The Brisk Machine: A Simplified STG Machine

Ian Holyer and Eleni Spiliopoulou

Department of Computer Science, University of Bristol
Merchants Venturers Building, Woodland Road,
Bristol BS8 1UB, United Kingdom
{ian,spilio}@cs.bris.ac.uk

Abstract. This work presents the Brisk Machine, a machine model for the implementation of functional languages. It is especially designed to be flexible and dynamic, so that it can support a uniform and efficient implementation of multiple paradigms such as computational mobility, dynamic loading and linking, and logic programming.
The Brisk Machine is based on the STG Machine, though its model is simplified and adapted so that the various paradigms it supports can be accommodated easily without interference between them.

1 Introduction

Many different machine models have been proposed as intermediate forms in the compilation of functional languages, to act as the target of the front end of compilers and as a starting point for code generation. The resulting models tend to be complicated though, partly due to the effort to optimise them in a particular setting.

The Brisk[1] Machine has been designed as a target for Haskell compilation which is flexible enough to support a number of different run-time execution models. In particular, we aim to support concurrency, distribution and computational mobility, dynamic loading and linking, debugging tools, and logic programming in the form of the Escher [11] extensions to Haskell.

The Brisk Machine is heavily based on the STG Machine. Although the underlying model of the STG Machine is simple, design decisions made because of efficiency considerations radically restrict its flexibility and extensibility. To achieve a flexible machine model, and allow extensions to be built without interfering with the optimisations, many issues concerning the optimisation of execution are relegated to special built-in functions rather than being built into the model itself. This leaves a lean and adaptable basic execution model. Furthermore, the Brisk Machine provides a dynamic model, where functions are stored as heap nodes. This helps support distributed execution and mobility of computation where code is communicated dynamically between machines, or the development of an environment where newly compiled code is loaded dynamically into the running environment. A uniform representation of heap nodes,

[1] Brisk stands for Bristol Haskell [2] compiler

```
prog → decl1 ; ... ; decln                  n ≥ 1
decl → fun arg1 ... argn = expr             n ≥ 0
expr → let bind1 ; ... ; bindn in expr      n ≥ 1
     | case var of alt1 ; ... ; altn        n ≥ 1
     | appl
bind → var = appl
 alt → con arg1 ... argn -> expr            n ≥ 0
     | _ -> expr
appl → fun arg1 ... argn                    n ≥ 0
 fun → var
 con → var
 arg → var
```

Fig. 1. The Syntax of the Brisk Kernel Language

including those which represent functions, allows different execution strategies to coexist, e.g. compiled code, interpretive code, and code with logic programming extensions.

The compiler's intermediate language, the Brisk Kernel Language, is lower level than other functional intermediate languages. This allows the compiler to perform more low level transformations. A number of extensions and optimisations can be added easily, allowing for the efficient implementation of various computational paradigms.

There are three main sections in this paper. The first one analyses the Brisk Kernel Language, a minimal intermediate language, the second one describes the representations used in the run-time system, and the third explains the abstract machine model and its instruction set.

2 The Brisk Kernel Language

The Brisk compiler translates a source program into a minimal functional language called the *Brisk Kernel Language*, or BKL, which is described in detail in [7]. This is similar to functional intermediate languages used in other compilers, e.g. the STG language [13]. However, it is lower level than most others, allowing more work to be done in the compiler as source-to-source transformations, and thus making the run-time system simpler. Its syntax is shown in Figure 1. All mention of types, `data` statements etc. have been omitted for simplicity.

BKL provides expressions which are simple versions of *let* constructs, *case* expressions and function applications. Operationally, a *let* construct causes suspension nodes representing subexpressions to be added to the heap, a *case* expression corresponds to a simple switch since its argument is assumed to be evaluated in advance, and a function application corresponds to a tail call.

Many complications, including issues to do with evaluation and sharing, are apparently missing from the above picture. These details are hidden away in a collection of built-in functions. This simplifies both BKL and the run-time system, and allows different approaches to evaluation and sharing to be chosen at different times by replacing these functions. The built-in functions are described briefly in section 2.1 and in more detail in [7].

In order to simplify the run-time system, and to allow a uniform representation for expressions, BKL has the following features:

1. A *case* expression represents an immediate switch; it does not handle the evaluation of its argument. The argument must represent a node which is guaranteed to be already in head normal form.
2. In BKL there is no distinction between different kinds of functions such as defined functions, constructor functions and primitive functions. All functions are assumed here to be represented in a uniform way.
3. Every function application must be saturated. That is, every function has a known arity determined from its definition, and in every call to it, it is applied to the right number of arguments.
4. In every application, whether in an expression or on the right hand side of a local definition, the function must be in evaluated form. This allows a heap node to be built in which the node representing the function acts as an info node for the application.
5. Local functions are lifted out, so that local *let* definitions define simple variables, not functions. This is described further in 2.1.

Evaluation of expressions is not assumed to be implicitly triggered by the use of *case* expressions or primitive functions. Instead (points 1, 2) the evaluation of *case* expressions or arguments to strict primitive functions is expressed explicitly using calls to built-in functions such as the `strict` family, as discussed below.

The arity restriction (point 3) means that partial applications are also dealt with explicitly by the compiler during translation into BKL, using built-in functions. This frees the run-time system from checking whether the right number of arguments are provided for each function call, and from building partial applications implicitly at run-time.

The saturation of calls, and the requirement that functions are in evaluated form before being called (point 4), allows for a uniform representation of expressions and functions in the run-time system. Every expression can be represented as a node in the form of a function call where the first word points to a function node. Arity information in the function node is always available to describe the call node. Function nodes themselves can all be represented uniformly (point 2), and in fact follow the same layout as call nodes, being built from constructors as if they were data.

