Compiling Haskell by program transformation: a report from the trenches

Simon L Peyton Jones
Department of Computing Science, University of Glasgow, G12 8QQ
Email: simonpj@dcs.gla.ac.uk. WWW: http://www.dcs.gla.ac.uk/~simonpj

January 31, 1996

Abstract

Many compilers do some of their work by means of correctness-preserving, and hopefully performance-improving, program transformations. The Glasgow Haskell Compiler (GHC) takes this idea of "compilation by transformation" as its war-cry, trying to express as much as possible of the compilation process in the form of program transformations.

This paper reports on our practical experience of the transformational approach to compilation, in the context of a substantial compiler.

The paper appears in the Proceedings of the European Symposium on Programming, Linkoping, April 1996.

1 Introduction

Using correctness-preserving transformations as a compiler optimisation is a well-established technique (Aho, Sethi & Ullman [1986]; Bacon, Graham & Sharp [1994]). In the functional programming area especially, the idea of compilation by transformation has received quite a bit of attention (Appel [1992]; Fradet & Metayer [1991]; Kelsey [1989]; Kelsey & Hudak [1989]; Kranz [1988]; Steele [1978]).

A transformational approach to compiler construction is attractive for two reasons:

- Each transformation can be implemented, verified, and tested separately. This leads to a more modular compiler design, in contrast to compilers that consist of a few huge passes each of which accomplishes a great deal.
- In any framework (transformational or otherwise) each optimisation often exposes new opportunities for other optimisations — the "cascade effect". This makes it difficult to decide *a priori* what the best order to apply them might be. In a transformational setting it is easy to "plug and play", by re-ordering transformations, applying them more than once, or trading compilation time for code quality by omitting some. It allows a late commitment to phase ordering.

This paper reports on our experience in applying transformational techniques in a particularly thorough-going way to the Glasgow Haskell Compiler (GHC) (Peyton Jones et al. [1993]), a compiler for the non-strict functional language Haskell (Hudak et al. [1992]). Among other

things this paper may serve as a useful jumping-off point, and annotated bibliography, for those interested in the compiler.

A pervasive theme is the close interplay between theory and practice, a particularly satisfying aspect of functional-language research.

2 Overview

Haskell is a non-strict, purely functional language. It is a relatively large language, with a rich syntax and type system, designed for full-scale application programming.

The overall structure of the compiler is conventional;

1. The front end parses the source, does scope analysis and type inference, and translates the program into a small intermediate language called the *Core language*. This latter stage is called *de-sugaring*.

2. The middle consists of a sequence of Core-to-Core transformations, and forms the subject of this paper.

3. The back end code-generates the resulting Core program into C, whence it is compiled to machine code (Peyton Jones [1992]).

To exploit the advantages of compilation by transformation mentioned above, we have worked particularly hard to move work out of the front and back ends — especially the latter — and re-express it in the form of a transformation. We have taken the "plug and play" idea to an extreme, allowing the sequence of transformation passes to be completely specified on the command line.

In practice, we find that transformations fall into two groups:

1. A large set of simple, local transformations (e.g. constant folding, beta reduction). These transformations are all implemented by a single relatively complex compiler pass that we call the *simplifier*. The complexity arises from the fact that the simplifier tries to perform as many transformations as possible during a single pass over the program, exploiting the "cascade effect". (It would be unreasonably inefficient to perform just one at a time, starting from the beginning each time.) Despite these efforts, the result of one simplifier pass often still contains opportunities for further simplifier transformations, so we apply the simplifier repeatedly until no further transformations occur (with a set maximum to avoid pathological behaviour).

2. A small set of complex, global transformations (e.g. strictness analysis, specialising overloaded functions), each of which is implemented as a separate pass. Most consist of an analysis phase, followed by a transformation pass that uses the analysis results to identify appropriate sites for the transformation. Many also rely on a subsequent pass of the simplifier to "clean up" the code they produce, thus avoiding the need to duplicate transformations already embodied in the simplifier.

Rather than give a superficial overview of everything, we focus in this paper on three aspects of our compiler that play a key role in compilation by transformation:

Program	*Prog*	→	$Bind_1$; ... ; $Bind_n$	$n \geq 1$
Binding	*Bind*	→	*var* = *Expr*	Non-recursive
		|	`rec` var_1 = $Expr_1$; ... ; var_n = $Expr_n$	Recursive $n \geq 1$
Expression	*Expr*	→	*Expr Atom*	Application
		|	*Expr ty*	Type application
		|	`\` $var_1 \ldots var_n$ `->` *Expr*	Lambda abstraction
		|	`/\` $tyvar_1 \ldots tyvar_n$ `->` *Expr*	Type abstraction
		|	`case` *Expr* `of {` *Alts* `}`	Case expression
		|	`let` *Bind* `in` *Expr*	Local definition
		|	*con* $var_1 \ldots var_n$	Constructor $n \geq 0$
		|	*prim* $var_1 \ldots var_n$	Primitive $n \geq 0$
		|	*Atom*	
Atoms	*Atom*	→	*var*	Variable
		|	*Literal*	Unboxed Object
Literals	*Literal*	→	*integer* | *float* | ...	
Alternatives	*Alts*	→	$Calt_1$; ... ; $Calt_n$; *Default*	$n \geq 0$
		|	$Lalt_1$; ... ; $Lalt_n$; *Default*	$n \geq 0$
Constr. alt	*Calt*	→	*Con* $var_1 \ldots var_n$ `->` *Expr*	$n \geq 0$
Literal alt	*Lalt*	→	*Literal* `->` *Expr*	
Default alt	*Default*	→	`NoDefault`	
		|	*var* `->` *Expr*	

Figure 1: Syntax of the Core language

- The Core language itself (Section 3).
- Two groups of transformations implemented by the simplifier, inlining and beta reduction (Section 4), and transformations involving `case` expressions (Section 5).
- One global transformation pass, the one that performs and exploits strictness analysis (Section 6).

We conclude with a brief enumeration of the other main transformations incorporated in GHC (Section 7), and a summary of the lessons we learned from our experience (Section 8).

3 The Core language

The Core language clearly plays a pivotal role. Its syntax is given in Figure 1, and consists essentially of the lambda calculus augmented with `let` and `case`.

