The Glasgow Haskell compiler: a technical overview

Simon L Peyton Jones Cordy Hall Kevin Hammond Will Partain
Phil Wadler
Department of Computing Science, University of Glasgow, G12 8QQ.
email: simonpj@dcs.glasgow.ac.uk

December 22, 1992

This paper appears in the Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference, Keele, 1993.

Abstract

We give an overview of the Glasgow Haskell compiler, focusing especially on way in which we have been able to exploit the rich theory of functional languages to give very practical improvements in the compiler.

The compiler is portable, modular, generates good code, and is freely available.

1 Introduction

Computer Science is both a *scientific* and an *engineering* discipline. As a scientific discipline, it seeks to establish generic principles and theories that can be used to explain or underpin a variety of particular applications. As an engineering discipline, it constructs substantial artefacts of software and hardware, sees where they fail and where they work, and develops new theory to underpin areas that are inadequately supported. (Milner [1991] eloquently argues for this dual approach in Computer Science.)

Functional programming is a research area that offers an unusually close interplay between these two aspects (Peyton Jones [1992b]). Theory often has immediate practical application, and practice often leads directly to new demands on theory. This paper describes our experience of building a substantial new compiler for the purely functional language Haskell. We discuss our motivations, major design decisions and achievements, paying particular attention to the interaction of theory and practice.

Scaling prototypes up into large "real" systems appears to be less valued in the academic community than small systems that demonstrate concepts, being sometimes being dismissed as "just development work". Nevertheless, we believe that many research problems can only be exposed during the act of constructing large and complex systems. We hope that this paper serves to substantiate this point.

The compiler work described in this paper is one of the two strands of the SERC-funded GRASP project. The other concerns parallel functional programming on the GRIP multiprocessor, but space precludes coverage of both.

2 Goals

Haskell is a purely functional, non-strict language designed by an international group of researchers (Hudak et al. [1992]). It has now become a *de facto* standard for the non-strict (or lazy) functional programming community, with at least three compilers available.

Our goals in writing a new compiler were these:

- *To make freely available a robust and portable compiler for Haskell that generates good quality code.* This goal is more easily stated than achieved. Haskell is a rather large language, incorporating: a rich syntax; a new type system that supports systematic overloading using so-called *type classes* (Wadler & Blott [1989]); a wide variety of built-in data types including arbitrary-precision integers, rationals, and arrays; a module system; and an input/output system.
- *To provide a modular foundation that other researchers can extend and develop.* In our experience, researchers are often unable to evaluate their ideas because the sheer effort of building the framework required is too great. We have tried very hard to build our compiler in a well-documented and modular way, so that others will find it (relatively) easy to modify.
- *To learn what real programs do.* The intuition of an implementor is a notoriously poor basis for taking critical design decisions. The RISC revolution in computer architecture was based partly on the simple idea of measuring what real programs actually do most often, and implementing those operations very

well (Hennessy & Patterson [1990]). Lazy functional programs execute in a particularly non-intuitive fashion, and we plan to make careful, quantitative measurements of their behaviour.

3 An overview of the compiler

The compiler and its runtime system have the following major characteristics:

- It is written almost entirely in Haskell. The only exception is that the parser is written in Yacc and C.
- It generates C as its target code. This has now become relatively common, conferring as it does wide portability. The big question is, of course, what efficiency penalty is paid, a matter we discuss in Section 6.
- We have extended the language to allow mixed-language programming, by supporting arbitrary in-line statements written in C. It is, of course, not possible to do this in a completely secure way (for example, the C procedure could overwrite the Haskell heap), but our technique *is* referentially transparent; that is, all the usual program transformations remain valid.

