A transformation-based optimiser for Haskell

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

André L M Santos
Departmento de Informática, Universidade Federal de Pernambuco
Email: alms@di.ufpe.br.
WWW: http://www.di.ufpe.br/~alms

October 13, 1997

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.

This paper is based in part on Peyton Jones [1996] and Peyton Jones, Partain & Santos [1996]. *It will appear in Science of Computer Programming 1998.*

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 for compiler-writers 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. The following distinctive themes emerge, all of which are elaborated later in the paper:

- We frequently find a close interplay between theory and practice, a particularly satisfying aspect of functional-language research.
- Often, a single transformation elegantly generalises a textbook compiler optimisation, or effectively subsumes several such optimisations.
- Our compiler *infers* types for the source program, but then *maintains* types throughout the compilation process. We have found this to be a big win, for two reasons: it supports a powerful consistency check on the correctness of the compiler, and it provides information that is used to drive some optimisations (notably strictness analysis).

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 our 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 *desugaring.*
2. The middle consists of a sequence of Core-to-Core transformations, and forms the subject of this paper.
3. The back end translates the resulting Core program into C, whence it is compiled to machine code (Peyton Jones [1992]).

In what sense does this structure perform "compilation by transformation"? After all, since the middle just transforms one Core program to another, it is presumably optional — and indeed our compiler has this property. The idea, however, is to *do as much work as possible in the middle, leaving the irreducible minimum in the front and back ends.* The front end should concentrate entirely on scope analysis, type inference[1], and a simple-minded translation

[1] Why do we not instead translate to Core and then typecheck? Because one gets much better error messages from the typechecker if it is looking at source code rather than desugared source code. Furthermore, to require the translation to Core to preserve exactly Haskell's type-inference properties places undesirable extra constraints on the translation. Lastly, the translation of Haskell's system of overloading into Core can only be done in the knowledge of the programs typing.

to Core, ignoring efficiency. The back end should include optimisations only if they cannot be done by a Core-to-Core transformation. This paper describes several examples of optimisations that are traditionally done in the desugarer, or in the code generator, which we re-express as Core-to-Core transformations.

In short, just about everything that could be called an "optimisation" — and optimisations constitute the bulk of what most quality compilers do — appears in the middle.

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 fixed 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.

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.

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:

- 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).

- Two global transformation passes, one that performs and exploits strictness analysis (Section 6), and one that moves bindings to improve allocation and sharing (Section 7).

We conclude with a brief enumeration of the other main transformations incorporated in GHC (Section 8), a short discussion of separate compilation (Section 9), some measurements of the performance improvements achievable by transformation (Section 10), and a summary of the lessons we learned from our experience (Section 11).

Program	*Prog*	→	$Bind_1$; … ; $Bind_n$	$n \geq 1$
Binding	*Bind*	→	*var* = *Expr*	Non-recursive
		|	`rec` var_1 = $Expr_1$;	Recursive $n \geq 1$
			… ;	
			var_n = $Expr_n$	
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

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.

A Core expression is in *weak head normal form (or WHNF)* if it is a lambda abstraction, constructor application, variable, or literal.

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. In what follows we give an informal operational model, but it can readily be formalised along the lines described by Launchbury [1993].

Like any higher-order language, 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 as follows: "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". If `x` is an as-yet-unevaluated thunk, the act of evaluating it is typically implemented by saving live variables and a return address on the stack, and jumping to the code stored inside the thunk. When evaluation is complete, the now-evaluated thunk returns to the saved return address.

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

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

The syntax in Figure 1 requires that function arguments must be atoms[2] (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.

[2] This syntax is becoming quite widely used (Ariola et al. [1995]; Flanagan et al. [1993]; Launchbury [1993]; Peyton Jones [1992]; Tarditi et al. [1996]).

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:

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

Operationally, there is no need to build a thunk for `reverse xs` and then evaluate it; instead, we can simply save a return address, load `xs` into an argument register, and jump to the code for `reverse`. 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

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

In an untyped Core language, `compose` might be defined like this

```
  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:

$$\texttt{compose show double v} \quad\Longrightarrow\quad \texttt{let y = double v in show y}$$

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. For example if we inline `compose` in another call

```
  compose toUpper show v
```

where `toUpper` converts a `String` to upper case, we obtain

$$\texttt{compose toUpper show v} \quad\Longrightarrow\quad \texttt{let y = show v in toUpper y}$$

and here `y` has type `String`.

This difficulty arises because `y`'s type in the body of `compose` itself is just a type variable, β. Evidently, 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

The first transformation that we discuss is inlining. 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%[3] (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.

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.

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

In the case of WHNFs, the trade-off is simply between code size and the benefit of inlining. Atoms and constructor applications are easy: they always small enough to inline. (Recall that constructor applications must have atomic arguments.)

Functions, in contrast, can be large, so the effect of unrestricted inlining on code size can be substantial. Like most compilers, we use a heuristic (but no formal analysis) for deciding when to inline functions. More precisely, we compute the "space penalty" of inlining a function at a call site as follows:

- Compute the size (in syntax nodes) of the body of the function.
- Subtract one for each argument, since we are going to replace a function call with an instance of the body.
- Lastly, subtract a "discount" for each argument that (a) is scrutinised by a `case` expression in the function body, and (b) is bound to a constructor at the call site. If inlining is performed, a case-of-known-constructor transformation will throw away all but one branch of the `case` expression; hence the discount. The discount is very crude, however: it is just a constant.

If the space penalty thus computed is smaller than some fixed (command-line settable) constant then we inline the function at the call site.

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] do 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.

Most seriously, though, 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[4]. 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 (Turner, Wadler & Mossin [1995]). The type system assigns a type of $\mathtt{Int}^\omega$ to `y` in the above example, the superscript ω indicating that `y` might be evaluated more than once. However, it assigns a type of $\mathtt{Int}^1$ 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 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[5]

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

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

[4] 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])!

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

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 desugaring 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 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.

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}
```