Though we do not give explicit syntax for them here, the Core language includes algebraic data type declarations exactly as in any modern functional programming language. For example, in Haskell one might declare the type of trees thus:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

This declaration implicitly defines *constructors* `Leaf` and `Branch`, that are used to construct data values, and can be used in the pattern of a `case` alternative. Booleans, lists, and tuples are simply pre-declared algebraic data types:

```
data Boolean      = False | True
data List a       = Nil   | Cons a (List a)
data Tuple3 a b c = T3 a b c  -- One for each size of tuple
```

Throughout the paper we take a few liberties with the syntax: we allow ourselves infix operators (e.g. `E1 + E2`), and special syntax for lists (`[]` for `Nil` and infix `:` for `Cons`), and tuples (e.g. `(a,b,c)`). We allow multiple definitions in a single `let` expression to abbreviate a sequence of nested `let` expressions, and often use layout instead of curly brackets and semicolons to delimit `case` alternatives. We use an upper-case identifier, such as `E`, to denote an arbitrary expression.

3.1 The operational reading

The Core language is of course a functional language, and can be given the usual denotational semantics. However, *a Core program also has a direct operational interpretation.* If we are to reason about the usefulness of a transformation we must have some model for how much it costs to execute it, so an operational interpretation is very desirable.

The operational model for Core requires a garbage-collected *heap*. The heap contains:

- *Data values*, such as list cells, tuples, booleans, integers, and so on.
- *Function values*, such as `\x -> x+1` (the function that adds 1 to its argument).
- *Thunks* (or suspensions), that represent suspended (i.e. as yet unevaluated) values.

Thunks are the implementation mechanism for Haskell's non-strict semantics. For example, consider the Haskell expression `f (sin x) y`. Translated to Core the expression would look like this:

```
let v = sin x
in  f v y
```

The `let` allocates a thunk in the heap for `sin x` and then, when it subsequently calls `f`, passes a pointer to the thunk. The thunk records all the information needed to compute its body, `sin x` in this case, but it is not evaluated before the call. If `f` ever needs the value of `v` it will *force* the thunk which provokes the computation of `sin x`. When the thunk's evaluation is complete the thunk itself is *updated* (i.e. overwritten) with the now-computed value. If `f` needs the value of `v` again, the heap object now contains its value instead of the suspended computation. If `f` never needs `v` then the thunk is not evaluated at all.

The two most important operational intuitions about Core are as follows:

1. `let` *bindings (and only* `let` *bindings) perform heap allocation.* For example:

```
let v = sin x
in
let w = (p,q)
in
f v w
```

   Operationally, the first `let` allocates a thunk for `sin x`, and then evaluates the `let`'s body. This body consists of the second `let` expression, which allocates a pair `(p,q)` in the heap, and then evaluates its body in turn. This body consists of the call `f v w`, so the call is now made, passing pointers to the two newly-allocated objects.

   In our implementation, each allocated object (be it a thunk or a value) consists only of a code pointer together with a slot for each free variable of the right-hand side of the `let` binding. Only one object is allocated, regardless of the size of the right-hand side (older implementations of graph reduction do not have this property). We do not attempt to share environments between thunks (Appel [1992]; Kranz et al. [1986]).

2. `case` *expressions (and only* `case` *expressions) perform evaluation.* For example:

```
case x of
      []     -> 0
      (y:ys) -> y + g ys
```

   The operational understanding is "evaluate `x`, and then scrutinise it to see whether it is an empty list, `[]`, or a `Cons` cell of form `(y:ys)`, continuing execution with the appropriate alternative".

   `case` expressions subsume conditionals, of course. The Haskell expression `if C E1 E2` is de-sugared to

```
case C of {True  -> E1; False -> E2}
```

The syntax in Figure 1 requires that function arguments must be atoms[1] (that is, variables or literals), and now we can see why. If the language allowed us to write

```
f (sin x) (p,q)
```

the operational behaviour would still be exactly as described in (1) above, with a thunk and a pair allocated as before. The `let` form is simply more explicit. Furthermore, the `let` form gives us the opportunity of moving the binding for `v` elsewhere, if that turns out to be desirable, which the apparently-simpler form does not. Lastly, the `let` form is more economical, because many transformations on `let` expressions (concerning strictness, for example) would have to be duplicated for function arguments if the latter were non-atomic.

It is also important to note where atoms are *not* required. In particular, the scrutinee of a `case` expression is an arbitrary expression, not just an atom. For example, the following is quite legitimate:

[1] This syntax is becoming quite widely used (Ariola et al. [1995]; Flanagan et al. [1993]; Flanagan et al. [1993]; Launchbury [1993]; Peyton Jones [1992]).

```
case (reverse xs) of { ... }
```

Operationally, there is no need to build a thunk for `reverse xs` and then evaluate it; rather, we can simply save a return address and call `reverse xs`. Again, the operational model determines the syntax.

3.2 Polymorphism

Like any compiler for a strongly-typed language, GHC infers the type of every expression and variable. An obvious question is: can this type assignment be maintained through the translation to the Core language, and through all the subsequent transformations that are applied to the program? If so, both transformations and code generator might (and in GHC sometimes do) take advantage of type information to generate better code.

In a monomorphic language the answer is a clear "yes", but matters are not initially so clear in a polymorphic setting. The trouble is that program transformation involves type manipulation. Consider, for example, the usual composition function, `compose`, whose type is

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

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

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

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 we 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.

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 correctly during transformation, seemed rather difficult at first. We finally realised that an off-the-shelf solution was available, namely the second-order lambda calculus (Girard [1971]; Reynolds [1974]).

The idea is that every polymorphic function, such as `compose` has a type abstraction 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 parameters* (`a`, `b` and `c`), as well as its value parameters `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. It turns out to be easy to introduce the extra type abstractions and applications as part of the type inference process.

Other compilers for polymorphic languages are beginning to carry type information through to the back end, and use it to generate better code. Shao & Appel [1995] use type information to improve data representation, though the system they describe is monomorphic after the front end. Our implementation uses type abstractions and applications only to keep the compiler's types straight; no types are passed at runtime. It is possible to take the idea further, however, and pass types at runtime to specialise data representations (Morrison et al. [1991]), give fast access to polymorphic records (Ohori [1992]), guide garbage collection (Tolmach [1994]). The most recent and sophisticated work is Harper & Morrisett [1995].