  This mixed-language working allows us to extend Haskell easily, for example to provide access to existing procedure libaries. Without a general way of calling C, each such extension would require a separate modification to the code generator.
- Haskell's monolithic arrays are fully implemented, with $O(1)$ access time. In addition, we have extended the language with incrementally-updatable arrays; indeed, the monolithic arrays are implemented using these mutable arrays.
- The interface between the storage manager and the compiler is carefully defined and highly configurable. For example, the storage manager comes with no fewer than four different garbage collectors, including a generational one.
- The compiler and its runtime system also support comprehensive runtime profiling of both space and time, at both the user level and the evaluation-model level (Sansom & Peyton Jones [1993]).

3.1 Organisation

The overall organisation of the compiler is quite conventional. A driver program runs a sequence of Unix processes, namely: a "literate-script" pre-processor, the main parser, the compiler itself, the C compiler, the Unix assembler, and the Unix linker. The main passes of the compiler itself are shown in Figure 1. They are as follows (Figure 2 summarises their sizes):

Module	Lines
Major passes	
Main	997
Parser	1055
Renamer	1828
Type inference	3352
Desugaring	1381
Core-language transformation	1631
STG-language transformation	814
Code generation	2913
Data type definition and manipulation	
Haskell abstract syntax	2546
Core language	1075
STG language	517
Abstract C	1416
Identifier representations	1831
Type representations	1628
Prelude definitions	3111
Utility modules	
Utilities	1989
Profiling	191
TOTAL	28,275

Figure 2: Breakdown of module sizes

1. A simple recursive-descent *parser* recognises the simple syntax output by the separate main parser process. The abstract syntax tree produced by this parser faithfully represents every construct in the Haskell source language, even where a distinction is purely syntactic. This improves the readability of error messages from the type checker.
2. The *renamer* resolves scoping and naming issues, especially those concerned with module imports and exports.
3. The *type inference pass* annotates the program with type information, and transforms out all the overloading. The details of the latter transformation are given by Wadler & Blott [1989]; we do not discuss it further here.
4. The *desugarer* converts the rich Haskell abstract syntax into a very much simpler functional language we call the *Core language*. Notice that desugaring *follows* type inference. As a result, type error messages are expressed in terms of the original source, and they also appear more quickly.
5. A variety of optional *Core-language transformation passes* improve the code.

