infinite loop.
- To do thread synchronisation in a parallel system.

The first two can be dealt with in a different way by the garbage collector. Space leaks can be prevented by black-holing the things on the update stack *at GC time.* Infinite loops can be detected by a post-mortem on the update stack, following a stack overflow. So until we need parallelism, black-holing is a waste of instructions.

A.3 Adding fillers

The great invariant is this: if a closure is to be modified, either by being overwritten with a black hole, or by updating, a filler is written over the "tail" of the closure, so that the initial segment is exactly `FIXED_HDR_SIZE+MIN_UPD_SIZE` words long.

MIN_UPD_SIZE is at least 2, to allow for cons cells and linked indirections. Another truth: GEN objects are always bigger than FIXED_HDR_SIZE+MIN_UPD_SIZE+2; that leaves room for a GEN filler following a min-size closure slot.

That means that the black hole and update code have a well-defined fixed-size thing to modify.

All such closures are thunks, of either SPEC or GEN form. If the latter, the filler is set with

```
SET_GEN_FILLER( Node, slop )
```

where slop is the size of the closure (excl fixed hdr, of course) − MIN_UPD_SIZE.

If the closure is of SPEC form, and there are more than MIN_UPD_SIZE words of ptrs + non-ptrs, the filler is set with

```
SET_SPEC_FILLER( Node, slop )
```

where slop is the size of the closure - MIN_UPD_SIZE. In the SPEC case, the slop argument is guaranteed to be an integer, so can be used in building a label. (Remember, SPEC closures have no variable header, so the size is just ptrs + non-ptrs, which the compiler can calculate.)

All this is done at the end of the basic block, just before Node is discarded (or assigned). But if Node is being loaded from the closure itself we have to go via a temporary, so we need:

```
SET_GEN_FILLER_AND_LOAD_NODE( Node, final_node, size )
SET_SPEC_FILLER_AND_LOAD_NODE( Node, final_node, size )
```

If one-space collection isn't being done, these filler macros generate nothing (except of course the LOAD_NODE variants still load Node).

A.4 Performing updates

The update-performing code behaves as follows:

- *If the new closure doesn't fit in a min-size closure slot:*

  ```
  Allocate the new object from the heap
  UPD_IND( &Hp[-xxx], HeapCheck_xxxLive )
  ```

- *If the new closure fits in a min-size closure slot:*

  ```
  UPD_INPLACE( UpdPtr, HeapCheck_xxxLive )
  Fill in UpdPtr[0...]
  if incompletely filled, use SET_UPD_FILLER( UpdPtr, slop )
  ```

  In generational GC the UPD_INPLACE code checks for a old-gen update, and if so allocates a min-size closure in new-space, updates the old-gen thing with a pointer to the new-space thing, and makes UpdPtr point to it.

  The SET_UPD_FILLER fills in any remaining slop. The slop argument is guaranteed to be an integer.

A.5 Lambda-form info

Now we are ready to summarise the (remarkably subtle) questions of entry and update conventions.

	Reentrant	Updatable	Single-entry
Node must point to it	If has fvs; or arity=0 and using cost centres (note 9,12)	Yes (note 6)	If has fvs or using cost-centres (note 9)
Can jump direct to code	Yes (note 8)	No	Yes
Push update frame	No	Yes (note 5)	No
Black hole on entry (note 7)	No	Optional (notes 2,3) `UPD_BH_UPDATABLE`	iff has fvs (notes 4) `UPD_BH_SINGLE_ENTRY`
Use filler (if 1s gc)	No	Yes, unless static (note 11) `SET_xxx_FILLER`	Iff has fvs (note 10) ditto

Note 1. We NO LONGER assume that any closure with no free variables is allocated statically. However, we do assume that (HAS FVS implies NOT STATIC).

Note 2. We never black-hole an updatable static closure. Instead, it will be overwritten with a CAFList cell pointing to a newly allocated black hole.

Note 3. Black-holing can be done by the garbage collector (by running down the update stack and black-holing any pending updatees); and infinite loops detected by a post-mortem on the update stack.

Note 4. A single-entry closure which has free vars is *always* black-holed, to avoid space leaks. The trick mentioned in Note 3 doesn't work for them because they don't sit on the update stack.