2.1 Built-in Functions

The main contribution of the Brisk Machine is that it makes the run-time system match the graph reduction model more closely, by freeing it from many of the usual execution details. In order to achieve this, several issues that arise during execution are handled via a collection of built-in functions. Built-in functions not only handle constructors and primitive functions, but also handle issues concerning evaluation and sharing. As a result, the run-time system becomes simple and flexible. Furthermore, several extensions regarding computational mobility as well as logic programming are treated as built-ins but we do not discuss these issues here. In the next sections the role of built-in functions in evaluation and sharing is discussed.

Partial Applications Every function in BKL has an arity indicating how many arguments it expects. Every call must be saturated, i.e. the correct number of arguments must always be supplied, according to the arity. This means that the compiler transformations must introduce explicit partial applications where necessary. For example, given that `(+)` has arity two, then the expression `(+) x` is translated into the Brisk Kernel Language as a partial application, using the built-in function `pap1of2`:

```
(+) x  -->  pap1of2 (+) x
```

The function `pap1of2` takes a function of arity two and its first argument and returns a function of arity one which expects the second argument. The function `pap1of2` is one of a family dealing with each possible number of arguments supplied and expected respectively.

The compiler may also need to deal with over-applications. For example, if `head`, which has arity one, is applied to a list `fs` of functions to extract the first, and the result applied to two more arguments x and y, then the over-application `head fs x y` would need to be translated:

```
head fs x y  -->  let f = head fs in strict100 f x y
```

Since `f` is an unevaluated expression, a member of the `strict` family of functions described below is used to evaluate it before applying it.

The arity restriction applies to higher order functions as well. For example, if the compiler determines that the function `map` has arity two, and that its first argument is a function of arity one, then a call `map (+) xs` must be transformed, e.g. to `let f x = pap1of2 (+) x in map f xs`.

The compiler has some choice in determining arities when translating a source program into BKL. For example, given a definition of the *compose* operator:

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
(.) f g = let h x = f (g x) in h
```

the compiler may choose to implement the definition as it stands, giving the operator arity two, or to transform it by adding an extra argument so that it has arity three. It may even be appropriate to compile two versions of the operator for use with different arities.

In the rest of this paper, the arity information is carried in the type, with brackets to indicate the number of arguments expected. The number of unbracketed arrows → forms the arity of the function. Thus, if the *compose* operator is compiled with arity two, the type signature above will be used, but if it is compiled with arity three, it would be written `(a -> b) -> (c -> a) -> c -> b`.

The arity restrictions we have described mean that functions are evaluated by calling and returning, as with other values. Other compilers include optimisations which avoid this, implementing a call to a possibly unevaluated function simply as a jump, but the approach we have taken has compensating advantages. It allows for greater uniformity and simplicity in the run-time system; every application can be represented in a direct way, without the need for argument stacks or for testing to see whether enough arguments are available. This argument testing is normally needed to check for partial applications; in Brisk, all partial application issues are dealt with at compile-time.

Evaluation and Strictness In BKL, it is assumed that the compiler has already dealt with the issue of forcing the evaluation of subexpressions, via built-in functions. Thus various subexpressions such as the argument to the *case* construct are assumed to be already in evaluated form.

Moreover, in any function application such as `f x y`, the function `f` is assumed to be in an evaluated form before being called, and not to be represented as a suspension node. This enables the expression `f x y` to be represented in the heap as a 3-word node; the first word points to the function node representing `f`. The function node provides layout information for the application node, as well as information on how to evaluate it, so `f` must be in evaluated form.

To express this requirement more formally we introduce the notion of *strict type* `!t` for each type `t` to indicate that a variable or a function is in head normal form. For example, in the type signature

```
f :: !a -> b -> c
```

the function `f` expects two arguments, the first of which has to be in evaluated form as indicated by the `!` symbol. As another example, given:

```
g :: !(a -> b) -> c
```

the function g takes as its argument a function which must be in evaluated form.

To force explicit evaluation of expressions, a built-in family of strict functions is provided. For example, `strict01 f x` forces evaluation of `x` before calling `f`. In the family, the 0s and 1s attached to the name `strict` refer to the function and each of its arguments; a 1 indicates that the respective item needs to be evaluated before the function is called, and 0 means that no evaluation is required. There is one member of the family for each possible combination of requirements.

The expression `strict01 f x` indicates that in the call `f x`, `f` is assumed to be in evaluated form but `x` has to be evaluated before executing the call `f x`. The type signature of `strict01` is:

```
strict01 :: !(!(!a -> b) -> a -> b)
strict01 f x = ...
```

where `!a -> b` indicates that `f` expects `x` to be in evaluated form, `!(!a -> b)` indicates that `f` must be evaluated before being passed to `strict01` and the outermost `!` indicates that `strict01` itself is assumed to be in evaluated form. An example which illustrates the use of the strict family of functions is the definition of `(+)`:

```
(+) :: !(Int -> Int -> Int)
x + y = strict011 (!+) x y

(!+) :: !(!Int -> !Int -> Int)
x !+ y = ...
```

Here `strict011` indicates that both `x` and `y` need to be evaluated before making the call `(!+) x y`. The `(!+)` function is a built-in one which assumes that its arguments are already evaluated.

To explain the operation of the `strict` functions, take `strict011 (!+) x y` as an example. Operationally `strict011` places a pointer to the original expression `strict011 (!+) x y` on a return stack with an indication that `x` is under evaluation (see section 3.2), and makes `x` the current node. When `x` is evaluated to `x'` (say), the expression is updated in place to become `strict001 (!+) x' y` with an indication that `y` is under evaluation, and `y` is made the current node. When `y` is evaluated to `y'`, the expression is popped off the stack and updated to `(!+) x' y'` which becomes the current node.

An alternative approach which allows a little more optimisation is to treat the `strict` functions as having arity one. Then `(+)` would be defined by `(+) = strict011 (!+)` and the compiler has the opportunity to generate code for `(+)` directly, inlining the call to `strict011`.

One benefit of using the `strict` family of functions to describe evaluation, and of using built-in functions generally, is that they can be replaced by alternative versions which represent different evaluation strategies (e.g. for mobility issues or in logic programming extensions to the language).

The notation for strict types that we have introduced here can be extended to deal with unboxing [14], by associating a suitable representation with each strict type. The issue of unboxing in the compiler and the Brisk Kernel Language are not discussed further here. However, the run-time system and abstract machine described later allow for unboxed representations.

Sharing One way to implement sharing is to introduce a built-in function `share` which ensures that a subexpression is evaluated only once. The call `share x` is equivalent to `x`, but when it is first accessed an update frame is put on the return stack while `x` is evaluated. When the evaluation of `x` finishes, the update frame overwrites the call `share x` with an indirection to the result. An example which illustrates this approach to sharing is the `map` function with definition:

```
map f [] = []
map f (x:xs) = f x : map f xs
```

Translating this into the Brisk Kernel Language gives:

```
map f xs = strict001 map' f xs

map' f xs = case xs of
   [] -> []
   (:) y ys ->
      let z = share z'
          z' = strict10 f y
          zs = share zs'
          zs' = map f ys
      in (:) z zs
```

The definition of `map` uses a member of the `strict` family to specify that the second argument to `map` needs to be evaluated before pattern matching can occur. In `map'`, the variable `z` is defined as a shared version of `z'` and `z'` is defined as `strict10 f y` because `f` is a dynamically passed function, which may not be in evaluated form. The variable `z'` is not mentioned anywhere else, so that the reference to it from `z` is the only reference. All other references point to the "share node" `z`. When `z` is evaluated, the built-in `share` function arranges for `z'` to be evaluated, and for the share node to be updated to become an indirection to the result. Thus all references now share the update.

This approach to sharing has a high space overhead. In the absence of sharing analysis, every node representing an unevaluated expression, i.e. every application of a function which is not a constructor, has to be assumed to need sharing, and for every such node a 2-word share node is added. An alternative optimised approach to sharing is to build it into the return mechanism. Every time the current node becomes evaluated, the return mechanism can check whether the result is different from the original expression node and, if so, overwrite the original expression node with an indirection. The overhead of this test on every return is offset by the fact that far fewer update frames need to be built – they are only needed to cover certain special cases.

Lifting Lifting is an important issue in the Brisk compiler, since there are no local function definitions in BKL; local definitions represent values not functions. In Brisk, it is possible to support either the traditional approach to lifting, or the approach taken in the Glasgow compiler; we describe both briefly here.

In the traditional approach to lifting [8], the free variables (or maximal free subexpressions) of a local function are added as extra arguments, so that the function can be lifted to the top level. The local function is then replaced by a partial application of the lifted version. For example, given a local definition of a function `f`:

```
... let f x y = ... g ... h ... in e
```

two extra arguments g and h are added and the definition is lifted out to the top level as f':

```
... let f = pap2of4 f' g h in e

f' g h x y = ... g ... h ...
```

An explicit partial application is built using pap2of4 in order to preserve the arity restrictions of BKL.

Peyton Jones & Lester [10] describe a slightly different approach to lifting. For a local function such as f, a node is built locally to represent f and this node contains references to the free variables of f. The code for f can be lifted out to the top level; it obtains the free variables from the node representing f and obtains the other arguments in the normal way.

To express this form of lifting in BKL, the language can be extended so that free variables are represented explicitly using a tuple-like notation. For example, the lifted version of f above becomes:

```
... let f =  f' (g, h) in e

f' (g,h) x y = ... g ... h ...
```

The local definition of f represents the idea that f is built directly as a function node, using f' as its code and including references to g and h. The definition of f' indicates that self-contained code should be generated which gets g and h from the function node and x and y from the call node, as shown in Figure 2.

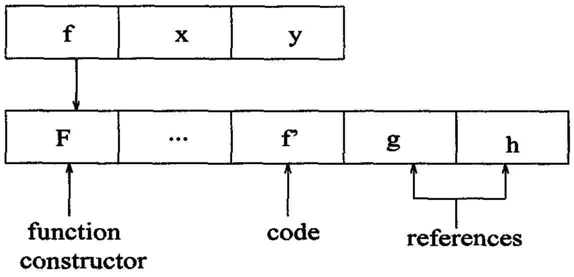

Fig. 2. A Function with References and Arguments

This approach to lifting fits in well with the way in which global functions are treated in Brisk. In order to support dynamic movement of code, a global function is represented as a node in the heap containing explicit references to the other global values which it uses. Thus a global definition:

```
f x y = ... g ... h ...
```

can also be represented in the extended notation as:

```
f = f' (g,h)

f' (g,h) x y = ... g ... h ...
```

indicating that `f` is represented as a node referencing `g` and `h`, and that self-contained code is generated for `f'`.

3 The Brisk Run-Time System

In this section we describe the way in which programs are represented in Brisk. The representation is based heavily on the one used for the STG Machine. However, it is simplified and made more uniform so that the link with conventional simple graph reduction is as clear as possible, making it easier to adapt the run-time system to alternative uses. At the same time, enough flexibility is retained to allow most of the usual optimisations to be included in some form.

The state of a program consists of a heap to hold the graph, a current node pointer to represent the subexpression which is currently being evaluated, and a return stack which describes the path from the root node down to the current node. There are no argument or update stacks.

3.1 The Heap

The heap contains a collection of nodes which holds the program graph. Every node in the heap can be thought of as a function call. A spineless representation is used; nodes vary in size, with the number of arguments being determined by the arity of the function. For example, an expression `f x y z` is represented as a 4-word node in which the first word points to a node representing `f` and the other words point to nodes representing `x`, `y` and `z`. There is a single *root node* representing the current state of the program as a whole, and all active nodes are accessible from it. Free space is obtained at the end of the heap, and a copying or compacting garbage collector is used to reclaim dead nodes.

In general, functions may have both unboxed and boxed arguments, with the unboxed ones preceding the boxed ones, and the arity of a function reflects how many of each. A node may thus in general consist of a function pointer followed by a number of raw words, followed by a number of pointer words. Constructors with unboxed arguments can be used to represent raw data. A large contiguous data structure such as array can be stored in a single node of any size; a suitable constructor node can be generated dynamically to act as the function pointer.

A node representing a function contains information about the arity of the function, the code for evaluating the function, code for other purposes such as garbage collection or debugging, and references to other nodes needed by the evaluation code, i.e. the free variables. The arity information allows the function node to be used as an *info node*, i.e. one which describes the size and layout of call nodes. To ensure that the size and layout information is always available, the function pointer in a call node must always refer to an evaluated function node, and not to a suspension which may later evaluate to a function.

This uniform representation of nodes means that any heap node can be viewed in three ways. First, a node represents an expression in the form of a function call, as in figure 3:

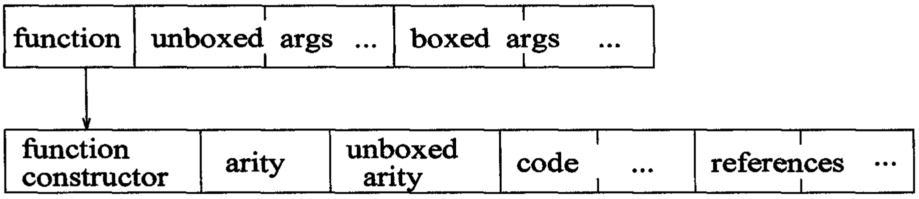

Fig. 3. A node as a function call

Second, a node can be treated as a data structure which can be manipulated, e.g. by the garbage collector or by debugging tools, in a uniform way, as in figure 4:

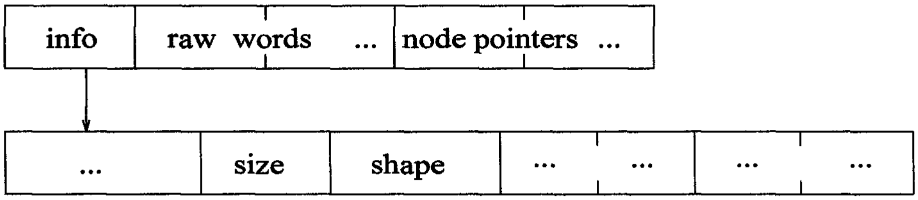

Fig. 4. A node as a uniform data structure

Third, a node can be regarded as an active object, responsible for its own execution, with methods for evaluation and other purposes available via its info pointer, as in figure 5.

Function nodes follow the same pattern. The function pointer at the beginning of a function node can be regarded as a constructor, with functions being thought of as represented using a normal data type, hidden from the programmer by an abstraction. Such a function constructor has a number of unboxed arguments representing arity and code information, and a number of boxed arguments representing references to other nodes. All functions, including global ones, are represented as heap nodes which contain references to each other. New info nodes can be created dynamically to cope with special situations, newly compiled code can be dynamically loaded, and code can be passed from one process to another, which provides greater flexibility. This contrasts with other

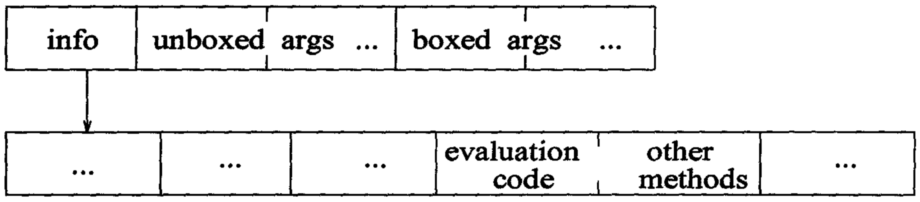

Fig. 5. A node as an object

systems [5], [13], where info structures are stored statically, which would prevent such dynamic movement of nodes.