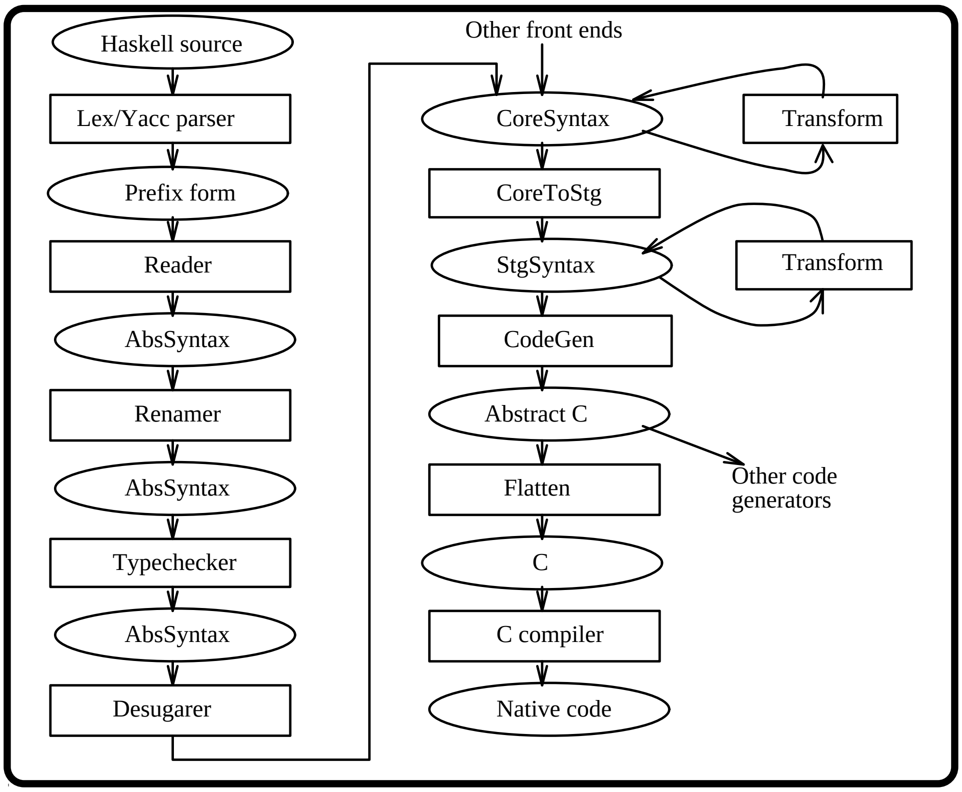

Figure 1: Overview of the Glasgow Haskell Compiler

6. A simple pass then converts Core to the *Shared Term Graph (STG) language*[1], an even simpler, but still purely functional, language.

7. A variety of optional *STG-language transformation passes* improve the STG code.

8. The *code generator* converts the STG language to *Abstract C*. Abstract C is no more than an internal data type that can be printed in C syntax, or if preferred (though we have not done this) in assembly-language syntax for a particular machine.

9. The *target-code printer* prints Abstract C in a form acceptable to a C compiler.

3.2 Compilation by transformation

A consistent theme runs through all our design decisions, namely that *most of the compilation process is expressed as correctness-preserving transformations of a purely-functional program.* A wide variety of conventional imperative-program optimisations have simple counterparts as functional-language transformations, each of which "replaces equals by equals". Here are just three examples:

- "Copy propagation" is a special case of inlining a `let`-bound variable:

$$\texttt{let}\ v\texttt{=}w\ \texttt{in}\ e \quad \equiv \quad e[w/v]$$

  where v and w are variables. (It is a special case because the rule remains valid even if w is an arbitrary expression, which is certainly not true in an imperative language.)

- "Procedure inlining" is an example of beta reduction. If f is defined like this:

$$f\ \texttt{= \textbackslash}xy\texttt{->}b$$

  then an expression with a call of f, such as $(f\ a_1\ a_2)$ can be transfomed to $b[a_1/x, a_2/y]$.

- Lifting invariant expressions out of loops corresponds to a simple transformation called the "full laziness transformation" (Hughes [1983]; Peyton Jones & Lester [1991]).

This idea, of compilation by transformation, is not new (Appel [1992]; Fradet & Metayer [1991]; Kelsey [1989]), but it is particularly applicable in a non-strict language. Although non-strict semantics carries an implementation cost, it also means that transformation rules such as those above can be applied globally, and in a wholesale fashion

[1] The "STG" language was originally short for Spineless Tagless G-machine language, but in fact the language is entirely independent of the abstract machine model used to implement it.

without side conditions. This is not true of a strict language, let alone a language with side effects. (Suppose, for example that b above did not mention x, and that the evaluation of a_1 did not terminate. In a strict language the semantics of the original call is non-termination, while after inlining f it may not be.)

In short, we have tried to gain the maximum leverage from the purity of the language we are compiling. Indeed, we have gone further, and developed several new techniques to express within a purely-functional notation a number of matters that are more commonly left implicit. Expressing them in this way exposes them to code-improving transformations. In particular:

- We developed the idea of *unboxed data types* in order to expose the degree of evaluation of a value to the transformation system (Peyton Jones & Launchbury [1991]).
- We developed the STG language so that a number of usually implicit matters, such as just when a closure is constructed and whether it is updated, can be exposed (Peyton Jones [1992a]).
- We developed the use of *monads* to support input/output, calling C procedures with side effects, and incrementally-updatable arrays (Peyton Jones & Wadler [1993]). This formulation allows arbitrary transformation of side-effecting computations to be performed without affecting their meaning.

We have now completed an overview of the compiler. The rest of the paper gives more details about some selected aspects.

