J. Functional Programming **8** (1): 61–81, January 1998. Printed in the United Kingdom

The dynamic compilation of lazy functional programs

DAVID WAKELING

Department of Computer Science, University of Exeter, Exeter, EX4 4PT, UK
(*web:* `http://www.dcs.exeter.ac.uk/~david`)

Abstract

Lazy functional languages seem to be unsuitable for programming embedded computers because their implementations require so much memory for program code and graph. In this paper we describe a new abstract machine for the implementation of lazy functional languages on embedded computers called the X-machine. The X-machine has been designed so that program code can be stored compactly as byte code, yet run quickly using dynamic compilation. Our first results are promising – programs typically require only 33% of the code space of an ordinary implementation, but run at 75% of the speed. Future work needs to concentrate on reducing the size of statically-allocated data and the run-time system, and on developing a more detailed understanding of throw-away compilation.

Capsule Review

This paper takes an important first step toward making functional programming languages available and attractive in settings where excessive storage requirements could be problematic.

Despite advantages including ease of program construction, analysis, and verification, lazy functional languages have not been an option for those who create software for embedded systems. One reason is that many implementations optimize reduction rules by compiling them into native code, often at the expense of drastically increasing the code size.

The author proposes a new intermediate representation for lazy functional languages, called the X-machine. Dynamic compilation is then applied to X-machine code to reduce the space needed to represent the optimized program, with a modest increase in overhead due to translation from the compressed representation. Experiments reported in the paper show that code size is indeed substantially reduced. However, code size is only one factor in considering a program's overall storage requirements: functional programs can reference a run-time heap and static data.

From the experiments, the paper concludes that future work must address these factors to bring a functional program's storage requirements truly within reach of embedded systems.

1 Introduction

Today computers lurk everywhere, heavily disguised as digital diaries, mobile telephones, video cameras and countless other electronic devices. Whilst it is most unlikely that any of these embedded computers was programmed with a functional

language, two characteristics of the market lead us to believe that functional languages may one day be used to program consumer electronics products. First, it is vital that these products should be delivered *on time* – the pace of the market is such that a delay of six months can make the difference between high sales with big profits, and low sales with big losses. Secondly, it is vital that these products should be delivered *to specification* – any careless oversight can result in the expense and embarrassment of recalling many thousands of units. There are strong arguments that functional programs can be written more quickly and proved correct more easily than imperative ones (Turner, 1982; Hudak and Jones, 1994).

Unfortunately, the *lazy* functional programs with which we work often require large amounts of memory. They consist of an expression to be evaluated, which is represented by a *graph*, and some functions, which are represented by *reduction rules* for the graph. A program is executed by reducing the graph to normal form, printing the result as it becomes available. To make programs run fast, many implementations *compile* the reduction rules into machine code, even though this may produce a lot of code. Suppose, for example, that we are using the popular Chalmers HBC/LML compiler (Augustsson and Johnsson, 1989) and an SGI 02 computer with an R5000 processor. At one extreme, consider the 120-line calendar program from Bird and Wadler (1988). This program compiles into 288 kbytes of code (of which 34 kbytes are for statically-allocated data and 70 kbytes are for run-time support); only an additional 30 kbytes are needed for the graph. At the other extreme, consider the 25,000-line HBC/LML compiler itself. This program compiles into 3 800 kbytes of code (of which 708 kbytes are for statically-allocated data and 70 kbytes are for run-time support); an additional 4 000 kbytes for the graph are enough to compile most modules. These figures show that code size can be a real problem, and they are quite unacceptable in an industry where economic and physical constraints severely limit the amount of memory available. Think of an electric shaver which may contain one or two kilobytes, or a television which may contain a few hundred.

In recent years, we have shown that the program graph can often be made much smaller with careful reprogramming informed by the results of *heap profiling*. In this paper we show that the program code can also be made much smaller, without running much more slowly, by using *dynamic compilation*. This work leaves the run-time system (a fixed 70 kbytes) and statically-allocated data (typically, 25% of the remaining code size) untouched. Further work needs to be done to reduce the size of both of these. Nonetheless, our results give us real hope that functional programs can indeed be used to program embedded computers.

This paper is organised as follows. Section 2 outlines two traditional interpretive techniques that have been used to implement functional languages on small personal computers. Section 3 explains why interpreters cannot exploit some important features of modern computer architecture in the way that compilers can. Section 4 describes a new abstract machine designed so that programs can be stored with the compactness of an interpreter, yet run with the efficiency of a compiler. Section 5 gives details of how this abstract machine's programs are represented and executed. Section 6 notes some difficulties in incorporating the abstract machine

into a test-bed implementation. Section 7 provides some performance figures for this implementation. Section 8 considers some possible future work, section 9 some closely related work, and section 10 concludes.

A certain familiarity with the implementation of functional languages and the basics of computer architecture is assumed throughout this paper. Those without such familiarity may occasionally need to consult the textbooks by Peyton Jones (1987) and Patterson and Hennessy (1994).

2 Interpretation

One way to program embedded systems using a functional language is to further cut down an implementation designed for a small personal computer. Such implementations save code space by using either *byte code* or *threaded code interpretation* (Röjemo, 1995; Leroy, 1990). This section briefly reviews these two techniques.

2.1 Byte code interpretation

A byte code interpreter works as follows. At compile-time, the program is translated into a simple abstract machine code, conventionally called *byte code* because each instruction is represented by a one-byte opcode followed by further bytes for any operands. At run-time, an interpreter executes the byte code instructions. This works particularly well when the abstract machine is a stack machine. Most instructions load their arguments from the stack and store their result there. They can be represented by a single opcode byte, giving a byte code with a high instruction density* which is easy to interpret. However, for simple stack operations there is a large *interpretive overhead* – around 30% of the total execution time can be spent dispatching on the opcode (Röjemo, 1995).

2.2 Threaded code interpretation

The interpretive overhead can be reduced considerably by using a threaded code interpreter. The opcode byte of each instruction is replaced by the address of the code that implements the instruction. An instruction is executed by making a call to the opcode. In this case, dispatching on the opcode takes place at compile-time rather than run-time and the interpretive overhead can be halved (Leroy, 1990). But the instruction density is also lower because the opcode is represented by an address (typically, 4 bytes) instead of a byte.

3 Interpreters and modern computer architecture

A serious drawback of both byte code and threaded code interpreters is that they cannot exploit some important features of modern computer architecture.

* A machine that uses a small number of bits to encode a program is said to have a high *instruction density* (Hennessy and Patterson, 1990)

- Modern computers are *general purpose register* machines: they provide plenty of registers (typically 32 or more) that can be accessed more quickly than main memory, and good performance depends on making full use of them. Unfortunately, though, an interpreter deals with one abstract machine instruction at a time, and it is difficult for it to make serious use of machine registers.
- Modern computers have *instruction pipelines*: while one instruction is being executed, the next is being decoded, and the next-but-one is being fetched. Unfortunately, though, an interpreter makes a large number of branches to addresses which are decided very late. These branches stall the pipeline and reduce its effectiveness.
- Modern computers *cache* instructions: they exploit locality of reference by copying blocks of main memory into cache memory which can be accessed more quickly. Interpretive loops usually contain a small block of code to implement each abstract machine instruction. Unfortunately, though, these blocks do not necessarily enjoy good locality of reference.