4 Inlining and beta reduction

Functional programs often consist of a myriad of small functions — functional programmers treat functions the way C programmers treat macros — so good inlining is crucial. Compilers for conventional languages get 10-15% performance improvement from inlining (Davidson & Holler [1988]), while functional language compilers gain 20-40%[2] (Appel [1992]; Santos [1995]). Inlining removes some function-call overhead, of course, but an equally important factor is that inlining brings together code that was previously separated, and thereby often exposes a cascade of new transformation opportunities. We therefore implement inlining in the simplifier.

We have found it useful to identify three distinct transformations related to inlining:

Inlining itself replaces an occurrence of a `let`-bound variable by (a copy of) the right-hand side of its definition. Notice that inlining is not limited to function definitions; any `let`-bound variable can potentially be inlined. (Remember, though, that occurrences of a variable in an argument position are not candidates for inlining, because they are constrained to be atomic.)

Dead code elimination discards `let` bindings that are no longer used; this usually occurs when all occurrences of a variable have been inlined.

[2] This difference may soon decrease as the increased use of object-oriented languages leads to finer-gained procedures (Calder, Grunwald & Zorn [1994]).

Beta reduction replaces `(\x->E) A` by `E[A/x]`. (An analogous transformation deals with type applications.)

Beta reduction is particularly simple in our setting. Since the argument `A` is bound to be atomic, there is no risk of duplicating a redex, and we can simply replace `x` by `A` throughout `E`. There is a worry about name capture, however: what if `A` is also bound in `E`? We avoid this problem by the simple expedient of renaming every identifier as we go, which costs little extra since we have to construct a new, transformed expression anyway. Whilst beta reduction is simple, inlining is more interesting.

4.1 Simple inlining

It is useful to distinguish two cases of inlining:

WHNFs. If the variable concerned is bound to a *weak head normal form (WHNF)* — that is, an atom, lambda abstraction or constructor application — then it can be inlined without risking the duplication of work. The only down-side might be an increase in code size.

Non-WHNFs. Otherwise, inlining carries the risk of loss of sharing and hence the duplication of work. For example,

```
let x = f 100 in ...x...x...
```

it might be be unwise to inline `x`, because then `f 100` would be evaluated twice instead of once. Informally, we say that a transformation is $\mathcal{W}$*-safe* if it guarantees not to duplicate work.

In the case of WHNFs everything is as one would expect. The trade-off is between code size and the benefit of inlining and, like any compiler, we have a variety of heuristics (but no formal analysis) for deciding when a function is "small enough" to inline. Many functions are "small", though, and code size can actually *decrease* when they are inlined, both because the calling code is eliminated, and also because of other consequential transformations that are exposed.

The other sorts of WHNF, an atom or constructor application, is always small enough to inline. (Recall that constructor applications must have atomic arguments.)

For non-WHNFs, attention focuses on how the variable is used. If the variable occurs just once, then presumably it is safe to inline it. Our first approach was to perform a simple occurrence analysis that records for each variable how many places it is used, and use this information to guide the inlinings done by the simplifier. There are three complications with this naive approach.

The first is practical. As mentioned earlier, the simplifier tries to perform as many transformations as possible during a single pass over the program. However, many transformations (notably beta reduction and inlining itself) change the number of occurrences of a variable. Our current solution to this problem is to do a great deal of book-keeping to keep occurrence information up to date. (Appel & Jim [1996] does something similar.)

The second complication is that a variable may occur multiple times with no risk of duplicating work, namely if the occurrences are in different alternatives of a `case` expression. In this case, the only issue to consider is the tradeoff between code size and inlining benefit.

Lastly, inlining based on naive occurrence counting is not $\mathcal{W}$-safe! Consider this expression:

```
let x = f 100
    g = \y -> ...x...
in ...(g a)...(g b)...
```

If we replace the single occurrence of `x` by `(f 100)` we will recompute the call to `f` every time `g` is called, rather than sharing it among all calls to `g`. Our current solution is conservative: we never inline inside a lambda abstraction. It turns out, though, that this approach is sometimes too conservative. In higher-order programs where lots of inlining is happening, it is not unusual to find functions that are sure to be called only once, so it would be perfectly safe to inline inside them.

4.2 Using linearity

Because of these complications, the book-keeping required to track occurrence information has gradually grown into the most intricate and bug-prone part of the simplifier. Worse, work-duplication bugs manifest themselves only as performance problems, and may go unnoticed for a long time[3]. This complexity is especially irritating because we have a strong intuitive notion of whether a variable can be "used more than once", and that intuitive notion is an invariant of $\mathcal{W}$-safe transformations. That suggests that a *linear type system* would be a good way to identify variables that can safely be inlined, even though they occur inside lambdas, or that cannot safely be inlined even though they (currently) occur only once. Just as all transformations preserve the ordinary typing of an expression (Section 3.2) so $\mathcal{W}$-safe transformations preserve the linear type information too, and hence guarantee not to duplicate work.

Unfortunately, most linear type systems are inappropriate because they do not take account of call-by-need evaluation. For example, consider the expression

```
let x = 3*4
    y = x+1
in y + y
```

Under call by need evaluation, even though `y` is evaluated many times, `x` will be evaluated only once. Most linear systems would be too conservative, and would attribute a non-linear type to `x` as well as `y`, preventing `x` from being inlined.

Thus motivated, we have developed a linear type system that *does* take account of call by need evaluation (Wadler & Turner [1995]). The type system assigns a type of $\texttt{Int}^{\omega}$ to `y` in the above example, the superscript ω indicating that `y` might be evaluated more than once. However, it assigns a type of $\texttt{Int}^{1}$ to `x`, indicating that `x` can be evaluated at most once, and hence can $\mathcal{W}$-safely be inlined.

The type system is capable of dealing with "usage polymorphism". For example, consider

[3] One such bug caused the compiler, which is of course written in Haskell, to rebuild its symbol table from scratch every time a variable was looked up in the table. The compiler worked perfectly, albeit somewhat slowly, and it was months before we noticed (Sansom [1994])!

this definition of `apply`:

```
apply f x = f x
```

In a particular application `(apply g y)`, whether or not `y` is used more than once depends on whether `g` uses its argument more than once. So the type of `apply` is[4]

$$\forall u, v. \forall \alpha, \beta. (\alpha^u \to \beta^v) \to \alpha^u \to \beta^v$$

The two occurrences of α^u indicate that the usage u of `g`'s argument is the same as that of `y`.

Our implementation of this linear type system is incomplete, so we do not yet have practical experience of its utility, but we are optimistic that it will provide a systematic way of addressing an area we have only dealt with informally to date, and which has bitten us badly more than once.