4 The Core language and the second-order lambda calculus

The Core language is meant to be just large enough to express efficiently the full range of Haskell programs, and no larger. The obvious choice for such a language is the lambda calculus, augmented with `let`, `letrec`, and `case` expressions.

However, a prime consideration for us is *to preserve type information through the entire compiler*, right up to the code generator, despite the wholesale transformation that we expected to be applied to the program. There are various ways in which type information is either desirable or essential to the later stages of compilation: for example, a higher-order strictness analyser may need accurate type information in order to construct correct fixed points.

The trouble is that program transformation involves type manipulation. Consider, for example, the function `compose`, whose type is

$$\texttt{compose} :: \forall \alpha\beta\gamma.(\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$$

The function might be defined like this in an untyped Core language:

```
compose = \f -> \g -> \x ->
          let { y = g x } in f y
```

(We have written the definition a little curiously, using a `let` expression, for reasons which will become apparent.) Now, suppose that we wished to unfold a particular call to `compose`, say `(compose show double v)` where `v` is an `Int`, `double` doubles it, and `show` converts the result to a `String`. The result of unfolding the call to `compose` is an instance of the body of `compose`, thus:

```
let { y = double v } in show y
```

Now, we want to be able to identify the type of every variable and sub-expression, so must calculate the type of `y`. In this case, it has type `Int`, but in another application of `compose` it may have a different type. All this is because its type in the body of `compose` itself is just a type variable, β. It is clear that in a polymorphic world it is insufficient merely to tag every variable of the original program with its type, because this information does not survive across program transformations. Indeed no other compiler known to us for a polymorphically-typed language preserves type information across arbitrary transformations.

What, then, is to be done? Clearly, the program must be decorated with type information in some way, and every program transformation must be sure to preserve it. Deciding exactly how to decorate the program, and how to maintain these decorations correct during transformation seemed rather difficult, until we realised that a well-developed off-the-shelf solution was available from computer science theory, namely the second-order lambda calculus!

The idea is that every polymorphic function, such as `compose` receives a type argument for each universally-quantified polymorphic variable in its type (α, β, and γ in the case of `compose`), and whenever a polymorphic function is called, it is passed extra type arguments to indicate the types to which its polymorphic type variables are to be instantiated. The definition of `compose` now becomes:

```
compose = /\a,b,c     ->
          \f::(b->c) ->
          \g::(a->b) ->
          \x::a      ->
          let { y::b = g x } in f y
```

The function takes three type arguments (`a`, `b` and `c`), as well as its value arguments `f`, `g` and `x`. The types of the latter can now be given explicitly, as can the type of the local variable `y`. A call of `compose` is now given three extra type arguments, which instantiate `a`, `b` and `c` just as the "normal" arguments instantiate `f`, `g` and `x`. For example, the call of `compose` we looked at earlier is now written like this:

```
compose Int Int String show double v
```

It is now simple to unfold this call, by instantiating the body of `compose` with the supplied arguments, to give the expression

```
let { y::Int = double v } in show y
```

Notice that the `let`-bound variable `y` is now automatically attributed the correct type.

In short, the second-order lambda calculus provides us with a well-founded notation in which to express and transform polymorphically-typed programs. The type inference pass produces a translated program in which the "extra" type abstractions and applications are made explicit.

5 Monads

Monads are an idea from category theory for which we have found a number of fruitful applications, both in writing the compiler itself, and to encapsulate side-effecting operations in the code being compiled (Moggi [1989]; Wadler [1990]). We focus here on the use of monads in the compiler.

5.1 Monads in the compiler

To take a particular example, here is the (slightly simplified) code from the compiler for type-checking an application:

```
tcExpr (Ap fun arg)
 = tcExpr fun  `thenTc` (\ fun_ty ->
   tcExpr arg  `thenTc` (\ arg_ty ->
   newTyVar    `thenTc` (\ res_ty ->

   unify fun_ty (arg_ty `arrow` res_ty)
             `thenTc` (\ () ->

   returnTc res_ty
   ))))
```

At an informal level this code should be legible even by readers with no experience of type inference. It can be read like this: "To typecheck an application `Ap fun arg`, first typecheck `fun`, inferring the type `fun_ty`, then typecheck `arg`, inferring the type `arg_ty`. Now invent a fresh type variable `res_ty`, and unify `fun_ty` with `arg_ty -> res_ty`. Finally, return `res_ty` as the type of the application."