Even older processors now have all these features. An implementation that exploits modern computer architecture better than an interpreter may make it possible to use a processor with a reduced clock rate, or from the previous generation. This in turn may substantially improve the competitiveness of a product.

4 An abstract machine

Ideally, we would like a way to store programs that gives a high instruction density, and a way to execute them that makes good use of the machine's registers, pipeline and cache. With this as our goal, we have developed a new abstract machine called the X-machine. It is the result of some experiments with the Chalmers LML/HBC compiler (Augustsson and Johnsson, 1989), and is best described in that context. The Chalmers compiler generates code first for an abstract stack machine called the G-machine, and then for a less-abstract register machine called the M-machine. As we shall see, the X-machine lies somewhere between these two.

4.1 The G-machine

The G-machine provides instructions for constructing and manipulating graphs. To give some idea of its operation, here is the G-machine code for the function `dup` defined as `dup x = (x,x)`

```
dup: PUSH 0
     PUSH 1
     CPAIR 0
     UPDATE 2
     POP 1
     RET
```

Fig. 1 shows how this code is used to perform graph reduction. Execution begins with

https://doi.org/10.1017/S0956796897002955 Published online by Cambridge University Press

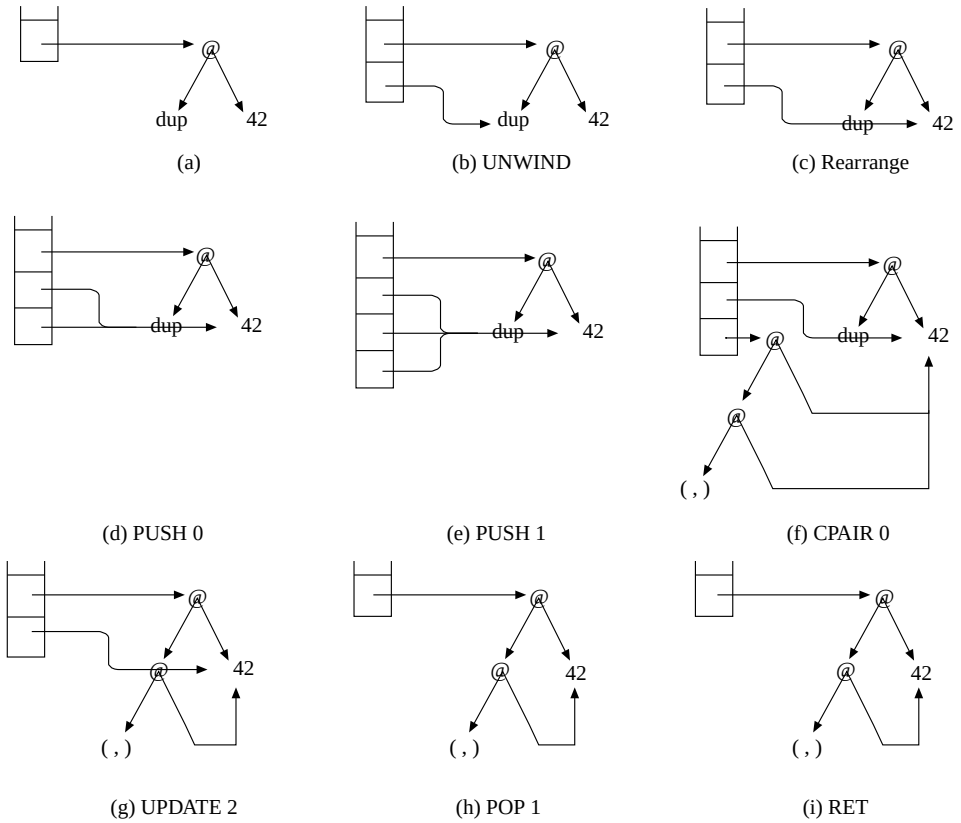

Fig. 1. Graph reduction of `dup 42`.

a pointer to an initial graph on the pointer stack (Fig. 1(a)). The 'spine' of the graph is then *unwound* by pushing pointers to the vertebrae (Fig. 1(b)). When the head of the spine has been reached the stack is rearranged to provide efficient access to the argument (Fig. 1(c)). The machine code instructions for the `dup` function shown above are then executed in sequence to perform one graph reduction (Fig. 1(d)–(i)).

Among the G-machine instructions are PUSH *n*, which copies a pointer on the pointer stack; CPAIR *m*, which takes two pointers from the pointer stack and puts back one to a pair node tagged with *m*; UPDATE *n*, which overwrites one graph node with another; POP *n*, which removes *n* pointers from the pointer stack; and RET, which returns from a function call and restores the previous abstract machine state from the dump stack (not shown in Fig. 1).

G-instructions tend to be complex, and G-code converts to a byte code with a high instruction density. This instruction density can be further increased by inventing new instructions to replace common sequences. For example, the last four instructions for `dup` are of the form

$$\text{CPAIR } m;\quad \text{UPDATE } n;\quad \text{POP } (n-1);\quad \text{RET}$$

https://doi.org/10.1017/S0956796897002955 Published online by Cambridge University Press

These could be replaced by

CPAIR_RET *m n*

This instruction is an example of a *superoperator* as described in Proebsting (1995), although the ones in the Chalmers LML/HBC compiler were derived manually rather than automatically.

4.2 The M-machine

The M-machine is an abstract general-purpose register machine. For historical reasons, it has a CISC-like instruction set, although for modern processors the Chalmers compiler simplifies programs to use only a RISC-like subset. By way of example, here is the M-code corresponding to the CPAIR_RET *m n* instruction above.

```
move    0(Vp),    Sp
move    -1(Sp),   r0
move    $PAIRm,   0(r0)
move    p,        1(r0)
move    q,        2(r0)
return
```

This code uses three registers: `Sp`, `Vp` and `r0`. The first two are stack pointers for the pointer and dump stacks; the last is a return register for function results. The addressing modes *p* and *q* depend on the context in which the M-code is generated. Note that *n* is not used because here the node to be updated can be found using `Sp` alone.

Since there is such a direct relationship between M-instructions and native code instructions, M-code converts to a byte code with almost the same instruction density as native code.