5 Transforming conditionals

Most compilers have special rules to optimise conditionals. For example, consider the expression

```
if (not x) then E1 else E2
```

No decent compiler would actually negate the value of `x` at runtime! Let us see, then, what happens if we simply turn the transformation handle. After de-sugaring the conditional, and inlining the definition of `not`, we get

```
case (case x of {True -> False; False -> True}) of
      True  -> E1
      False -> E2
```

Here, the outer `case` scrutinises the value returned by the inner `case`. This observation suggests that we could move the outer `case` inside the the branches of the inner one, thus:

```
case x of
      True  -> case False of {True -> E1; False -> E2}
      False -> case True  of {True -> E1; False -> E2}
```

Notice that the originally-outer `case` expression has been duplicated, but each copy is now scrutinising a known value, and so we can make the obvious simplification to get exactly what we might originally have hoped:

```
case x of
      True  -> E2
      False -> E1
```

Both of these transformations are generally applicable. The second, the case-of-known-constructor transformation, eliminates a `case` expression that scrutinises a known value. This is always a Good Thing, and many other transformations are aimed at exposing opportunities for such `case` elimination. We consider another useful variant of `case` elimination in Section 5.3. The first, which we call the case-of-case transformation, is certainly correct in general, but it appears to risk duplicating `E1` and/or `E2`. We turn to this question next.

[4] In fact, for the purposes of this paper we have simplified the type a little.

5.1 Join points

How can we gain the benefits of the case-of-case transformation without risking code duplication? A simple idea is to make local definitions for the right-hand sides of the outer `case`, like this:

```
case (case S of {True -> R1; False -> R2}) of
      True  -> E1
      False -> E2
```

$\Longrightarrow$

```
let e1 = E1; e2 = E2
in case S of
      True  -> case R1 of {True -> e1; False -> e2}
      False -> case R2 of {True -> e1; False -> e2}
```

Now `E1` and `E2` are not duplicated, though we incur instead the cost of implementing the bindings for `e1` and `e2`. In the `not` example, though, the two inner `case`s are eliminated, leaving only a single occurrence of each of `e1` and `e2`, so their definitions will be inlined leaving exactly the same result as before.

We certainly cannot guarantee that the newly-introduced bindings will be eliminated, though. Consider, for example, the expression:

```
if (x || y) then E1 else E2
```

Here, `||` is the boolean disjunction operation, defined thus:

```
|| = \a b -> case a of {True  -> True; False -> b}
```

De-sugaring the conditional and inlining `||` gives:

```
case (case x of {True -> True; False -> y}) of
      True  -> E1
      False -> E2
```

Now applying the (new) case-of-case transformation:

```
let e1 = E1 ; e2 = E2
in case x of
      True  -> case True of {True -> e1; False -> e2}
      False -> case y    of {True -> e1; False -> e2}
```

Unlike the `not` example, only one of the two inner `case`s simplifies, so only `e2` will certainly be inlined, because `e1` is still mentioned twice:

```
let e1 = E1
in case x of
      True  -> e1
      False -> case y of {True -> e1; False -> E2}
```

The interesting thing here is that `e1` plays exactly the role of a label in conventional compiler technology. Given the original conditional, a C compiler will "short-circuit" the evaluation of the condition if `x` turns out to be `True` generating code like:

```
    if (x) {goto l1};
    if (y) {goto l1};
```

```
    goto l2;
l1: ...code for E1...; goto l3
l2: ...code for E2...
l3: ...
```

Here, `l1` is a label where two possible execution paths (if `x` is `True` or if `x` is `False` and `y` is `True`) join up; we call it a "join point". That suggests in turn that our code generator should be able to implement the binding for `e1`, not by allocating a thunk as it would usually do, but rather by simply jumping to some common code (after perhaps adjusting the stack pointer) wherever `e1` is subsequently evaluated. Our compiler does exactly this. Rather than somehow mark `e1` as special, the code generator does a simple syntactic escape analysis to identify variables whose evaluation is certain to take place before the stack retreats, and implements their evaluation as a simple adjust-stack-and-jump. As a result we get essentially the same code as a C compiler for our conditional.

Seen in this light, the act of inlining `E2` is what a conventional compiler might call "jump elimination". A good C compiler would probably eliminate the jump to `l2` thus:

```
    if (x) {goto l1};
    if (y) {goto l1};
l2: ...code for E2...
l3: ...
l1: ...code for E1...; goto l3
```

Back in the functional world, if `E1` is small then the inliner might decide to inline `e1` at its two occurrences regardless, thus eliminating a jump in favour of a slight increase in code size. Conventional compilers do this too, notably in the case where the code at the destination of a jump is just another jump, which would correspond in our setting to `E1` being just a simple variable.

The point is not that the transformations achieve anything that conventional compiler technology does not, but rather that a single mechanism (inlining), which is needed anyway, deals uniformly with jump elimination as well as its more conventional effects.

5.2 Generalising join points

Does all this work generalise to data types other than booleans? At first one might think the answer is "yes, of course", but in fact the modified case-of-case transformation is simply nonsense if the originally-outer `case` expression binds any variables. For example, consider the expression

```
f (if b then B1 else B2)
```

where `f` is defined thus:

```
f = \as -> case as of {[] -> E1; (b:bs) -> E2}
```

De-sugaring the `if` and inlining `f` gives:

```
case (case b of {True -> B1; False -> B2}) of
      []     -> E1
      (b:bs) -> E2
```

But now, since `E2` may mention `b` and `bs` we cannot `let`-bind a new variable `e2` as we did before! The solution is simple, though: simply `let`-bind a *function* `e2` that takes `b` and/or `bs` as its arguments. Suppose, for example, that `E2` mentions `bs` but not `b`. Then we can perform a case-of-case transformation thus:

```
let e1 = E1; e2 = \bs -> E2
in case b of
      True  -> case B1 of {[] -> e1; (b:bs) -> e2 bs}
      False -> case B2 of {[] -> e1; (b:bs) -> e2 bs}
```

All the inlining mechanism discussed above for eliminating the binding for `e2` if possible works just as before. Furthermore, even if `e2` is not inlined, the code generator can still implement `e2` efficiently: a call to `e2` is compiled to a code sequence that loads `bs` into a register, adjusts the stack pointer, and jumps to the join point.