The type of `tcExpr` is as follows:

```
tcExpr :: Expr -> TcM UniType
```

One might have though that `tcExpr` would have type `Expr -> UniType`, a function from expressions to types. but quite a lot of "plumbing" is required behind the scenes. Firstly, if there is a type error in the program, the type-inference process can *fail* when type-checking `fun` or `arg`, or during the unification step. Secondly, there must be a *unique-name supply* from which to manufacture a new type variable. Thirdly, there must be some way of collecting and displaying *error messages*. Fourthly, the unification process works by incrementally augmenting a *substitution* mapping type variables to types. All of these would usually be handled in an imperative language by side effects, or by some sort of exception-handling mechanism. We handle them all using a monad.

The `TcM` type constructor encapsulates all this plumbing. This is most easily seen by looking at the definition of `TcM`:

```
type TcM a =  Uniq -> Subst -> Maybe (a, Uniq, Subst)
data Maybe a = Nothing | Just a
```

That is, a value of type `TcM a` is a function that takes a unique-name supply and a substitution, and delivers either `Nothing` (indicating failure) or `Just v`, where `v` is a triple of: a value of type `a`, a depleted name supply, and an augmented substitution. (We have omitted the gathering of error messages from this definition of `TcM` to reduce clutter.)

5.2 Benefits of monadic programming

Programming using monads has a number of benefits. Firstly, *the plumbing is implicit*, which makes the program much easier to read and write.

Secondly, *because the plumbing is all encapsulated in one place, it is very easy to modify.* For example, well over a year after the type checker was working we added an error recovery mechanism, so that a single type error would not cause the entire type-checker to halt. It is now possible to recover from the error, and gather several accurate type error messages in one run of the compiler. This was achieved in a single afternoon's work, modifying very localised parts of the type checker.

6 Using C as a target language

We generate code in the C language as our primary target, rather than generating native code direct. In this way we gain "instant" portability, because C is implemented on such a wide variety of architectures, and we benefit directly from others' improvements in C code generation.

This approach, of using C as a "high-level assembler" has gained popularity recently (Bartlett [1989]; Miranda [1991]).[2] In particular, the work of Tarditi *et al* on compiling SML to C, developed independently and concurrently with ours, addresses essentially the same problems (Tarditi, Acharya & Lee [1992]).

One approach to compiling into C is to translate the entire program into a giant C procedure. This turns out to be utterly impractical. Not only do most C compilers have trouble in compiling million-line procedures, but the approach also defeats separate compilation.

Instead, we simply compile each extended basic block into a parameter-less C procedure, giving rise to a large number of easily-compiled procedures. The "registers" of the evaluation model, such as the heap pointer, stack pointer, environment pointer, and so on, are mapped to global C variables. Temporary variables are mapped to local C variables, whose scope is the procedure for that basic block. The latter are in turn usually mapped to registers by the compiler.

There is one major problem with this approach: we need to be able to jump to an extended basic block, *but since such a block is implemented as a C procedure, we can only call it.* Unfortunately, every such call would make C's return stack grow by one more word, which is certain to cause stack overflow.

We solve this problem using a neat, and completely portable, trick. Each parameterless function, representing an extended basic block, *returns* the code pointer to which it would like to jump, rather than *calling* it. The execution of the entire program is controlled by the following one-line "interpreter":

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

That is, `cont` is the address of the code block (that is, C function) to be executed next. The function to which it points is called, and returns the address of the next one, and so on. It turns out that this idea is actually very old, and that we only reinvented it. Like several other clever and oft-reinvented ideas, Steele seems to have been its inventor; he called it the "UUO handler" in his Rabbit compiler for Scheme (Steele [1978]).