4.3 The X-machine

Our X-machine is a cross between the G-machine and the M-machine. Each X-instruction is created by studying a G-machine instruction and the corresponding M-instructions side-by-side. The X-instruction opcode comes from the name of the G-machine instruction; any variables in the M-instructions appear as X-instruction operands, with small integers followed by a list of general addressing modes. Sometimes, a little common sense is then used to improve the instruction. So, to continue with our current example, instead of a single X-instruction of the form

`cpair_ret` *m* [*p*, *q*]

there are six of the form

`cpair_ret`*m* [*p*, *q*]

because the peculiar Chalmers implementation of constructed types means that there are only six possible values of *m* (0–4, and 'don't care'), and these can conveniently

https://doi.org/10.1017/S0956796897002955 Published online by Cambridge University Press

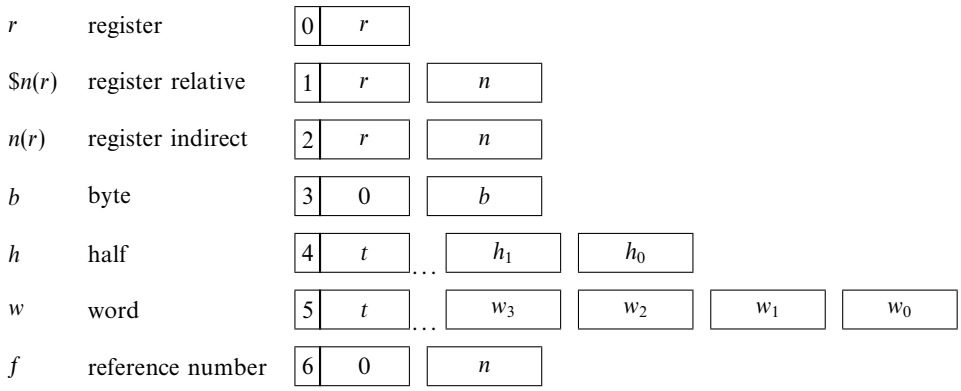

Fig. 2. Operands and their byte code representation.

be made part of the opcode. The interested reader is referred to Appendix A for a summary of the full X-machine instruction set, and to Appendix B for two example functions.

5 Compilation

The X-machine has been designed so that programs can be stored compactly (like the G-machine) and executed efficiently (like the M-machine).

5.1 Byte code representation

X-instructions are represented by a single-byte opcode followed by further bytes for any operands. Fig. 2 describes the possible operands and their byte code representation. The upper three bits of the first byte give the addressing mode, and the lower five bits give either a register number r or an amount of padding t depending on the mode. For the register relative and register indirect modes, another byte gives the offset n (it is up to the compiler to ensure that offsets are small enough). For the byte, half and word modes, one, two or four more bytes give a constant. On an aligned machine, t padding bytes (indicated by ellipses) ensure that the constant is aligned for easy access. A reference number is an index into an array of addresses of functions and constant applicative forms that is kept with each function for use by the garbage collector. The compiler uses reference numbers in preference to addresses because they can be stored more compactly. This operand representation is the most compact possible for register addressing modes (which we have found are common), and it is also reasonably compact for constants (which we have found are less common).

5.2 Dynamic compilation

X-code is designed to be executed using *throw-away compilation* (Brown, 1976), one of the oldest forms of dynamic compilation. A throw-away compiler performs

https://doi.org/10.1017/S0956796897002955 Published online by Cambridge University Press

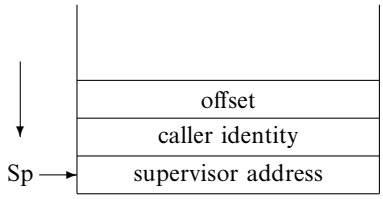

Fig. 3. Stack set up for an indirect return.

both *static* and *dynamic* compilation of the program. At what we usually think of as compile-time, the static compiler divides the source code into suitable units (in Brown's case, these were BASIC statements) and produces byte code for each unit. At what we usually think of as run-time, a *supervisor* oversees execution of the byte code program. It hands successive units of byte code to the dynamic compiler which produces native code for direct execution. Native code is kept in a *workspace*. When the native code for a unit is added to the workspace, this is noted by the supervisor to avoid producing another copy. When the workspace becomes full, it is emptied by throwing the native code for all units away. Thus, the workspace serves as a native code cache with a particularly brutish replacement policy that is cheap to implement.

6 A test bed implementation

Modifying the LML/HBC compiler (Augustsson and Johnsson, 1989) to implement the ideas above requires little imagination. The static compiler divides the source code into (lambda-lifted) functions and produces X-code for them. The dynamic compiler then produces native code from X-code by simple macro expansion, taking the offsets for jump instructions from the byte code. It is easiest to make the workspace part of the heap. By doing so, we make best use of the limited available memory – it would be a shame for a program to fail with no space available for the graph, but with plenty available for code. All native code is thrown away during garbage collection. Three points are worth mentioning in more detail: the management of function return, the compilation of case-expressions and the maintenance of cache coherency.

6.1 The management of function return

Obviously, every function call must go via the supervisor, which either locates or generates the function's native code before transferring control to it. Less obviously, every function return must also go via the supervisor because throwing native code away leaves return addresses dangling. A return via the supervisor can be managed by storing three items on the stack when a function call is made from dynamically compiled code. These three items replace the usual return address (see Fig. 3). A return from the callee is directed to the supervisor, which normally just pops the caller identity and the offset before passing control to the caller's native code address plus the offset. But when the caller's native code has been thrown away,

https://doi.org/10.1017/S0956796897002955 Published online by Cambridge University Press

the supervisor must get it regenerated before passing control to the new native code address plus the offset.

6.2 The compilation of case-expressions

There is a dilemma when dynamically compiling `case`-expressions. On the one hand, if the function containing the `case`-expression is applied once then it is a waste of effort to compile code for every alternative because only one will eventually be chosen. On the other hand, if the function containing the `case`-expression is applied several times then the effort to compile code for every alternative may not be wasted because all may eventually be chosen. Our solution to this problem is to bound the effort required to dynamically compile a `case`-expression by *thinning*. A `case`-expression is thinned if it has more than a few alternatives (we use five), or if it is nested within several other `case`-expressions (we use a nesting depth of two). Thinning involves floating alternatives containing `let-` or `case`-expressions out to the top level as new functions whose arguments are the free variables of the alternative. The alternatives of the thinned `case`-expression then become applications of these new functions to the appropriate variables. Thinning is easily implemented by modifying the lambda-lifting pass of the compiler.

6.3 The maintenance of cache coherency

As we have already noted, modern computers cache instructions by copying blocks of main memory into faster cache memory. Often, there are separate on-chip caches for instructions and data, and the instruction cache does not track memory writes. This means that a newly-compiled instruction in main memory may not be 'seen' until a previously-compiled one happens to lose its place in the instruction cache. Happily, many computers provide a cache coherency operation – for example, a system call to flush a range of addresses from the instruction cache – and so this difficulty can be overcome. This issue is explored further in the paper by Keppel (1991).

7 Experimental results

To evaluate throw-away compilation, we performed some experiments on an SGI O2 computer. This machine has a 180 MHz R5000 processor and 32 Mb of main memory. Although the R5000 processor has a 32 kb instruction cache and a 32 kb data cache, there is no other cache. The machine runs version 6.3 of the IRIX operating system.

Six of the larger programs from the 'nofib' suite (Partain, 1992) were used as benchmarks. These programs were modified in two ways. First, the input was made much larger, usually by repeating it, so that the programs would run for a reasonable length of time. Secondly, the programs were made to print just the last character of their results, so that output time was negligible. For each program, we measured two things.