This goes beyond what conventional compiler technology achieves. Our join points can now be parameterised by arguments that embody the differences between the execution paths that led to that point. Better still, the whole setup works for arbitrary user-defined data types, not simply for booleans and lists.

5.3 Generalising `case` elimination

Earlier, we discussed the case-of-known-constructor transformation that eliminates a `case` expression. There is a useful variant of this transformation that also eliminates a `case` expression. Consider the expression:

```
if null xs then r else tail xs
```

where `null` and `tail` are defined as you might expect:

```
null = \as -> case as of {[] -> True; (b:bs) -> False}
tail = \cs -> case cs of {[] -> error "tail"; (d:ds) -> ds}
```

After the usual inlining we get:

```
case (case xs of {[] -> True; (b:bs) -> False}) of
      True  -> r
      False -> case xs of
                     []     -> error "tail"
                     (d:ds) -> ds
```

Now we can do the case-of-case transformation as usual, giving after a few extra steps:

```
case xs of
      []     -> r
      (b:bs) -> case xs of
                     []     -> error "tail"
                     (d:ds) -> ds
```

Now, it is obvious that the inner evaluation of `xs` is redundant, because in the `(b:bs)` branch of the outer `case` we know that `xs` is certainly of the form `(b:bs)`! Hence we can eliminate the inner `case`, selecting the `(d:ds)` alternative, but substituting `b` for `d` and `bs` for `ds`:

```
case xs of
      []     -> r
```

```
(b:bs) -> bs
```

We will see another application of this form of `case` elimination in Section 6.1.

5.4 Summary

We have described a few of the most important transformations involving `case` expressions, but there are quite a few more, including case merging, dead alternative elimination, and default elimination. They are described in more detail by Santos [1995] who also provides measurements of their frequency.

Like many good ideas, the case-of-case transformation — limited to booleans, but including the idea of using `let`-bound variables as join points — was incorporated in Steele's Rabbit compiler for Scheme (Steele [1978]). We re-invented it, and generalised it for `case` expressions and parameterised join points. `let`-bound join points are also extremely useful when desugaring complex pattern matching. Lacking join points, most of the standard descriptions are complicated by a special FAIL value, along with special semantics and compilation rules, to express the "joining up" of several execution paths when a pattern fails to match (Augustsson [1987]; Peyton Jones [1987]).

6 Unboxed data types and strictness analysis

Consider the expression `x+y`, where `x` and `y` have type `Int`. Because Core is non-strict, `x` and `y` must each be represented by a pointer to a possibly-unevaluated object. Even if `x`, say, is already evaluated, it will still therefore be represented by a pointer to a "boxed" value in the heap. The addition operation must evaluate `x` and `y` as necessary, unbox them, add them, and box the result.

Where arithmetic operations are cascaded we would like to avoid boxing the result of one operation only to unbox it immediately in the next. Similarly, in the expression `x+x` we would like to avoid evaluating and unboxing `x` twice.

6.1 Exposing boxing to transformation

Such boxing/unboxing optimisations are usually carried out by the code generator, but it would be better to find a way to express them as program transformations. We have achieved this goal as follows. Instead of regarding the data types `Int`, `Float` and so on as primitive, we define them using algebraic data type declarations:

```
data Int   = I# Int#
data Float = F# Float#
```

Here, `Int#` is the truly-primitive type of unboxed integers, and `Float#` is the type of unboxed floats. The constructors `I#` and `F#` are, in effect, the boxing operations. (The `#` characters are merely cues to the human reader; the compiler treats `#` as part of a name, like any other letter.) Now we can express the previously-primitive `+` operation thus:

```
+ = \a b -> case a of
              I# a# -> case b of
```

```
                    I# b# -> case a# +# b# of
                                r# -> I# r#
```

where `+#` is the primitive addition operation on unboxed values. You can read this definition as "evaluate and unbox `a`, do the same to `y`, add the unboxed values giving `r#`, and return a boxed version thereof".

Now, simple transformations do the Right Thing to `x+x`. We begin by inlining `+` to give:

```
case x of
 I# a# -> case x of
           I# b# -> case a# +# b# of
                      r# -> I# r#
```

But now the inner `case` can be eliminated (Section 5.3), since it is scrutinising a known value, `x`, giving the desired outcome:

```
case x of
 I# a# -> case a# +# a# of
           r# -> I# r#
```

Similar transformations (this time involving case-of-case) ensure that in expressions such as `(x+y)*z` the intermediate result is never boxed. The details are given by Peyton Jones & Launchbury [1991], but the important points are these:

- By making the Core language somewhat more expressive (i.e. adding unboxed data types) we can expose many new evaluation and boxing operations to program transformation.
- Rather than a few *ad hoc* optimisations in the code generator, the full range of transformations can now be applied to the newly-exposed code.
- Optimising evaluation and unboxing may itself expose new transformation opportunities; for example, a function body may become small enough to inline.

6.2 Strictness analysis

Strictness analysers attempt to figure out whether a function is sure to evaluate its argument, giving the opportunity for the compiler to evaluate the argument before the call, instead of building a thunk that is forced later on. There is an enormous literature on strictness analysis itself, *but virtually none explaining how to exploit its results*, apart from general remarks that the code generator can use it. Our approach is to express the results of strictness analysis as a program transformation, for exactly the reasons mentioned at the end of the previous section.

As an example, consider the factorial function with an accumulating parameter, which in Haskell might look like this:

```
afac :: Int -> Int -> Int
afac a 0 = a
afac a n = afac (n*a) (n-1)
```

Translated into the Core language, it would take the following form:

```
one = I# 1#
afac = \a n ->  case n of
                  I# n# -> case n# of
                             0#  -> a
                             n#' -> let a' = n*a; n' = n-one
                                    in afac a' n'
```

In a naive implementation this function sadly uses linear space to hold a growing chain of unevaluated thunks for `a'`.

Now, suppose that the strictness analyser discovers that `afac` is strict in both its arguments. Based on this information we split it into two functions, a *wrapper* and a *worker* thus:

```
afac = \a n -> case a of I# a# -> case n of I# n# -> afac# a# n#

one = I# 1#
afac# = \a# n# -> let n = I# n#; a = I# a#
                  in case n of
                      I# n# -> case n# of
                                 0#  -> a
                                 n#' -> let a' = n*a; n' = n-one
                                        in afac a' n'
```

The wrapper, `afac`, implements the original function by evaluating the strict arguments and passing them unboxed to the worker, `afac#`. The wrapper is also marked as "always-inline-me", which makes the simplifier extremely keen to inline it at every call site, thereby effectively moving the argument evaluation to the call site.