One could argue that space leaks from omitting the black-hole for single-entry things are similar to other unavoidable space leaks, but we don't; we black-hole them.

A single-entry closure with no fvs is *never* black-holed; it cannot give rise to a space leak, and we trust the single-entry flag which should mean it can't form part of a loop. (Even if that's not true, the update stack post-mortem would reveal the loop.) Notice that this means that static single-entry closures (which have no fvs) are never black-holed.

Note 5. For updatable static closures, the update frame will point to the newly-allocated black hole.

Note 6. If a closure is updatable, Node must be made to point to it, even if it is a static no-fv closure. Reason: it may have been updated with a CAF indirection. We have to indirect via the closure to get the entry code anyway (because it might have been updated), so it is no big deal to load Node too. In theory we could have a different CAF indirection code for each CAF, which "knows" the closure address, but it saves no time.

Note 7. The node-must-point-to-it conditions ensure that Node always points to a closure which is to be black-holed.

Note 8. For reentrant things with arity > 0, there is still a choice as to whether to jump to the fast or slow entry point, depending on how many args the thing is applied to.

Note 9. When cost-centre profiling, Node must point to anything which isn't a HNF (even if it has no fvs) , so that the cost centre can be extracted from the closure.

Note 10. Single-entry closures only need a filler if they have been black-holed. Hence the condition. (NB fvs implies not static.)

Note 11. Static updatable closures don't need a filler, because the black hole freshly allocated for them is already the standard size.

Note 12. This rule says that Node doesn't need to point to no-fv closures with arity ¿ 0. But that could be a problem because such things have an argument-satisfaction check, which needs to know where the closure is. We solve this by allocating such closures statically: this can't result in a space leak because they are HNFs already and have no fvs; nor can it result in loss of laziness.

Imported things which we know nothing about are entered as if they were updatable things with no free vars.

A.6 Stack stubbing

Black holing closures is a way to avoid space leaks, but there is another important source which is not caught thereby, namely pointers on the stack which happen to be dead. For example

```
f x = case x of ...not involving x...
```

Here `x` is passed to `f` on the stack, but is dead as soon as it is entered.

The simplest solution is:

- Just before every tail call, overwrite any stack-held ptrs which are now dead with a pointer to a special `Stub` closure. `Stub` is a static closure with no pointers inside it, so it plugs any space leak.

  The entry code for `Stub` elicits an error message, because the stack slot is supposed to be dead.

How do we know which slots are dead? Because they are bound to variables which aren't free in the continuation.

Unfortunately, this stack-stubbing takes instructions to perform. We can improve matters somewhat:

- Use dead stack slots to save volatile variables in for a `case` expression. Not only does this mean that fewer slots need to be stubbed, but it also reduces stack growth.

There is one way of avoiding the remaining stack-stubbing instructions, namely by attaching a bit-mask to return (vector) addresses pushed on the B stack, which identify the dead A-stack pointers. This is quite a bit more work, so we don't do it at present — but we count how many stubbing instructions we execute.

A.6.1 Implementation

We need to keep extra information in the code generator state:

1. We need to keep track of which A-stack slots are used for what purpose; in particular, which slots are used to store which variables.

   This info is used when saving volatile variables (at a `case` expression), to identify dead slots, and at a tail call to identify slots which must be stubbed.

   We can do this just by adding an extra component to the code generator state, carried around by the monad, and making sure we keep it up to date when we alter the environment.

2. When compiling a tail call we need to know which variables, if any are used in the continuation (if any) so that any others occupying stack slots below the tail-call `SpA` can be stubbed.

   This is easy too: just add a piece of inherited information to the monad rather like the set-filler stuff. The difference is that at a `case` expression the needed-var info goes into the case *branches* rather than into the *scrutinee*.

The scheme so far is slightly pessimistic. Consider the expresion

```
f x = let y = ... in
      case x of
        ...y...
```

The code generated for this is:

```
Allocate y
Save y on the stack
Push the continuation
Enter x
```

Now, since `x` is still live at the point we save `y`, we will allocate a new stack slot for `y`, and have to stub the `x` slot just before entering `x`. It would be better to save `y` in `x`'s slot. We can spot this as a special case, perhaps including the slightly more general case where the scrutinee is a function application.

Index