1. The *code size*. This excludes functions from the standard prelude, which

https://doi.org/10.1017/S0956796897002955 Published online by Cambridge University Press

Table 1. *Code sizes for our compiler compiling to byte code*

Program	Text segment (bytes)	Data segment		Total (bytes)
		Byte code (bytes)	Other (bytes)	
`parser`	9,784	19,357	34,707	63,848
`infer`	9,216	9,678	23,506	42,400
`reptile`	10,656	24,491	39,285	74,432
`bspt`	10,848	18,783	64,465	94,096
`prolog`	3,936	6,425	12,119	22,480
`veritas`	33,600	109,972	126,668	270,240

Table 2. *Code sizes for our compiler compiling to native code*

Program	Text segment (bytes)	Data segment (bytes)	Total (bytes)
`parser`	80,520	34,544	115,064
`infer`	46,352	23,024	69,376
`reptile`	104,720	38,816	143,536
`bspt`	89,232	64,384	153,616
`prolog`	30,376	11,952	42,328
`veritas`	467,712	125,904	593,616

Table 3. *Code sizes for the HBC compiler compiling to native code*

Program	Text segment (bytes)	Data segment (bytes)	Total (bytes)
`parser`	80,136	38,336	118,472
`infer`	46,736	24,704	71,440
`reptile`	106,592	42,640	149,232
`bspt`	86,400	76,192	162,592
`prolog`	30,800	13,120	43,920
`veritas`	450,928	139,520	590,448

are tedious to pick out, and run-time support code, (where the throw-away compiler – essentially, a large C `switch` statement – adds 12 kbytes to the existing 70 kbytes).

2. The *execution time*. Following Hammond *et al.* (Hammond *et al.*, 1993), we made measurements with a large number of heap sizes to investigate whether using the heap as a native code cache has any strange effects. All measurements start from a heap size at which the program almost runs out of memory.

https://doi.org/10.1017/S0956796897002955 Published online by Cambridge University Press

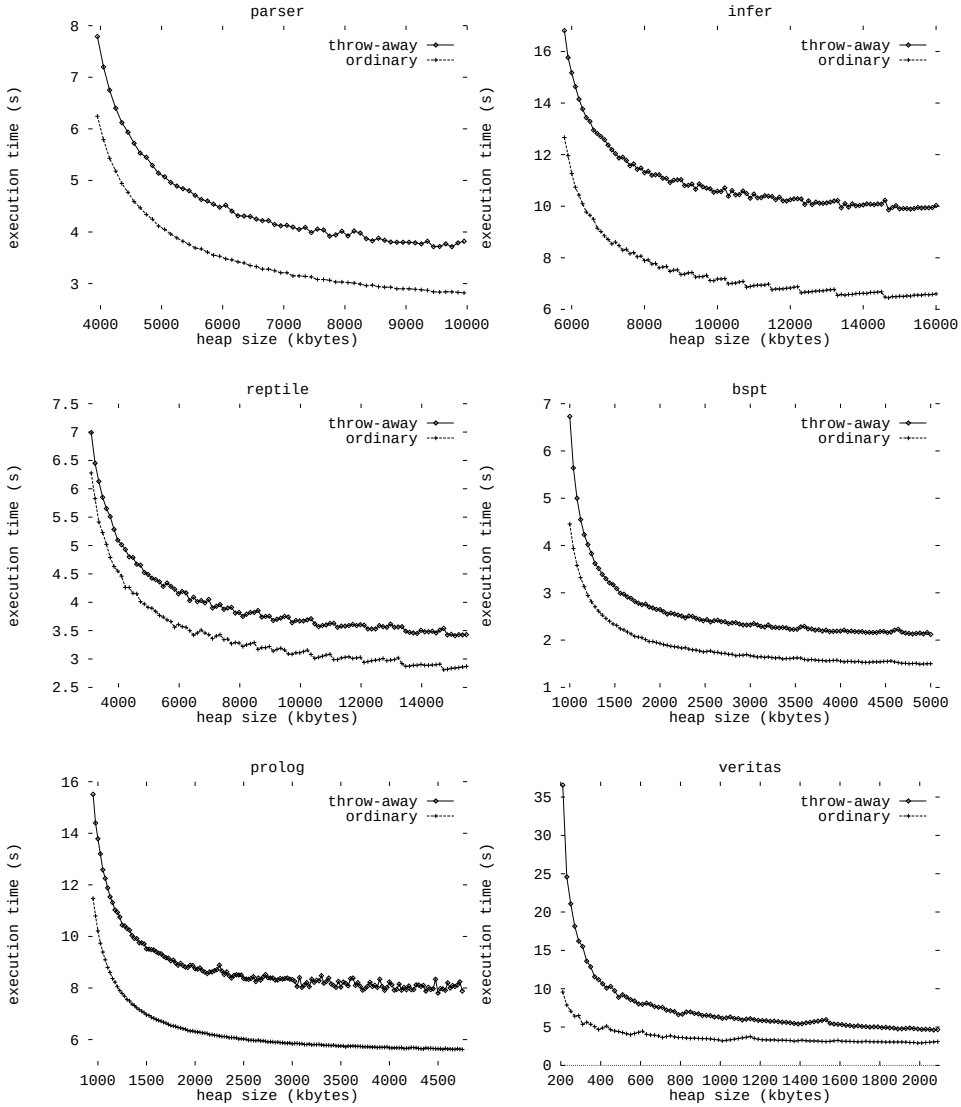

Fig. 4. Execution time *vs* heap size for our compiler.

Our compiler, which is based on version 0.9999.3 of the Chalmers LML/HBC compiler, was used to compile the six benchmark programs into byte code and native code. An ordinary version 0.9999.3 Chalmers LML/HBC compiler was also used to compile the six benchmark programs into native code. Tables 1, 2 and 3 give the text segment (native code) and data segment (statically-allocated data) sizes in each case.

Fig. 4 plots execution time *vs* heap size when all functions (including those in the standard prelude) are compiled into byte code and native code with our compiler. Fig. 5 plots execution time *vs* heap size when all functions (including those in

https://doi.org/10.1017/S0956796897002955 Published online by Cambridge University Press