6.1 Efficiency

The remaining big question is, of course, efficiency. What price is paid for going via C? If we were restricted to a completely standard C compiler, the answer would certainly be "considerable". But by exploiting some language extensions provided by the GNU C compiler, `gcc`, which is available on a wide range of architectures, *we get code that appears to be significantly better than we could generate using any hand-built code generator*, unless we were prepared to lavish an enormous amount of effort on it.

For each architecture for which we want fast code, we provide a header file defining a handful of C macros, that customise the compilation to that particular architecture. The two really important things to get right are these:

- *Register mapping.* `gcc` allows one to specify that a particular global variable is to be kept in a register, which allows us to keep (say) the heap pointer permanently in a register.
- *Jumps.* Rather than returning to the "interpreter" as discussed above, jumps are compiled to genuine jump machine instructions, using GNU C's inline assembly facility. This eliminates entirely the overhead of the "interpreter".

6.2 Debugging

We originally used C as our target to confer portability, and to reduce code-generation effort. The design decision had one major unexpected benefit: it made debugging the code generator much easier, in two particular ways:

- We made extensive use of the C-language `gdb` source-level debugger for stepping through compiled code.
- Since the "interpreter" is executed at every jump, it is easy to add code to record a trail of the most recent few dozen jumps, and to perform "hygiene checks" on the state of the system (eg does the stack contents look reasonable) at every jump. This simple technology often catches a heap corruption *as it happens* rather than millions of instructions later when it causes a crash.

It is difficult to convey the importance of this matter. Finding obscure code-generation errors has to be done, and can be enormously time consuming. Using C-level debugging together with regular hygiene-checking has made a huge contribution to the development of the code generator.

7 Profiling

Everyone knows that it is often possible to make substantial improvements in the performance of a program by changing only a small proportion of its code. The trick, of course, is to know just *which* part of the code to pay attention to. This question can be particularly difficult to answer in a lazy functional program, because the demand-driven evaluation mechanism leads to a rather

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

non-intutive execution order. Until recently, there have been essentially no profiling tools for lazy functional languages, but recent work by ourselves and by Runciman and Wakeling at York has begun to change the situation.

Runciman and Wakeling's space profiling tool displays a regular census of the contents of the heap, which can often lead directly to the discovery of a "space leak" (Runciman & Wakeling [1993]).

We have developed a related technique, called *cost centres*, which enables the *time* consumption of the program to be profiled as well as its *space* consumption (Sansom & Peyton Jones [1993]). Briefly, the idea is this. The programmer identifies the expressions against which costs should be accumulated with a call to `scc` ("set cost centre"); for example, the call

```
scc "sorting" (sort xs)
```

would attribute the costs of evaluating `(sort xs)` to the cost centre `"sorting"`. Only the costs of doing the sorting are so attributed; in particular, the costs of evaluating `xs` are not, even though its evaluation may occur interleaved with the sorting process.

Furthermore, *all* the costs of the call to `sort` are so attributed, including the costs of executing calls made by `sort` to separately-compiled library functions. This applies even if those library functions are also called from elsewhere, which is especially common in a polymorphic functional language: only the cost of the calls from `sort` are attributed to `"sorting"`.

The implementation is simple. Every heap-allocated "thunk" (that is, suspended computation) has an extra field which identifies the cost centre for the computation which allocated the thunk. The cost centre for the currently-executing computation is identified by a global register, which is stored in each thunk as the latter is constructed. When a thunk is evaluated the cost-centre register is saved, and its value is set from that in the thunk; when evaluation is complete the cost-centre register is restored to its previous value.

Profilers for lazy functional languages are in their infancy, but we regard them as extremely important for the widespread use of functional languages.