It is becoming common, e.g. in Java [6], to use an architecture independent bytecode representation of program code, which makes it portable. This can be used to ship programs over a network and interpret them remotely. The same technique is used in Brisk. However, statically compiled code is also supported. To allow both types of code to be mixed in the same heap, the code word in a function node points to a static entry point. In the case of statically compiled code, this is precisely the evaluation code for the function. In the case of bytecode, the code word points to the static entry point of the interpretive code, and the function node contains an extra reference to a data node containing the bytecode.

3.2 The Return Stack

The *return stack* in the Brisk run-time system keeps track of the current evaluation point. Each stack entry consists of a pointer to a node in the heap representing a suspended call, and the position within that node of an argument which needs to be evaluated before execution of the call can continue, as in figure 6.

When the evaluation of the current node is complete, i.e. the current node is in head normal form, a return is performed. This involves plugging the current node into the previous node (pointed to by the top of the stack) at the specified argument position, making the previous node into the new current node, and popping the stack.

An evaluation step consists of calling the evaluation code for the current node. If the current node is already in head evaluated form, the code causes an immediate return. Otherwise, the code carries out some processing, usually ending with a tail call. Notionally, a tail call consists of building a new node in the heap and making it current. However, this leads to a large turnover of heap space, with nodes being created for every intermediate expression.

As an optimisation to reduce the number of heap nodes created, a tail call can create a temporary node; this is a node allocated in the usual way at the end of the heap, but without advancing the free space pointer. This node will then usually immediately overwrite itself as it in turn creates new heap nodes. To support this, certain nodes such as data nodes must make themselves permanent

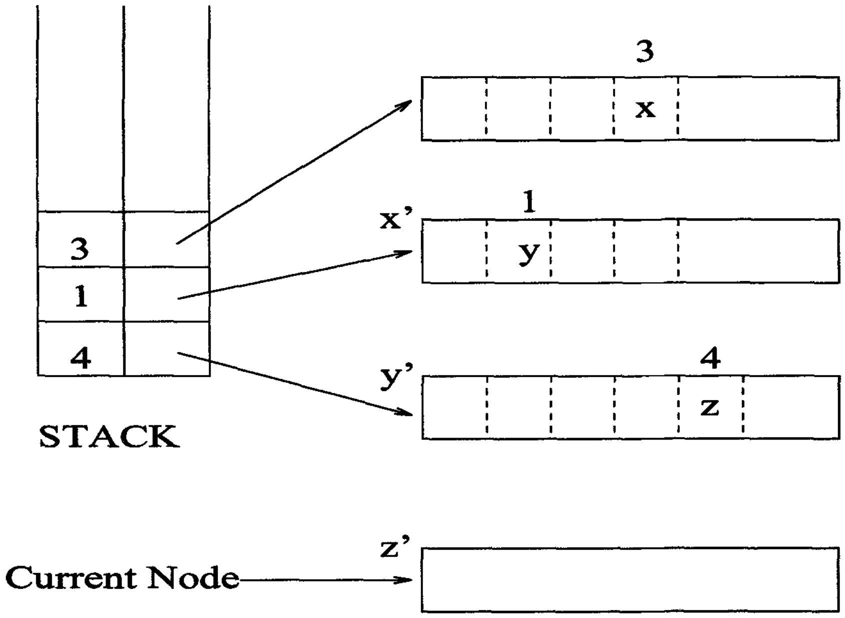

Fig. 6. The Return Stack

where necessary by advancing the free space pointer before continuing execution. This simple scheme provides many of the same advantages as more complex systems which use registers and stacks for temporary storage of intermediate expressions.

4 The Brisk Abstract Machine

Many approaches to compiling functional languages [9], [1], [16], [4], [15], use abstract machines with imperative machine instructions to provide the operational semantics of programs; see also [3], [12]. With the STG Machine [13], a different approach was taken. The STG language is a small, low level, intermediate language which is on the one hand still functional, and yet on the other can be given a direct operational state transition semantics to guide its translation into C or machine code.

The Brisk compiler uses the intermediate language BKL which, like STG, is a low level functional language. A direct semantics could similarly be given for it. However, the computational mobility issues, which Brisk addresses, mean that BKL needs to be compiled into interpretive bytecode in some circumstances, and C or machine code in others.

We have chosen to present the operational semantics of programs using a conventional abstract machine similar to the original G-machine, so that the machine instructions correspond very closely to the bytecode instructions produced via the interpretive compiling route. They also guide direct translation into C or machine code and provide a link with the design of the run-time system.

The abstract machine described here is called the Brisk Abstract Machine, or BAM for short. First, we show how to compile the Brisk Kernel Language into BAM code, and then we give the semantics of the individual instructions of the BAM code in terms of the run-time system described above.

4.1 Compiling into BAM Code

The state of the Brisk Abstract Machine consists of a *frame* of local variables in addition to the return stack and the heap. The frame contains the local variables used during execution. It is completely local to each individual function call, and starts out empty. It is extended in a stack-like manner as needed. This stack-like use is very similar to the stack in the original G-machine, and it allows the BAM instructions to have at most one argument each, often a small integer, which leads to a compact bytecode.

On the other hand, the BAM frame is not used as an argument stack or a return stack, items don't have to be popped off so frequently, and it does not have to be left empty at the end of a function call. This allows us to use it as an array as well a stack, indexed from its base. Initial instructions in the code for each function indicate the maximum size of frame needed, and the maximum amount of heap space to be used, but we omit those here. The function $\mathcal{F}$ compiles code for a function definition:

$$\begin{array}{l}
\mathcal{F}\ [\![f\,(g_1,\ldots,g_n)\ x_1\ldots x_m = e]\!] = \\
\quad [\texttt{GetRefs}\ n, \texttt{GetArgs}\ m] + \! + \\
\quad \mathcal{C}\ [\![e]\!]\ [g_1 \mapsto 0,\ldots,g_n \mapsto n-1, x_1 \mapsto n,\ldots,x_m \mapsto n+m-1]\ (n+m) \\
\quad\ + \! + [\texttt{Enter}]
\end{array}$$

A function f with references to free variables g_i and arguments x_i is compiled into code which first loads the references and arguments into the frame, then compiles code for the right hand side e which creates a pointer to a node representing e, and then ends by entering the code for this result node.

The function $\mathcal{C}$ generates code for an expression. A call $\mathcal{C}\ [\![e]\!]\ \rho\ n$ generates code for e, where ρ is an environment which specifies where the variables appear in the frame, and n is the current size of the frame.

The code generated for a variable or a constant is:

$$\begin{array}{l}
\mathcal{C}\ [\![x]\!]\ \rho\ n = [\texttt{GetLocal}\ (\rho\, x)] \\
\mathcal{C}\ [\![c]\!]\ \rho\ n = [\texttt{GetConst}\ c]
\end{array}$$

The GetLocal instruction gets a variable from its position in the frame and pushes it onto the end of the frame. The GetConst instruction pushes a one-word unboxed value onto the end of the frame. The frame is thus untyped; it contains both pointers and raw values. This does not cause any problems since the frame only lasts for a single function call so, for example, the garbage collector need not be concerned with it.

The code generated for an application is:

$$\boxed{\begin{array}{l} \mathcal{C}\ [\![f\ x_1 \ldots x_n]\!]\ \rho\ m = \\ \quad \mathcal{C}\ [\![x_n]\!]\ \rho\ m \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathcal{C}\ [\![x_1]\!]\ \rho\ m \mathbin{+\!\!+} \\ \quad \mathcal{C}\ [\![f]\!]\ \rho\ m \mathbin{+\!\!+} \\ \quad [\texttt{Mkap}\ (n+1)] \end{array}}$$

The code just gathers together the function and its arguments, and then creates a new application node in the heap from them.

The code generated for a `let` construct is:

$$\boxed{\begin{array}{l} \mathcal{C}\ [\![\texttt{let}\ y_1 = ap_1; \ldots ; y_n = ap_n\ \texttt{in}\ e]\!]\ \rho\ m = \\ \quad [\texttt{Alloc}\ k_1, \ldots, \texttt{Alloc}\ k_n] \\ \quad \mathcal{D}\ [\![y_1 = ap_1]\!]\ \rho'\ m' \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathcal{D}\ [\![y_n = ap_n]\!]\ \rho'\ m' \mathbin{+\!\!+} \\ \quad \mathcal{C}\ [\![e]\!]\ \rho'\ m' \\ \quad \text{where} \\ \qquad\quad k_i \text{ is the size of application } ap_i \\ \qquad\quad \rho' \ = \rho\ [y_1 \mapsto m, \ldots, y_n \mapsto m+n-1] \\ \qquad\quad m' = m + n \end{array}}$$

The code generated here assumes that the `let` construct is recursive. As usual, if the `let` construct is known to be non-recursive, simpler code can be generated, but we don't show that here. First, space is allocated on the heap for each node to be created, and the variables y_i are made to point to these blank nodes. Then each node is filled in in turn using the $\mathcal{D}$ function:

$$\boxed{\begin{array}{l} \mathcal{D}\ [\![y = f\ x_1 \ldots x_n]\!]\ \rho\ m = \\ \quad \mathcal{C}\ [\![x_n]\!]\ \rho\ m \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathcal{C}\ [\![x_1]\!]\ \rho\ m \mathbin{+\!\!+} \\ \quad \mathcal{C}\ [\![f]\!]\ \rho\ m \mathbin{+\!\!+} \mathcal{C}\ [\![y]\!]\ \rho\ m \mathbin{+\!\!+} \\ \quad [\texttt{Fill}\ (n+1)] \end{array}}$$

This generates code to fill in a blank node by gathering the function and its arguments, together with the pointer to the node to be filled, and then using the `Fill` instruction to fill in the node.

The code generated for a `case` construct is:

$$\boxed{\begin{array}{l} \mathcal{C}\ [\![\texttt{case}\ v\ \texttt{of}\ alt_1 \ldots alt_n]\!]\ \rho\ m = \\ \quad [\texttt{Table}\ k]\ \mathbin{+\!\!+} \\ \quad \mathcal{A}\ [alt_1]\ \rho\ m\ \mathbin{+\!\!+} \ldots \mathbin{+\!\!+}\ \mathcal{A}\ [alt_n]\ \rho\ m\ \mathbin{+\!\!+} \\ \quad [\texttt{Switch}\ (\rho\, v)] \\ \quad \text{where } k \text{ is the number of constructors in the type of } v \end{array}}$$

The code first allocates a jump table with one entry for every constructor in the type of v. This is followed by code for each alternative, generated using $\mathcal{A}$. An interpreter for the code scans the alternatives, filling in the jump table. Once the jump table is complete, the `case` variable is used by the `Switch` instruction to jump to the appropriate case.

The code generated for an alternative is:

```
A [[C x_1 ... x_n-> body]] ρ m =
    [Case i, Skip b, Split n] ++
    C [[body]] ρ' m' ++
    [Enter]
    where
        i is the sequence number of constructor C
        b is the amount of code generated for the body
        $\rho' = \rho\ [x_1 \mapsto m, \ldots, x_n \mapsto m + n - 1]$
        $m' = m + n$
A [[_ -> body]] ρ m =
    [Default, Skip b] ++
    C [[body]] ρ m ++
    [Enter]
```

The `Case` instruction is used to fill in an entry in the previously allocated jump table. The entry points to the code for the body of the alternative, just after the `Skip` instruction. The `Skip` instruction skips past the code for the alternative (including the `Split` and `Enter` instructions). The `Split` instruction loads the constructor arguments into the frame, and the environment is extended with their names. A default `case` alternative causes all the unfilled entries in the jump table to be filled in.

4.2 The BAM Instructions

Here, we give transition rules which describe the action of each of the instructions on the run-time system state. The state consists of the current sequence of instructions i, the frame f, the return stack s, and the heap h. When it is needed, the current node will be indicated in the heap by the name *cn*.

The frame will be represented in a stack-like way as $f = x : y : z : \ldots$, but indexing of the form $f!i$ will also be used, with the index being relative to the base of the stack.

The `GetLocal` n instruction pushes a copy of the n'th frame item onto the end of the frame:

`GetLocal` n	:	i	f	s	h
		i	$f!n : f$	s	h

The `GetConst` c instruction pushes a constant onto the end of the frame:

`GetConst` c	:	i	f	s	h
		i	$c : f$	s	h

The `GetRefs` n instruction loads n references into the frame, from the function node mentioned in the current node:

$$\boxed{\begin{array}{llllll}
\texttt{GetRefs}\ n : i & & f & s & h\left[\begin{array}{l} cn \mapsto <g, \ldots> \\ g \mapsto <\ldots, g_1, \ldots, g_n> \end{array}\right] \\
i & g_n : \ldots : g_1 : f & & s & h
\end{array}}$$

The `GetArgs` n instruction extracts n arguments from the current node into the frame:

$$\boxed{\begin{array}{lllll}
\texttt{GetArgs}\ n : i & & f & s & h[cn \mapsto <g, x_1, \ldots, x_n>] \\
i & x_n : \ldots : x_1 : f & & s & h
\end{array}}$$

The `Mkap` n instruction assumes that the top of the frame contains a function and its $n-1$ arguments, from which a node of size n is to be built:

$$\boxed{\begin{array}{llll}
\texttt{Mkap}\ n : i & g : x_1 : \ldots : x_{n-1} : f & s & h \\
i & p : f & s & h[p \mapsto <g, x_1, \ldots, x_{n-1}>]
\end{array}}$$

The `Alloc` n instruction allocates space for a node of size n on the heap, with uninitialised contents:

$$\boxed{\begin{array}{llll}
\texttt{Alloc}\ n : i & f & s & h \\
i & p : f & s & h[p \mapsto <?_1, \ldots, ?_n>]
\end{array}}$$

The `Fill` n instruction fills in a previously allocated node, given its pointer and contents:

$$\boxed{\begin{array}{llll}
\texttt{Fill}\ n : i & p : g : x_1 : \ldots : x_{n-1} : f & s & h[p \mapsto <?_1, \ldots, ?_n>] \\
i & f & s & h[p \mapsto <g, x_1, \ldots, x_{n-1}>]
\end{array}}$$

To implement the **case** construct, a jump table is needed. We temporarily extend the state, adding the table as an extra component t. The `Table` n instruction allocates an uninitialised table of size n:

$$\boxed{\begin{array}{lllll}
\texttt{Table}\ n : i & f & s & h & \\
i & f & s & h & t \mapsto <?_1, \ldots, ?_n>
\end{array}}$$

Each `Case` k instruction causes the k'th table entry to be filled in with the position i in the instruction sequence, just after the subsequent `Skip` instruction:

$$\boxed{\begin{array}{llllll}
\texttt{Case}\ k : \texttt{Skip}\ b : i & f & s & h & t \mapsto <\ldots, ?_k, \ldots> \\
\texttt{Skip}\ b : i & f & s & h & t \mapsto <\ldots, i, \ldots>
\end{array}}$$

A `Default` instruction causes any unfilled entries in the table to be filled in with the relevant position in the instruction sequence:

$$\boxed{\begin{array}{llllll}
\texttt{Default} : \texttt{Skip}\ b : i & f & s & h & t \mapsto <\ldots, ?, \ldots, ?, \ldots> \\
\texttt{Skip}\ b : i & f & s & h & t \mapsto <\ldots, i, \ldots, i, \ldots>
\end{array}}$$

Once the table has been filled in, the `Switch` instruction gets the **case** variable v from the frame and causes a jump to the relevant alternative. The sequence number k of the constructor C used to build v is used to index the jump table. The sequence number of C can be recorded in its heap node. Execution continues with position i_k in the instruction sequence, where i_k is the k'th table entry. The table is not needed any more:

$$\boxed{\begin{array}{lllll} \texttt{Switch}\,n : i & f & s & h[f!n \mapsto <C, \ldots>] & t \mapsto <\ldots, i_k, \ldots> \\ \phantom{\texttt{Switch}\,n :}\; i_k & f & s & h & \end{array}}$$

While the jump table is being built, the `Skip` instruction is used to skip past the code for the alternatives. The instruction `Skip` b simply skips over b instructions (or b bytes if the instructions are converted into bytecode):

$$\boxed{\begin{array}{llll} \texttt{Skip}\,b : i & f & s & h \\ \quad \mathit{drop}\;b\;i & f & s & h \end{array}}$$

At the beginning of the code for a `case` alternative, the instruction `Split` n is used to extract the fields from the `case` expression, which is still at the top of the frame at this moment:

$$\boxed{\begin{array}{lrll} \texttt{Split}\,n : i & v : f & s & h[v \mapsto <C, x_1, \ldots, x_n>] \\ \phantom{\texttt{Split}\,n :}\; i & x_n : \ldots : x_1 : f & s & h \end{array}}$$

At the end of the code for a function, or at the end of an alternative, the `Enter` instruction transfers control to a new function. The node on the top of the frame becomes the current node, a new instruction sequence is extracted from its function node, and execution of the new code begins with an empty frame:

$$\boxed{\begin{array}{llll} \texttt{Enter} : i & p : f & s & h \begin{bmatrix} p \mapsto <g, \ldots> \\ g \mapsto <\ldots, fi, \ldots> \end{bmatrix} \\ \qquad\; fi & [] & s & h\;[cn = p] \end{array}}$$

Instructions to manipulate the return stack are handled by special functions such as constructors and the *strict* family, rather than being produced by the compiler. As an example, the code for a constructor is [`Return`, `Enter`] where the `Return` instruction puts the current node into the appropriate argument position of the previous node and pops the return stack:

$$\boxed{\begin{array}{llll} \texttt{Return} : i & [] & <k, p>: s & h \begin{bmatrix} p \mapsto <\ldots, x_k, \ldots> \\ cn \mapsto <\ldots> \end{bmatrix} \\ \phantom{\texttt{Return} :}\; i & [p] & s & h\;[p \mapsto <\ldots, cn, \ldots>] \end{array}}$$

5 Conclusions and Further Work

The Brisk Machine been designed so that it provides a flexible model which allows for general experimentation with different execution models. Dynamic loading of newly compiled modules are supported by the ability to load functions as newly created nodes in the heap, and to mix compiled and interpretive bytecode on a function-by-function basis.

The problems of supporting debugging tools are eased by the uniform representation of nodes, the ease with which the layout of nodes can be changed dynamically, and the close relationship of the run-time system to simple graph reduction, without complex stacks and registers.

Concurrency is implemented, as usual, by providing multiple lightweight threads of execution, with time slicing, in the run-time system. The concurrency issues are easily separated from other compilation and execution issues.

Computational mobility in a distributed setting is facilitated by the direct and uniform representation of all expressions, including functions, in the heap. This allows functions to be shipped between processors. In general, communication between processors is implemented in such a way that it is equivalent to having a single distributed heap; this is eased by the ability to insert built-in functions to hide the communication details. A distributed program is thus kept equivalent to its non-distributed counterpart, which helps to make distribution orthogonal to other issues.

The Brisk model is being used by another Bristol group to support work on logic programming extensions. This involves adding run-time variables (in the logic programming sense), changing the details of execution order, and changing the approach to such things as strictness, sharing and updating, largely by replacing built-in functions by alternatives.

Although experiments on all these fronts have been carried out, more work is needed to provide the Brisk system with a convenient development environment, and to investigate its efficiency, particularly in comparison to the STG Machine. Almost all of the optimisations which can be carried out before reaching the STG language (or from STG to STG) are also applicable in the Brisk setting; however, they have not been implemented in Brisk, making direct comparisons difficult at present. In particular, we have yet to investigate the implementation of unboxing. Optimisations which come after reaching the STG language, i.e. during code generation are more difficult to incorporate since they involve stacks, registers, return conventions etc. Nevertheless, some of these have been investigated to see if similar optimisations are possible in Brisk. For example, in the STG approach, intermediate expressions are stored implicitly on stacks, which considerably reduces heap usage. In Brisk, intermediate expressions are stored as heap nodes. Nevertheless, by treating the last node in the heap as a temporary node which can usually be overwritten by a node representing the next intermediate expression, most of the gain of using stacks can be obtained without complicating Brisk's simple approach to representation. Another interesting question is to compare the two approaches to partial applications; on the one hand Brisk avoids the overheads of run-time argument testing and possible subsequent building of partial application nodes. On the other, unevaluated functions have to be evaluated before being called. The tradeoffs are not clear.

Although more work needs to be done, the conceptual simplicity of the Brisk Machine has already proven to be a great advantage in adding experimental features to the system.

References

[1] Lennart Augustsson. *Compiling lazy functional languages, part II.* PhD thesis, Chalmers University, 1987.

[2] Lennart Augustsson, Brian Boutel, Warren Burton, Joseph Fasel, Andrew D. Gordon, John Hughes, Paul Hudak, Thomas Johnson, Mark Jones, Erik Meijer, Simon L. Peyton Jones, Alastair Reid, and Philip Wadler. Haskell 1.4, A Non-strict, Purely Functional Language. Technical report, Yale University, 1997.

[3] Geoffrey Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* Pitman, 1991.

[4] Geoffrey Burn and Simon L. Peyton Jones. The Spineless G-machine. In *Conference on Lisp and Functional programming*, 1988.

[5] Manuel M.T. Chakravarty and Hendrik C.R. Lock. The JUMP-machine: a Generic Basis for the Integration of Declarative Paradigms. In *Post-ICLP Workshop*, 1994.

[6] J. Gosling, J. B. Joy, and G. Steele. *The Java Language Specification.* Addisson Wesley, 1996.

[7] Ian Holyer. The Brisk Kernel Language. Technical report, University of Bristol, Department of Computer Science, 1997.

[8] Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In Jouannaud, editor, *Functional Programming and Computer Architecture*, volume 201 of *LNCS*, pages 190–205. Springer Verlag, 1985.

[9] Thomas Johnsson. *Compiling lazy functional languages.* PhD thesis, Chalmers University, 1987.

[10] David Lester and Simon L. Peyton Jones. A modular fully-lazy lambda lifter in Haskell. *Software Practice and Experience*, 21(5), 1991.

[11] J. W. Lloyd. Declarative programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, June 1995.

[12] Simon L. Peyton Jones. *Implementing Functional Languages.* Prentince Hall, 1992.

[13] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, July 1992.

[14] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming and Computer Architecture*, Sept 1991.

[15] Simon L. Peyton Jones and John Salkild. The Spineless Tagless G-machine. In MacQueen, editor, *Functional Programming and Computer Architecture.* Addisson Wesley, 1989.

[16] Stuart Wray and J. Fairbairn. Tim - a simple lazy abstract machine to execute supercombinators. In *Functional Programming and Computer Architecture*, volume 274 of *LNCS*, Portland, Oregon, 1987. Springer Verlag.