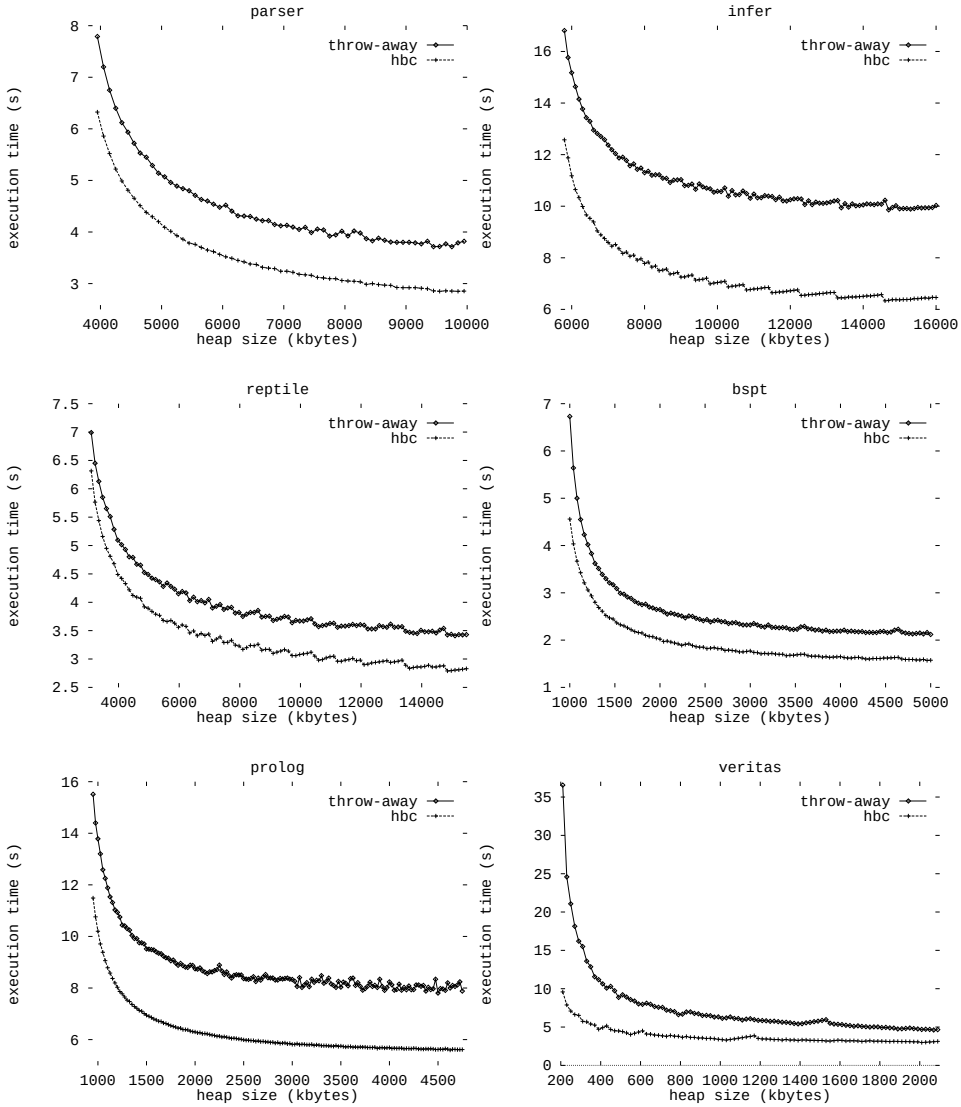

Fig. 5. Execution time *vs* heap size for our compiler and the HBC compiler.

the standard prelude) are compiled into byte code with our compiler and native code with the ordinary HBC compiler. The ratio of these execution times *vs* heap size is plotted in Fig. 6. As well as execution times, we were interested in the number of function (re)compilations that take place. Figure 7 plots the number of (re)compilations *vs* heap size.

From these results, we observe two things. First, there is practically no difference between compiling to native code with our compiler and with the ordinary HBC compiler. This isn't a great surprise. The two compilers have much in common, including the same native code generator. Secondly, compiling to byte code makes

https://doi.org/10.1017/S0956796897002955 Published online by Cambridge University Press

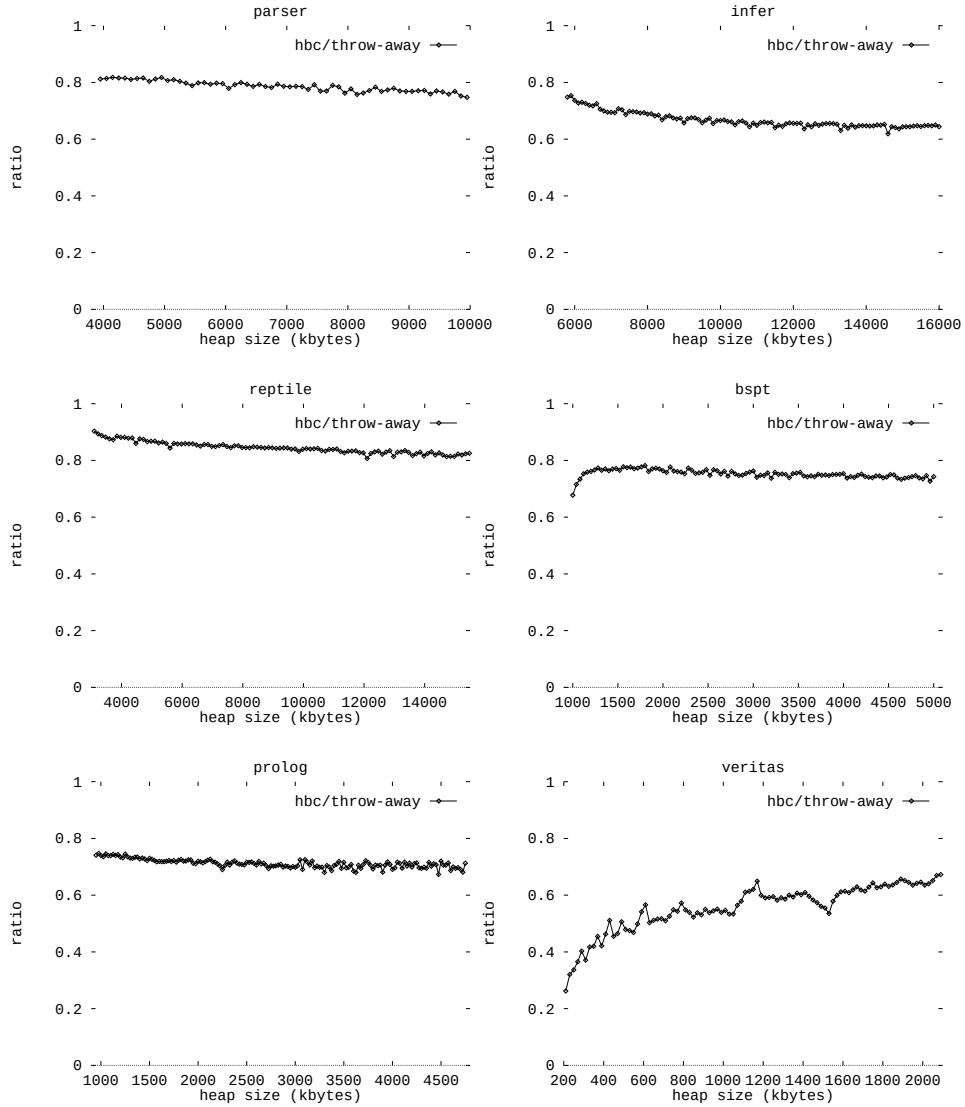

Fig. 6. Ratio of execution times *vs* heap size.

the text segment smaller – the only native code left is the transfer of control to the supervisor at the start of each function – and the data segment larger – the byte code adds to the statically-allocated data. The combined size of this residual native code and byte code for our compiler is typically only 33% of the size of the ordinary native code produced by our compiler or the HBC compiler. Compiling to byte code also makes programs run more slowly because thousands of (re)compilations must take place. Regardless of the heap size, there will always be some supervisor overhead on function call and return. Programs with throw-away compilation typically run at 75% of the speed of those with ordinary compilation.

https://doi.org/10.1017/S0956796897002955 Published online by Cambridge University Press