8 Generational garbage collection

Like all other systems for symbolic computation, our Haskell implementation is built on top of a garbage-collected heap. Over the last decade, *generational garbage collection* has become widespread in the symbolic computation community (Lieberman & Hewitt [1983]). Generational collectors exploit the fact that most objects die young, by dividing the heap into several generations, and collecting young generations more often than old ones.

Generational collectors behave badly if there are many write operations into old generations. Unfortunately, implementations of lazy functional languages tend to perform lots write operations, as they update suspended computations (or thunks) with their values, so it looks as if they might interact badly with generational collectors. Perhaps as a result, garbage collectors for lazy functional languages have tended to be a variant of the classic Baker two-space collector. Indeed, apart from Seward [1992] there is esentially no literature on generational garbage collection for lazy languages.

We have begun investigation of a variety of garbage-collection technology for lazy languages, including generational schemes. Some of our findings are surprising (to us at least), and we summarise them briefly here, to give their flavour. Fuller details can be found in Sansom & Peyton Jones [1992].

- With a little care it is possible to build a one-space inplace compacting garbage collector that is as efficient or better than a two-space collector, unless the live data is a very small fraction (about 20%) of the heap size. A one-space collector has the big advantage that it can allow a heap almost twice as large as that of a two-space collector before the paging system starts to thrash.
- Most objects die even younger than in Lisp systems. For example, typically 85-95% of objects die before a further 100 kbytes of heap have been allocated.
- Similarly, objects are also updated young — typically more than 95% of all updates are for objects which are less than 100 kbytes old.
- A large majority (around 75%) of all updates are unnecessary; that is, the thunk which is updated is never again referenced. This indicates substantial scope for program analyses that detect and avoid unnecessary updates.
- Even with the simple generational schemes we have implemented so far, and even without considering paging effects, the generational garbage collecter outperforms both one-space and two-space collectors when the live data exceeds about 40% of the heap size.
- When paging is taken into account, the improved locality of the generational collector makes it degrade much more gracefully as the heap size is increased.

In short, contrary to popular belief, it seems that lazy functional language implementations are rather good subjects for generational garbage collection.

9 Benchmarks and test suites

The lack of a standard language has long hindered the development of a serious benchmark suite for lazy functional languages. There are a number of obvious reasons why such a suite is desirable:

- It encourages implementors to improve aspects of their compiler that give benefits to "real" applications, rather than merely improving the performance of "toy" programs.
- It give a concrete basis for comparing different implementations.

As part of our work we have developed the `nofib` benchmark suite (Partain [1993]), a collection of application programs written by people other than ourselves, mostly with a view to getting a particular task done. The range of applications is wide, including, for example, a theorem prover, a particle-in-cell simulation, a solid geometry application and a strictness analyser. Their size ranges from a few hundred lines of Haskell to several thousand.

A quite separate question from that of benchmarking is that of whether a particular Haskell implementation is complete, or whether it correctly implements some of the more obscure corners of the language. We have also developed a test suite consisting of a large number of (mostly) tiny programs. Every time we found a bug in the compiler we added a new program to the test suite that shows up the bug.

Each night an automated system rebuilds the compiler from its source code, compiles the test suite and benchmark suite, and runs the resulting programs. This simple technology catches a large number of errors, and ensures that no bug can be inadvertently re-introduced.

This process is surprisingly compute- and disc-intensive. Every program is compiled and run with several different combinations of options (profiling enabled or not, optimisation enabled or not, and so on), and we usually retain the debugging symbol tables in the binaries. A full run takes the whole weekend for a 96 Mbyte Sparc 10 workstation, using the whole of a local 2 Gbyte disc.

