The Brisk Machine: the Next Step in the Execution of Functional Languages

Eleni Spiliopoulou and Ian Holyer

7^{th} July 2000

1 The Brisk Machine

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. Their investment has mainly been to increase sequential performance and to provide industrial strength compilers, aiming at making functional languages industrially competitive and generalising the use of functional languages outside of the academia.

The Brisk machine [3, 1] has been designed to offer a machine model, which is simple and flexible enough to support dynamic loading and linking as well as a number of different run-time execution models, such as concurrency, distribution and computational mobility, debugging tools, logic programming etc. To achieve a simple machine model and allow extensions to be built without interfering with the optimisations, many issues concerning 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, which matches the graph reduction model. As a consequence, the compiler's intermediate language, the *Brisk Kernel Language* (`BKL`), is lower level than other functional intermediate languages [2]. This simplicity of the `BKL` makes the machine model *extensible* and *dynamic*. *Extensible* because it allows alternative approaches to evaluation and sharing to be chosen. As a result, extensions can be added without any interference between them. *Dynamic* because heap nodes have a uniform representation in the run-time system, which is the key issue to enable dynamic loading and computational mobility. This means that functions are also represented as heap nodes with the same layout as call nodes, being built from constructors as if they were data. As a result, new nodes can be created dynamically to cope with special situations, functions can be loaded dynamically as newly created nodes in the heap, newly compiled code can be dynamically loaded in the heap, code can be passed from one process to another. This uniform representation of expressions has been achieved by building compile-time partial applications and by forcing both subexpressions and functions in any function application to be in evaluated form.

2 Built-in Functions

To make the Brisk machine execution model match the graph reduction model, several issues that arise during execution such as constructors and primitive functions as well as issues to do with evaluation and sharing are handled via built-in functions. Furthermore, several extensions have been added as built-in functions without any interference between them.

2.1 Partial Applications

In `BKL` every function has an arity indicating how many arguments it expects. For example, given that `(+)` has arity two, the expression `(+) x` has arity one. In the Brisk machine, every call must be saturated, i.e. the correct number of arguments must always be supplied. This implies that the compiler transformations must introduce explicit partial applications at compile-time, when necessary. Given the function `(+)`, the expression `(+) x` is translated into the Brisk Kernel Language using the built-in function `(+) x -> pap1of2 (+) x`.

2.2 Evaluation and Strictness

The uniform representation of expressions and functions in the run-time system has been achieved by the fact that the evaluation of subexpressions is forced by built-in functions introduced by the compiler.

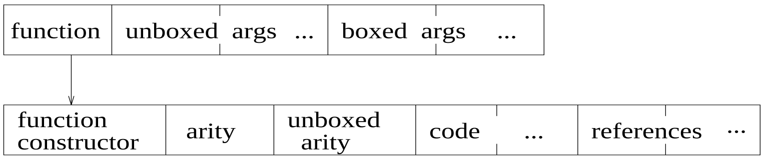

Figure 1: A Node as a Function Call

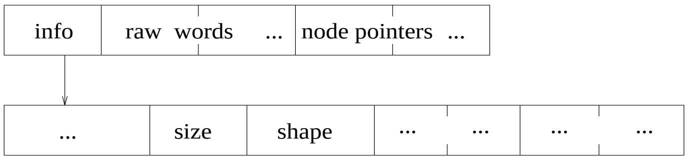

Figure 2: A Node as a Uniform Data Structure

Various subexpressions such as the argument to the *case* construct or the right hand side of a local definition 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 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 both layout information for the application node and information on how to evaluate it, so `f` 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`. There is one member of the family for every possible number of requirements. An example, which illustrates the strict family of functions, is the definition of `(+)`.

```
x + y = strict011 (!+) x y
x (!+) y = ...
```

Here `(!+)` indicates a built-in function which assumes that the arguments are already evaluated. Operationally, `strict011` places a pointer to the original expression `strict011 (!+) x y` on a return stack with an indication that `x` is under evaluation and makes `x` the current node. When `x` is evaluated to `x'` (say), the expression is updated in place and becomes `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.

3 The Run-Time System

The Brisk run-time system is especially designed to support the properties of the Brisk machine. One of its main characteristics is its ability to load functions as newly created nodes in the heap and to mix compiled and interpreted code in a function-by-function basis.

3.1 The Heap and Return Stack

In the Brisk machine functions are represented the same way as call nodes. A node representing a function contains information about the arity of the function, the code 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.

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.
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 1. Second, a node can be treated as a data structure which can be manipulated in a uniform way, as in Figure 2. Third, a node can be

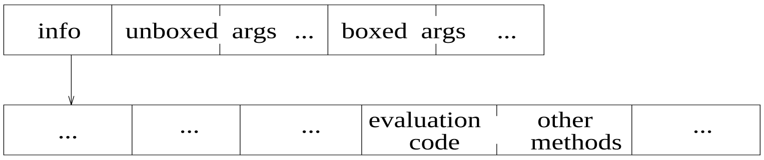

Figure 3: A Node as an Object

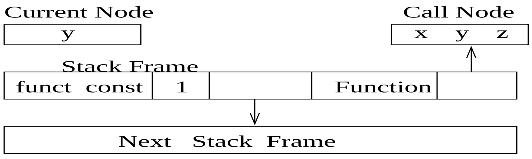

Figure 4: The Return Stack

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 3.

The *return stack* keeps track of the current evaluation point. Each entry point points to a node in the heap representing a suspended call and also keeps track of the position within that node of an arguments which is under evaluation before execution of the call can continue. Each stack node consists of a function constructor, an unboxed argument which specifies the position within that node of an argument which needs to be evaluated before execution of the call can continue, a pointer to the next stack frame, a continuation function to replace the one in the original node, once evaluation of all the fields in the pattern is complete and the suspended call.

3.2 Other Run-Time System Issues

Special techniques have been employed to facilitate dynamic loading, the main one being that the evaluation code is allowed to access functions in the run-time system only indirectly, see also [3]. This can be done by two techniques. First, through special macros defined in the header nodules of the run-time system that transfer the control of execution from the execution code to the run-time system. Second, by including pointers to special procedures as registers at the machine state. These can be accessed by the execution code as registers. Another design issue that facilitates dynamic loading is that the machine state registers are not passed as global variables to procedures. Instead, they are kept in a structure, which is passed to the execution code as a pointer argument.

4 Conclusions

The Brisk machine has been added several extensions such as dynamic loading, concurrency and distribution [3], debugging and logic programming. Several optimisations such as unboxed arguments are orthogonal to the design and are now being added.

References

[1] Ian Holyer and Eleni Spiliopoulou. The Brisk Machine: A Simplified STG Machine. In Chris Clack, Tony Davie, and Kevin Hammond, editors, *9th International Workshop on the Implementation of Functional Languages*, volume 1467 of *LNCS*, pages 20–38, Saint-Andrews, Scotland, September 1997. Springer Verlag.

[2] 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.

[3] Eleni Spiliopoulou. *Concurrent and Distributed Functional Systems*. PhD thesis, Bristol University, 1999.