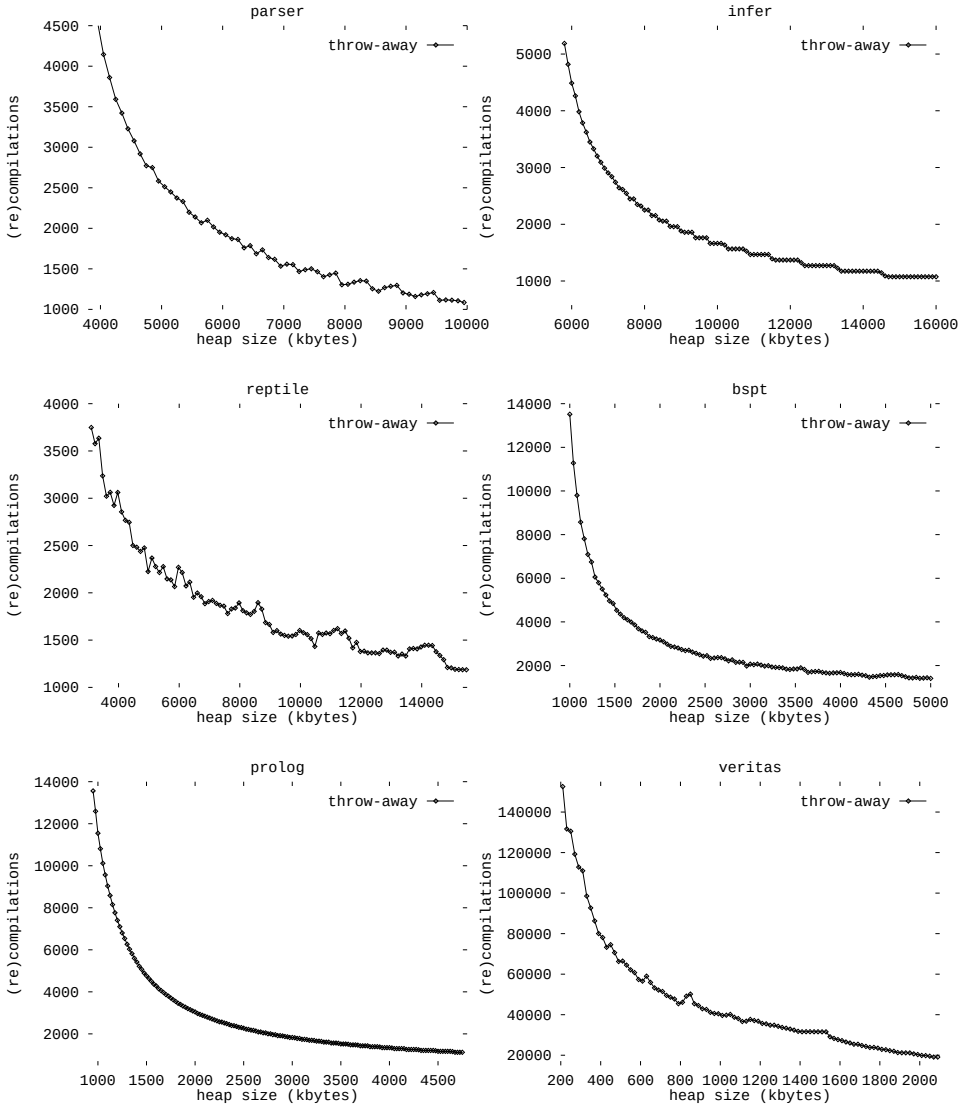

Fig. 7. Number of function (re)compilations *vs* heap size.

8 Future work

Our first experiments have yielded promising results. Unfortunately though, the large reduction in *code* size does not yet result in such a large reduction in *program* size because of statically-allocated data and run-time support. To reduce the size of statically-allocated data, it would be necessary to move from the HBC word-based encoding to a byte-based one. So far we have not done this because compatibility with the HBC compiler makes development of our own so much easier. Of the 70 kbytes for the run-time system, roughly a third is for input/output, a third for memory management and a third for miscellaneous small routines written in M-

https://doi.org/10.1017/S0956796897002955 Published online by Cambridge University Press

code. Perhaps 20% of this code is to support LML rather that Haskell, but again we have so far avoided removing it because the compiler is written in LML and bootstrapping is such an important test.

By analogy with Denning's work on virtual memory (Denning, 1968), we suppose that a 'working set' of functions exists for most functional programs. We plan to collect data to confirm this supposition, perhaps by separating the storage areas for the code and the graph so that we can study the behaviour of the code alone. This may lead us to develop better rules for throwing code away, akin to those found in operating systems.

Even in its most basic form, throw-away compilation seems to be making good use of the features of modern computer architecture described in section 3. However, it would be nice to have numbers for cache misses, pipeline stalls, and so on from a processor simulator to compare throw-away compilation, ordinary compilation and interpretive execution. These numbers may lead us to improve the design of the X-machine and the implementation of throw-away compilation.

It is not clear how well throw-away compilation would work in an application with demanding with real-time constraints, and this remains to be investigated.

In this paper we have considered only lazy functional languages, such as Haskell. Although we see no reason why the dynamic compilation approach we have described should not work equally well with a strict functional language such as Standard ML, we need to build an implementation (perhaps based on Leroy's byte code interpreter (Leroy, 1990)) to be sure.

At the moment, there is no easy way for the programmer to take advantage of profiling information. Although functions compiled with ordinary compilation and those compiled with the throw-away compilation can be freely mixed, each module must be compiled one way or the other. This is not very flexible. A programmer who wishes to use profiling information to make the best time/space trade off must shuffle functions between a pair of modules, one compiled with each scheme. In future, it is our intention to allow the programmer to annotate each function to indicate the compilation scheme required.

Our first throw-away compiler ran on a cast-off SUN SPARCstation SLC. After that machine expired, we ported the compiler to an SGI Indigo, and in doing so discovered that making the changes necessary to deal with a new instruction encoding could be a tedious and error-prone business. Upto now, portability has not been a concern, but in future as we move towards embedded processors we plan to save work by using something akin to Engler's VCODE (Engler, 1996).

9 Related work

To our knowledge, no one yet programs embedded computers with lazy functional languages. The nearest work that we know of is by Wallace and Runciman in the area of real-time control (Wallace and Runciman, 1995). Ericcson Telecom use a mostly-functional language Erlang (Armstrong *et al.*, 1993) for programming telephone switches.

Dynamic compilation is an old idea whose popularity waxes and wanes according

to the fashion in computer architecture. In their 1991 paper, Keppel *et al.* (1991) argued that 'new' computer architecture made it worthwhile again, and currently (Autumn 1997) there is a considerable amount of research going on this area. Here, we concentrate on some of the most recent papers since they provide a good way into the otherwise scattered literature on dynamic compilation.

Dynamic compilation has long been used to improve the performance of object-oriented programming languages. Both Deutsch and Schiffman's (1984) SmallTalk system and Chambers and Ungar's (1989) SELF compiler generate native code from the byte code of an object method when that method is invoked. The native code is cached and used if the method is invoked again. Should the cache fill up, some native code is flushed, to be regenerated when needed. SmallTalk was among the first languages to represent programs as byte code for portability. The work mentioned here, however, is concerned with increasing execution speed rather than with decreasing code size.