10 Availability

The Glasgow Haskell Compiler, its profiling tools, its test suite, and the `nofib` benchmark suite are all available free by FTP. You can use anonymous FTP (username: `anonymous`; password: your e-mail address) to these hosts, where you will find things in `pub/haskell` and its subdirectories:

`animal.cs.chalmers.se`	129.16.225.66
`ftp.dcs.glasgow.ac.uk`	130.209.240.50
`nebula.cs.yale.edu`	128.36.13.1

11 Summary: successes and failures

What lessons have we learned from our experience? Here is a selection:

- The act of building a large system has forced us into addressing research issues that we could not otherwise have addressed. This has resulted in a collection of new ideas that are applicable far more widely than our implementation alone: unboxed values, monads, use of the second-order lambda calculus, the STG language, update analysis, and so on.
- The Haskell type-class system caused us an enormous amount of extra work (beyond simple Hindley-Milner types), and we are still far from satisfied with the efficiency of the resulting programs. The compiler technology required to recover an acceptably efficient implementation is very considerable.
- We consistently underestimated how long it would take us to do the job, largely because of the scale of the compiler. Apparently-small changes would cause a chain of effects all of which would have to be chased through before the compiler could be rebuilt.
- If we were beginning again we would almost certainly write a Core-language "lint" pass, to check the type consistency of Core programs. Several of our more obscure compiler bugs turned out to be caused by a Core transformation which failed to maintain type information properly.
- Our attempt to build the compiler in such a way that it can form a framework for other researcher's work has been quite successful. Even prior to the compiler's official release, researchers in Manchester, Imperial and Queen Mary & Westfield College are using it in just this way, as well as several PhD students at Glasgow. Their work has in turn directly benefitted ours. For example, the storage management system, the profiling technology and much of the program transformation system, have been contributed by students.

Acknowledgement

This project could not have been carried out without funding from SERC for the GRASP project (GR/F98444 and GR/F34671). Our partner project, FLARE, is jointly funded by DTI and SERC. We benefit enormously from interaction with the Functional Programming Group at

Glasgow, especially John Launchbury. Our gallant beta-site users, Denis Howe, Jon Hill and Julian Seward, also deserve particular thanks.

References

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

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

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

JL Hennessy & DA Patterson [Feb 1990], *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, ISBN 1-55860-069-8.

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

RJM Hughes [July 1983], "The design and implementation of programming languages," PhD thesis, Programming Research Group, Oxford.

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

H Lieberman & C Hewitt [June 1983], "A real-time garbage collector based on the lifetimes of objects," *CACM* 26, 419–429.

R Milner [June 1991], "Computer Science: The Core IT Research Discipline," Lab for the Foundations of Computer Science, Edinburgh.

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

E Moggi [June 1989], "Computational lambda calculus and monads," in *Logic in Computer Science, California*, IEEE.

WD Partain [1993], "The nofib Benchmark Suite of Haskell Programs," in *Functional Programming, Glasgow 1992*, J Launchbury, ed., Workshops in Computing, Springer Verlag.

SL Peyton Jones [Apr 1992a], "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine," *Journal of Functional Programming* 2, 127–202.

SL Peyton Jones [Autumn 1992b], "UK research in functional programming," *SERC Bulletin* 4.

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

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

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

C Runciman & D Wakeling [1993], "Profiling a compiler," in *Functional Programming, Glasgow 1992*, J Launchbury, ed., Workshops in Computing, Springer Verlag.

P Sansom & SL Peyton Jones [1993], "Profiling Lazy Fuctional Languages," in *Functional Programming, Glasgow 1992*, J Launchbury, ed., Workshops in Computing, Springer Verlag.

PM Sansom & SL Peyton Jones [Dec 1992], "Generations of lazy functional languages," Department of Computing Science, University of Glasgow.

J Seward [March 1992], "Generational garbage collection for lazy graph reduction," Department of Computer Science, University of Manchester.

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

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

PL Wadler [June 1990], "Comprehending monads," in *Proc ACM Conference on Lisp and Functional Programming, Nice*, ACM.

PL Wadler & S Blott [Jan 1989], "How to make ad-hoc polymorphism less ad hoc," in *Proc 16th ACM Symposium on Principles of Programming Languages, Austin, Texas*, ACM.