The code for the worker starts by reconstructing the original arguments in boxed form, and then concludes with the *original unchanged* code for `afac`. Re-boxing the arguments may be correct, but it looks like a weird thing to do because the whole point was to avoid boxing the arguments at all! Nevertheless, let us see what happens when the simplifier goes to work on `afac#`. It just inlines the definitions of `*`, `-`, *and* `afac` *itself*; and applies the transformations described earlier. A few moments work should convince you that the result is this:

```
afac# = \a# n# -> case n# of
                    0#  -> I# a#
                    n'# -> case (n# *# a#) of
                             a1# -> case (n# -# 1#) of
                                      n1# -> afac# a1# n1#
```

Bingo! `afac#` is just what we hoped for: a strict, constant-space, efficient factorial function. The reboxing bindings have vanished, because a `case` elimination transformation has left them as dead code. Even the recursive call is made directly to `afac#`, rather than going via `afac` — it is worth noticing the importance of inlining the wrapper in the body of the worker, even though the two are mutually recursive. Meanwhile, the wrapper `afac` acts as an "impedance-matcher" to provide a boxed interface to `afac#`.

6.3 Data structures

We have found it very worthwhile to extend the strictness analyser a bit further. Suppose we have the following function definition:

```
f :: (Int,Int) -> Int
f = \p -> E
```

It is relatively easy for the strictness analyser to discover not only `f`'s strictness in the pair `p`, but also `f`'s strictness in the two components of the pair. For example, suppose that the strictness analyser discovers that `f` is strict both in `p` and in the first component of `p`, but not in the second. Given this information we can transform the definition of `f` into a worker and a wrapper like this,

```
f = \p -> case p of (x,y) -> case x of I# x# -> f# x# y

f# = \x# y -> let x = I# x#; p = (x,y)
              in E
```

The pair is passed to the worker unboxed (i.e. the two components are passed separately), and so is the first component of the pair.

We soon learned that looking inside (non-recursive) data structures in this way exposed a new opportunity: absence analysis. What if `f` does not use the second component of the pair at all? Then it is a complete waste of time to pass `y` to `f#` at all. Whilst it is unusual for programmers to write functions with arguments that are completely unused, it is rather common for them to write functions that do not use some parts of their arguments. We therefore perform both strictness analysis and absence analysis, and use the combined information to guide the worker/wrapper split.

Matters are more complicated if the argument type is recursive or has more than one constructor. In these cases we are content simply to evaluate the argument before the call, as described in the next section.

Notice the importance of type information to the whole endeavour. The type of a function guides the "resolution" of the strictness analysis, and the worker/wrapper splitting.

6.4 Strict `let` bindings

An important, but less commonly discussed, outcome of strictness analysis is that it is possible to tell whether a `let` binding is strict; that is, whether the variable bound by the `let` is sure to be evaluated in the body. If so there is no need to build a thunk. Consider the expression:

```
let x = R in E
```

where `x` has type `Int`, and `E` is strict in `x`. Using a similar strategy to the worker/wrapper scheme, we can transform to

```
case R of { I# x# -> let x = I# x# in E }
```

As before, the reboxing binding for `x` will be eliminated by subsequent transformation. If `x` has a recursive or multi-constructor type then we transform instead to this:

```
case R of { x -> E }
```

This expression simply generates code to evaluate `R`, bind the (boxed) result to `x` and then evaluate `E`. This is still an improvement over the original `let` expression because no thunk is built.

6.5 Summary

Strictness analysis, exploited via unboxed data types, is a very worth while analysis and transformation. Even the relatively simple analyser we use improves execution time by 10–20% averaged across a wide range of programs (Peyton Jones & Partain [1993]).

7 Other GHC transformations

We have focused so far on three particular aspects of GHC's transformation system. This section briefly summarises the other main transformations performed by GHC:

The simplifier contains many more transformations than those described in Sections 4 and 5. A full list can be found in Peyton Jones & Santos [1994] and Santos [1995]; the latter also contains measurements of the frequency and usefulness of each transformation.

The specialiser uses partial evaluation to create specialised versions of overloaded functions.

Let-floating is a group of transformations that concern the placement of `let` bindings, and hence determine where allocation occurs. There are three main let-floating transformations:

- *Floating inwards* moves bindings as near their site of use as possible.
- The *full laziness* transformation floats constant sub-expressions out of lambda abstractions (Hughes [1983]; Peyton Jones & Lester [1991]); it generalises the standard idea of loop-invariant code motion (Aho, Sethi & Ullman [1986]).
- *Local let-floating* fine-tunes the location of each `let` binding.

Details of all three are given by Peyton Jones, Partain & Santos [1996], along with detailed measurements. Let-floating alone gives an average improvement in execution time of around 15%.

Eta expansion is an unexpectedly-useful transformation (Gill [1996, Chapter 4]). We found that other transformations sometimes produce expressions of the form:

```
let f = \x -> let ... in \y -> E
in B
```

If `f` is always applied to two arguments in `B`, then we can $\mathcal{W}$-safely – that is, without risk of duplicating work — transform the expression to:

```
let f = \x y -> let ... in E
in B
```

(It turns out that a lambda abstraction that binds multiple arguments can be implemented much more efficiently than a nested series of lambdas.) The most elegant way to achieve the transformation is to perform an eta-expansion — the opposite of eta reduction — on `f`'s right hand side:

```
\x -> R      ⟹      \x a -> R a
```

Once that is done, normal beta reduction will make the application to `a` "cancel" with the `\y`, to give the desired overall effect.

The crucial question is this: when is eta expansion guaranteed to be $\mathcal{W}$-safe? Unsurprisingly, this turns out to be another fruitful application for the linear type system sketched in Section 4.2.

Deforestation is a transformation that removes intermediate lists (Wadler [1990]). For example, in the expression `sum (map double xs)` an intermediate list `(map double xs)` is created, only to be consumed immediately by `sum`. Successful deforestation removes this intermediate list, giving a single pass algorithm that traverses the list `xs`, doubling each element before adding it to the total.