As part of the Fox Project aimed at improving the design and development of high-performance system software, Lee and Leone have built a run-time code generator for a first-order, purely-functional subset of ML (Lee and Leone, 1996). Here, curried functions take an 'early' argument followed by some 'late' arguments, the idea being that 'early' computations are performed by statically-generated code, and 'late' computations by dynamically-generated code. Each curried function is compiled into assembly code, with instructions being annotated as either 'early' or 'late'. The 'early' assembly instructions are simply executed, while the 'late' ones are specialised with now-known values and emitted into a dynamic code segment for immediate execution. There are no code templates to be copied and instantiated, and so dynamic code generation is cheap (only about six instructions per instruction generated). Lee and Leone are concerned with making progams faster by taking advantage of values that only become known at run-time, rather than with making programs smaller. They reported encouraging results for small programs, but their prototype implementation had no garbage collector, and so they could not say how well it would work for larger, more realistic programs.

The paper by Auslander *et al.* (1996) deals with the dynamic compilation of general-purpose, imperative, programming languages, such as C. Here, the program must be annotated to indicate *regions* where dynamic compilation may be worthwhile. For these regions, a static compiler produces machine code templates with holes for run-time constant values; the rest of the program is just compiled in the usual way. As the program is running, a dynamic compiler copies the templates, filling in the holes with appropriate constants and performing simple peephole optimisations. Again, the emphasis is on making programs faster by taking advantage of values that become known at run-time, rather than on making programs smaller. Pointers, side-effects and unstructured control-flow present the usual problems for the authors as they attempt to produce an optimising C compiler. Nonetheless, they have managed to achieve good speed-ups over a set of ordinarily-compiled C programs, ranging from 1.2 to 1.8 times as fast.

The Omniware Virtual Machine produced by Adl-Tabatabai *et al.* (1996) is an abstract RISC that runs *mobile programs* sent to it across a network. The Omniware

Virtual Machine has been designed so that code generation for it by compilers is easy, and the subsequent translation of this code to that of a modern processor is fast and efficient. Initially, the aim was to achieve portability without compromising efficiency. Even when performing safety checks to ensure that the program does not violate access restrictions, the Omniware Virtual Machine executes programs within 21% as fast as ordinary optimised ones without any such checks. This is claimed to be an order of magnitude faster than any other mobile code system. More recent research has concentrated on saving space too by compressing Omniware Virtual Machine code into a byte code that can either be interpreted directly or compiled to native code (Ernst *et al.*, 1997).

The Java Virtual Machine (Lindholm and Yellin, 1996) is an abstract stack machine that also runs mobile programs sent across a network. The byte code representation is used to achieve portability and save space. Although the original SUN implementation was a byte code interpreter, many Java implementors, including SUN, are now developing 'just-in-time' compilers to speed up execution. These translate the byte code for methods into native code as they are called, and then run the native code. SUN claim that this can be upto an order of magnitude faster than interpreting the byte code.

Engler's (1996) VCODE is a general-purpose system for implementing dynamic code generation. The VCODE system consists of a set of C macros and functions for generating machine code. Using them, a VCODE client (for example, a compiler) can construct machine code at run-time assuming the underlying processor is an idealied RISC. Chores such as constructing function prologue and epilogue code, backpatching labels and ensuring cache coherency can all be left to the VCODE system. As a result, programs that generate code dynamically are both portable and efficient (between six and ten instructions per instruction generated).

Throw-away compilation was first mentioned in the context of functional programming by Turner (1979), and this was where we got the idea. An earlier version of this paper, with the same results for small LML programs, appeared as Wakeling (1995).

10 Conclusions

In this paper we have described a new abstract machine for the implementation of lazy functional languages on embedded computers. It has been designed so that program code can be stored compactly as byte code, yet run quickly using dynamic compilation. Our first results are promising – programs typically require only 33% of the code space of an ordinary implementation, but run at 75% of the speed. Future work must now concentrate on reducing the size of statically-allocated data and the run-time system, and on developing a more detailed understanding of throw-away compilation.

Acknowledgements

Our thanks as usual to Lennart Augustsson and Thomas Johnsson, whose work on the LML/HBC compiler forms the basis of our own.

Operation	Operands	Comment
`alloc`	h	create $\langle$HOLE$\rangle$ at h(Hp)
`add`	*reg* $[p, q]$	add integers p and q
`addsf`	*reg* $[p, q]$	add single floats p and q
`adddf`	*reg* $[p, q]$	add double floats p and q
`cint`	h $[n]$	create $\langle$INT, $n\rangle$ at h(Hp)
`cnil`	h $[n]$	create $\langle$TAG0, $n\rangle$ at h(Hp)
`ctag`	h n $[p]$	create $\langle$TAG, $n, p\rangle$ at h(Hp)
`cap`	h $[p, q]$	create $\langle$AP, $p, q\rangle$ at h(Hp)
`cpair`	h n $[p, q]$	create $\langle$PAIRn, $p, q\rangle$ at h(Hp)
`cvap`	h *ref* $[p_1, p_2, \ldots p_n]$	create $\langle$VAP, *ref*$, p_1, p_2, \ldots p_n\rangle$ at h(Hp)
`cvek`	h $[p_1, p_2, \ldots p_n]$	create $\langle$VEK, $p_1, p_2, \ldots p_n\rangle$ at h(Hp)
`cint_update`	n	update with $\langle$INT, $n\rangle$
`cnil_update`	n	update with $\langle$TAG0, $n\rangle$
`ctag_update`	n $[p]$	update with $\langle$TAG, $n, p\rangle$
`cpair_update`	n $[p, q]$	update with $\langle$PAIRn, $p, q\rangle$
`cint_ret`	n	update with, and return $\langle$INT, $n\rangle$
`cnil_ret`	n	update with, and return $\langle$TAG0, $n\rangle$
`ctag_ret`	n $[p]$	update with, and return $\langle$TAG, $n, p\rangle$
`cpair_ret`	n $[p, q]$	update with, and return $\langle$PAIRn, $p, q\rangle$
`jfun`	n $[f]$	tail call unknown f with n arguments
`jglobal`	*ref*	tail call known *ref*
`jmp`	n	jump to label n
`jeq`	n $[p, q]$	jump to label n if integer $p = q$
`jeqsf`	n $[p, q]$	jump to label n if single float $p = q$
`jeqdf`	n $[p, q]$	jump to label n if double float $p = q$
`callfun`	n $[f]$	call unknown f with n arguments
`callglobal`	n *ref*	call known *ref* with n arguments
`eval`	$[p]$	evaluate
`gettag`	$[p]$	get tag into *tagreg*
`gettag_switch`	$(t_1, n_1) \ldots (t_k, n_k)$ $[p]$	get tag t_i and jump to label n_i
`move`	$[p, q]$	move integer p to q
`movesf`	$[p, q]$	move single float p to q
`movedf`	$[p, q]$	move double float p to q
`incsp`	n	increment stack pointer by n
`zap`	n	blackhole argument n
`garb`		initiate garbage collection if required
`ret`	$[p]$	return p
`update`	$[p, q]$	update q with p
`unwind`	$[p]$	unwind p
`splitpair`	n	copy components to $(n + 1, n + 2)$(Sp)

Fig. 8. Summary of the X-machine instruction set.

Colin Runciman provided some useful comments on earlier versions of this paper, and encouragement throughout. In addition, we are grateful for the comments of five anonymous referees.

This work was funded in part by Canon Research Centre Europe, and in part by the University of Exeter Research Fund.

Summary of the X-machine instruction set

Fig. 8 gives a summary of the X-machine instruction set. This summary will make most sense when read in conjunction with the description of the Chalmers HBC/LML compiler (Augustsson and Johnsson, 1989).

Two example functions

Below, we give two example Haskell functions and their translation into X-code.

Factorial

```
  fact :: Int -> Int
  fact n = if n == 0 then 1 else n * fact (n - 1)

  funstart fact 1              # --- fact ---
  zap 1                        # blackhole 1(Sp)
  eval [0(Sp)]                 # eval n
  jne L1 [$0,1(r0)]            # branch if n /= 0
  cint_ret 1                   # return <INT,1>
L1:
  garb                         # check for heap exhaustion
  alloc 0                      # create <HOLE> at 0(Hp)
  move [Hp,-1(Sp)]             # push address of HOLE
  move [0(Sp),r1]              # r1 <- <INT,n>
  sub r1 [$1,1(r1)]            # r1 <- n - 1
  bint 3 [r1]                  # create <INT,n-1> at 3(Hp)
  move [$3(Hp),-2(Sp)]         # push address of <INT,n-1>
  inc_hp 5                     # adjust Hp
  inc_sp -2                    # adjust Sp
  callglobal 0                 # call fact recursively
  move [0(Sp),r1]              # r1 <- <INT,n>
  mul r1 [1(r0),1(r1)]         # r1 <- n * result of call, n'
  bint_ret [r1]                # return <INT,n*n'>
  funend                       # --- fact ---
```

Map

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

  funstart map 2               # --- map ---
  zap 2                        # blackhole 2(Sp)
  eval [1(Sp)]                 # eval xs
  move [r0,-1(Sp)]             # push xs
  inc_sp -1                    # adjust Sp
  gettag_switch {35 => L1} [r0] # switch on tag(xs)
  garb                         # check for heap exhaustion
  move [0(Sp),r1]              # r1 <- xs
  cvap 0 0 [1(Sp),2(r1)]       # create <VAP,f,tl(xs)> at 0(Hp)
  cap 4 [1(Sp),1(r1)]          # create <AP,f,x> at 4(Hp)
  inc_hp 7                     # adjust Hp
  cpair_ret 1 [$-3(Hp),$-7(Hp)] # return <CONS,AP f x,VAP f,tl(xs)>
L1:
  cnil_ret 0                   # return <NIL>
  funend                       # --- map ---
```

https://doi.org/10.1017/S0956796897002955 Published online by Cambridge University Press

References

Adl-Tabatabai, A., Langdale, G., Lucco, S. and Wahbe, R. (1996) Efficient and language independent mobile programs. *Proc. ACM Conf. on Programming Language Design and Implementation*, pp. 127–136.

Armstrong, J. L., Virding, R. and Williams, M. (1993) *Concurrent Programming in Erlang*. Prentice-Hall.

Augustsson, L. and Johnsson, T. (1989) The Chalmers Lazy-ML Compiler. *Computer J.*, **32**(2): 127–141.

Auslander, J., Philipose, M., Chambers, C., Eggers, S. J. and Bershad, B. N. (1996) Fast, effective dynamic compilation. *Proc. ACM Conf. on Programming Language Design and Implementation*, pp. 149–159.

Bird, R. and Wadler, P. (1988) *Introduction to Functional Programming*. Prentice-Hall.

Brown, P. J. (1976) Throw-away Compiling. *Software—Practice and Experience*, **6**: 423–434.

Chambers, C. and Ungar, D. (1989) Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. *Proc. ACM Conf. on Programming Language Design and Implementation*, pp. 146–160.

Denning, P. J. (1968) The working set model for program behaviour. *Comm. ACM*, **11**(5): 323–333.

Deutsch, L. P. and Schiffman, A. M. (1984) Efficient implementation of the Smalltalk-80 system. *Proc. 11th Annual ACM Symposium on the Principles of Programming Languages*, pp. 297–302.

Ernst, J., Evans, W., Fraser, C. W., Lucco, S. and Proebsting, T. A. (1997) Code compression. *Proc ACM Conf. on Programming Language Design and Implementation*, pp. 358–365.

Engler, D. W. (1996) VCODE: A retargetable, extensible, very fast dynamic code generation system. *Proc. ACM Conf. on Programming Language Design and Implementation*, pp. 160–170.

Hammond, K., Burn, G. L. and Howe, D. B. (1993) Spiking your caches. *Proc. Glasgow Workshop on Functional Programming*, pp. 58–68. Springer-Verlag.

Hennessy, J. L. and Patterson, D. A. (1990) *Computer Architecture: A Quantative Approach*. Morgan Kaufmann.

Hudak, P. and Jones, M. P. (1994) Haskell vs. Ada vs. C++ vs Awk vs. .... *Technical report*, Department of Computer Science, Yale University.

Jones, M. P. (1994) The Implementation of the Gofer Functional Programming System. *Technical report*, Department of Computer Science, Yale University.

Keppel, D. (1991) A portable interface for on-the-fly instruction space modification. *Proc. ACM Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 86–95. (*SIGPLAN Notices*, **26**(4) 1991.)

Keppel, D., Eggers, S. J. and Henry, R. R. (1991) A Case for Runtime Code Generation. *Technical report 91–11–04*, Department of Computer Science and Engineering, University of Washington.

Lee, P. and Leone, M. (1996) Optimizing ML with run-time code generation. *Proc. ACM Conf. on Programming Language Design and Implementation*, pp. 137–148.

Leroy, X. (1990) The ZINC Experiment: An Ecomonical Implementation of the ML Language. *Technical report RT 117*, INRIA, France.

Lindholm, T. and Yellin, F. (1996) *The Java Virtual Machine*. Addison-Wesley.

Partain, W. (1992) The nofib benchmark suite of Haskell programs. *Proc. Glasgow Workshop on Functional Programming*, pp. 195–202. Springer-Verlag.

Patterson, D. A. and Hennessy, J. L. (1994) *Computer Organization and Design: The Hardware Software Interface*. Morgan Kaufmann.

Peyton Jones, S. L. (1987) *The Implementation of Functional Programming Languages*. Prentice-Hall.

Proebsting, T. A. (1995) Optimizing an ANSI C interpreter with superoperators. *Proc. ACM Conf. on Principles of Programming Languages*, pp. 322–332.

Röjemo, N. (1995) Highlights from nhc – a space-efficient Haskell compiler. *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, pp. 282–291.

Turner, D. A. (1979) A new implementation technique for applicative languages. *Software—Practice and Experience*, **9**(1): 31–50.

Turner, D. A. (1982) Recursion equations as a programming language. In: Darlington, J., Henderson, P. and Turner, D. A. (eds.), *Functional Programming and its Applications*, pp. 1–28. Cambridge University Press.

Wakeling, D. (1995) A throw-away compiler for a lazy functional language. In: Takeuchi, M. and Ida, T. (eds.), *Fuji International Workshop on Functional and Logic Programming*, pp. 287–300. World Scientific.

Wallace, M. and Runciman, C. (1995) Lambdas in the liftshaft – functional programming and an embedded architecture. *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, pp. 249–258.