Full-blown Wadler-style deforestation for higher-order programs is difficult; the only example we know of is described by Marlow [1996] and even that does not work for large programs. Instead, we developed a new, more practical, technique called *short cut deforestation* (Gill, Launchbury & Peyton Jones [1993]). As the name implies, our method does not remove all intermediate lists, but in exchange it is relatively easy to implement. Gill [1996] describes the technique in detail, and gives measurements of its effectiveness. Even on programs written without deforestation in mind the transformation reduces execution time by some 3% averaged over a range of programs.

Lambda lifting is a well-known transformation that replaces local function declarations with global ones, by adding their free variables as extra parameters (Johnsson [1985]). For example, consider the definition

```
f = \x -> letrec g = \y -> ...x...y...g...
          in ...g...
```

Here, `x` is free in the definition of `g`. By adding `x` as an extra argument to `g` we can transform the definition to:

```
f  = \x -> ...(g' x)...
g' = \x y -> ...x...y...(g' x)...
```

Some back ends require lambda-lifted programs. Our code generator can handle local functions directly, so lambda lifting is not required. Even so, it turns out that lambda lifting is sometimes beneficial, *but on other occasions the reverse is the case.* That is, the exact opposite of lambda lifting — lambda dropping, also known as the static argument transformation — sometimes improves performance. Santos [1995, Chapter 7] discusses the tradeoff in detail. GHC implements both lambda lifting and the static argument transformation. Each buys only a small performance gain (a percentage point or two) on average.

The "average" performance improvements mentioned in this paper are geometric means taken over the large `nofib` suite of benchmark programs, many of which are real applications (Partain [1993]). They are emphatically *not* best-case results on toy programs! Nevertheless, they should be taken only as a crude summary of the general scale of the effect; the papers cited give much more detail.

8 Lessons and conclusions

What general lessons about compilation by transformation have we learned from our experience?

The interaction of theory and practice is genuine, not simply window dressing. Apart from aspects already mentioned — second order lambda calculus, linear type systems, strictness and absence analysis — here are three other examples described elsewhere:

- We make extensive use of *monads* (Wadler [1992]), particularly to express input/output (Peyton Jones & Wadler [1993]) and stateful computation (Launchbury & Peyton Jones [1994]).
- Parametricity, a deep semantic consequence of polymorphism, turns out to be crucial in establishing the correctness of cheap deforestation (Gill, Launchbury & Peyton Jones [1993]), and secure encapsulation of stateful computation (Launchbury & Peyton Jones [1994]).
- GHC's time and space profiler is based on a formal model of cost attribution (Sansom [1994]; Sansom & Peyton Jones [1995]), an unusual property for a highly operational activity such as profiling. In this case the implementation came first, but the subtleties caused by non-strictness and higher-order functions practically drove us to despair, and forced us to develop a formal foundation.

Plug and play really works. The modular nature of a transformational compiler, and its late commitment to the order of transformation, is a big win. The ability to run a transformation pass twice (at least when going for maximum optimisation) is sometimes very useful.

The "cascade effect" is important. One transformation really does expose opportunities for another. Transformational passes are easier to write in the knowledge that subsequent transformations can be relied on to "clean up" the result of a transformation. For example, a transformation that wants to substitute `x` for `y` in an expression `E` can simply produce `(\y->E) x`, leaving the simplifier to perform the substitution later.

The compiler needs a lot of bullets in its gun. It is common for one particular transformation to have a dramatic effect on a few programs, and a very modest effect on most others. There is no substitute for applying a large number of transformations, each of which will "hit" some programs.

Some non-obvious transformations are important. We found that it was important to add a significant number of obviously-correct transformations that would never apply

directly to any reasonable source program. For example:

```
case (error "Wurble") of { ... }    ⟹    error "Wurble"
```

(`error` is a function that prints its argument string and halts execution. Semantically its value is just bottom.) No programmer would write a case expression that scrutinises a call to `error`, but such `case` expressions certainly show up after transformation. For example, consider the expression

```
if head xs then E1 else E2
```

After de-sugaring, and inlining `head` we get:

```
case (case xs of { [] -> error "head"; p:ps -> p } of
  True  -> E1
  False -> E2
```

Applying the case-of-case transformation (Section 5) makes (one copy of) the outer `case` scrutinise the call to `error`.

Other examples of non-obvious transformations include eta expansion (Section 7) and absence analysis (Section 6.3). We identified these extra transformations by eye-balling the code produced by the transformation system, looking for code that could be improved.

Elegant generalisations of traditional optimisations have often cropped up, that either extend the "reach" of the optimisation, or express it as a special case of some other transformation that is already required. Examples include jump elimination, copy propagation, boolean short-circuiting, and loop-invariant code motion. Similar generalisations are discussed by Steele [1978].

Maintaining types is a big win. It is sometimes tiresome, but never difficult, for each transformation to maintain type correctness. On the other hand it is sometimes indispensable to know the type of an expression, notably during strictness analysis.

Perhaps the largest single benefit came from an unexpected quarter: it is very easy to check a Core program for type correctness. While developing the compiler we run "Core Lint" (the Core type-checker) after every transformation pass, which turns out to be an outstandingly good way to detect incorrect transformations. Before we used Core Lint, bogus transformations usually led to a core dump when running the transformed program, followed by a long `gdb` hunt to isolate the cause. Now most bogus transformations are identified much earlier, and much more precisely. One of the dumbest things we did was to delay writing Core Lint.

Cross-module optimisation is important. Functional programmers make heavy use of libraries, abstract data types, and modules. It is essential that inlining, strictness analysis, specialisation, and so on, work between modules. So far we have achieved this goal by generating increasingly baroque textual "interface files" to convey information from the exporting module to the importing one. As the information becomes more elaborate this approach is less and less attractive. Like the object-oriented community (Chambers, Dean & Grove [1995]), we regard a serious assault on global (cross-module) optimisation as the most plausible next "big win".

Acknowledgements

The Glasgow Haskell Compiler was built by many people, including Will Partain, Jim Mattson, Kevin Hammond, Andy Gill, André Santos, Patrick Sansom, Cordelia Hall, and Simon Marlow. I'm very grateful to Sigbjorn Finne, Hanne Nielson, Will Partain, Patrick Sansom, and Phil Trinder for helpful feedback on drafts of this paper.

The Glasgow Haskell Compiler is freely available at

`http://www.dcs.gla.ac.uk/fp/software/ghc.html`

References

AV Aho, R Sethi & JD Ullman [1986], *Compilers - principles, techniques and tools*, Addison Wesley.

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

AW Appel & T Jim [1996], "Shrinking Lambda-Expressions in Linear Time," Department of Computer Science, Princeton University.

Z Ariola, M Felleisen, J Maraist, M Odersky & P Wadler [Jan 1995], "A call by need lambda calculus," in *22nd ACM Symposium on Principles of Programming Languages, San Francisco*, ACM, 233–246.

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

DF Bacon, SL Graham & OJ Sharp [Dec 1994], "Compiler transformations for high-performance computing," *ACM Computing Surveys* 26, 345–420.

B Calder, D Grunwald & B Zorn [Dec 1994], "Quantifying behavioural differences between C and C++ programs," *Journal of Programming Languages* 2, 313–351.

C Chambers, J Dean & D Grove [Apr 1995], "A framework for selective recompilation in the presence of complex intermodule dependencies," in *Proc International Conference on Software Engineering, Seattle*.

JW Davidson & AM Holler [1988], "A study of a C function inliner," *Software – Practice and Experience* 18, 775–790.

C Flanagan, A Sabry, B Duba & M Felleisen [June 1993], "The essence of compiling with continuations," *SIGPLAN Notices* 28, 237–247.

P Fradet & D Le Metayer [Jan 1991], "Compilation of functional languages by program transformation," *ACM Transactions on Programming Languages and Systems* 13, 21–51.

A Gill, J Launchbury & SL Peyton Jones [June 1993], "A short cut to deforestation," in *Proc Functional Programming Languages and Computer Architecture, Copenhagen*, ACM, 223–232.

AJ Gill [Jan 1996], "Cheap deforestation for non-strict functional languages," PhD thesis, Department of Computing Science, Glasgow University.

J Girard [1971], "Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination de coupures dans l'analyse et la theorie des types," in *2nd Scandinavian Logic Symposium*, JE Fenstad, ed., North Holland, 63–92.

R Harper & G Morrisett [Jan 1995], "Compiling polymorphism using intensional type analysis," in *22nd ACM Symposium on Principles of Programming Languages, San Francisco*, ACM, 130–141.

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.

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

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

R Kelsey & P Hudak [Jan 1989], "Realistic compilation by program transformation," in *Proc ACM Conference on Principles of Programming Languages*, ACM, 281–292.

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

DA Kranz, R Kelsey, J Rees, P Hudak, J Philbin & N Adams [1986], "ORBIT - an optimising compiler for Scheme," in *Proc SIGPLAN Symposium on Compiler Construction*, ACM.

J Launchbury [Jan 1993], "A natural semantics for lazy evaluation," in *20th ACM Symposium on Principles of Programming Languages, Charleston*, ACM, 144–154.

J Launchbury & SL Peyton Jones [June 1994], "Lazy functional state threads," in *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'94), Orlando*, ACM, 24–35.

S Marlow [March 1996], "Deforestation for Higher Order Functional Programs," PhD thesis, Department of Computing Science, University of Glasgow.

R Morrison, A Dearle, RCH Connor & AL Brown [July 1991], "An ad hoc approach to the implementation of polymorphism," *ACM Transactions on Programming Languages and Systems* 13, 342–371.

A Ohori [Jan 1992], "A compilation method for ML-style polymorphic record calculi," in *19th ACM Symposium on Principles of Programming Languages, Albuquerque*, ACM, 154–165.

WD Partain [1993], "The nofib Benchmark Suite of Haskell Programs," in *Functional Programming, Glasgow 1992*, J Launchbury & PM Sansom, eds., Workshops in Computing, Springer Verlag, 195–202.

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

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

SL Peyton Jones, CV Hall, K Hammond, WD Partain & PL Wadler [March 1993], "The Glasgow Haskell compiler: a technical overview," in *Proceedings of Joint Framework for Information Technology Technical Conference, Keele*, DTI/SERC, 249–257.

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, 636–666.

SL Peyton Jones & D Lester [May 1991], "A modular fully-lazy lambda lifter in HASKELL," *Software – Practice and Experience* 21, 479–506.

SL Peyton Jones & WD Partain [1993], "Measuring the effectiveness of a simple strictness analyser," in *Functional Programming, Glasgow 1993*, K Hammond & JT O'Donnell, eds., Workshops in Computing, Springer Verlag, 201–220.

SL Peyton Jones, WD Partain & A Santos [May 1996], "Let-floating: moving bindings to give faster programs," in *Proc International Conference on Functional Programming, Philadelphia*, ACM.

SL Peyton Jones & A Santos [1994], "Compilation by transformation in the Glasgow Haskell Compiler," in *Functional Programming, Glasgow 1994*, K Hammond, DN Turner & PM Sansom, eds., Workshops in Computing, Springer Verlag, 184–204.

SL Peyton Jones & PL Wadler [Jan 1993], "Imperative functional programming," in *20th ACM Symposium on Principles of Programming Languages, Charleston*, ACM, 71–84.

JC Reynolds [1974], "Towards a theory of type structure," in *International Programming Symposium*, Springer Verlag LNCS 19, 408–425.

PM Sansom [Sept 1994], "Execution profiling for non-strict functional languages," PhD thesis, Technical Report FP-1994-09, Department of Computer Science, University of Glasgow, (`ftp://ftp.dcs.glasgow.ac.uk/pub/glasgow-fp/tech_reports/FP-94-09_execution-profiling.ps.Z`).

PM Sansom & SL Peyton Jones [Jan 1995], "Time and space profiling for non-strict, higher-order functional languages," in *22nd ACM Symposium on Principles of Programming Languages, San Francisco*, ACM, 355–366.

A Santos [Sept 1995], "Compilation by transformation in non-strict functional languages," PhD thesis, Department of Computing Science, Glasgow University.

Z Shao & AW Appel [June 1995], "A type-based compiler for Standard ML," in *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'95), La Jolla*, ACM, 116–129.

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

A Tolmach [June 1994], "Tag-free garbage collection using explicit type parameters," in *ACM Symposium on Lisp and Functional Programming, Orlando*, ACM, 1–11.

PL Wadler [1990], "Deforestation: transforming programs to eliminate trees," *Theoretical Computer Science* 73, 231–248.

PL Wadler [Jan 1992], "The essence of functional programming," in *19th ACM Symposium on Principles of Programming Languages, Albuquerque*, ACM, 1–14.

PL Wadler & DN Turner [June 1995], "Once upon a type," in *Proc Functional Programming Languages and Computer Architecture, La Jolla*, ACM, 1–11.