Desugaring 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}
```

Desugaring 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.

6.1 Exposing boxing to transformation

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. 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 ordinary 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[6] Now we can express the previously-primitive `+` operation thus[7]:

[6] The `#` symbol has no significance to the compiler; we use it simply as a lexical reminder that the identifier has an unboxed type, or takes arguments of unboxed type.

[7] You may wonder why we write `(case a# +# b# of r# -> I# r#)` instead of the more obvious `I# (a# +# b#)`. The reason is that the latter is not a term in the Core language — constructor arguments must be atoms. Furthermore, the alternative `(let r# = a# +# b# in I# r#)` isn't legal either, because `r#` is an *unboxed* value and hence can't be heap-allocated by `let`. So the `case` expression is really just an eager let-binding.

```
+ = \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 b, 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 Exploiting 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#`. When it is created the wrapper is 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 A simple strictness and absence analyser

GHC uses a rather simple strictness analyser, the idea being to get a large fraction of the benefit of more sophisticated strictness analysis with a small fraction of the effort[8]. More specifically, GHC uses a simple, higher-order abstract interpretation, over a domain that includes just top, bottom, functions, and finite products. It is described by Peyton Jones & Partain [1993], but we briefly review the main design choices in the rest of this section.

6.3.1 Fixpoints and widening

The main challenge in abstract interpretation is usually that of finding the fixpoints of recursive abstract functions. We take a simple approach: after each iteration we *widen* the abstract value, so that it is easy to compare with the previous iteration. This results in a loss of accuracy, but it is fast, and easy to implement.

The widening operator we use is this: given an abstract function we find out whether it is strict in each of its arguments independently, and treat the vector of results as the widened approximation to the function. For example, consider the function:

```
f x y z = if x==0 then f (x-1) z y else y
```

The zeroth approximation to the abstract value of f, f_0, is by definition bottom. We write this approximation in widened form thus: f_0 = SSS — the "S" stands for "strict" — to indicate that f_0 is strict in all three arguments. After one iteration, we find that f_1 = SSL — the "L" stands for "lazy" — because z is not used in the else branch. After two iterations, using f_1 in the recursive call to f, we find that f_2 = SLL, and this turns out to be the fixpoint. There are well known pitfalls with this approach (Clack & Peyton Jones [1985]), but they can be avoided (at the expense of accuracy) by the simple expedient of *always using the widened function in the recursive call.*

A side benefit is that there is an obvious textual representation of the abstract value of a function, which we use for conveying strictness information between modules (Section 9).

In reality, we have found it essential to widen *non-recursive* functions as well, for a reason that is not initially obvious. Consider the following non-recursive definitions:

```
f x = case x of
        []     -> ...
        (p:ps) -> ...

g y = case y of
        C1 a b -> ...(f a)...
```

[8] Of course, it is impossible to know whether one has achieved this goal without implementing a sophisticated analyser as well, which we have not done!

```
          C2 c d -> ...(f c)...
          C3 e f -> ...(f e)...
```

If we do not widen the abstract values for f and g then consider what happens when the abstract interpreter finds a call such as (g a). It applies the abstract value of g to the abstract value of a; this application will take the least upper bound of the three case alternatives in g's right hand side. Each of these three will evaluate a call to (the abstract) f, and each of these calls will take the least upper bound of f's two case alternatives. There is a multiplicative effect, in which *every branch of every conditional reachable from a particular call is evaluated.* (In normal evaluation, of course, only one branch is taken from a conditional, but abstract interpretation, in effect, takes all of them.) This turns out to be far too slow in practice. The solution is to trade accuracy for time: simply widen non-recursive functions as well as recursive ones. Once g is widened its right hand side is no longer evaluated at every call. For example, to apply the widened function value SSL the abstract evaluator simply checks to see if either of its first two arguments are bottom, and if so returns bottom, otherwise top.

6.4 Products and absence analysis

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 — it is for precisely this reason that the abstract domain includes finite products. 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 such cases we revert to the simple two-point abstract domain.

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.

6.5 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 }
```

We call this the *let-to-case* transformation. As before, the reboxing binding for `x` usually 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.

7 Code motion

Consider the following expression:

```
        let v = let w = R
                in w : []
        in B
```

A semantically-equivalent expression which differs only in the positioning of the binding for `w` is this:

```
         let w = R
         in let v = w : []
         in B
```

While the two expressions have the same value, the second is likely to be more efficient than the first to evaluate. (We will say why this is so in Section 7.3.) A good compiler should transform the first expression into the second. However, the difference in efficiency is modest, and the transformation between the two seems almost too easy to merit serious study; as a result, not much attention has been paid to transformations of this kind. We call them "let-floating" transformations, because they concern the exact placement of `let` or `letrec` bindings; in the example, it is the binding for `w` which is floated from one place to another. They correspond closely to "code motion" in conventional compilers.

We have found it useful to identify three distinct kinds of let-floating transformations:

- *Floating inwards* moves bindings as far inwards as possible (Section 7.1).
- *The full laziness transformation* floats selected bindings outside enclosing lambda abstractions (Section 7.2)

- *Local transformations* "fine-tune" the location of bindings (Section 7.3).

After describing the three transformations we describe how they are implemented in GHC (Sections 7.4 and 7.5). Later on we quantify their effectiveness (Section 10.6). More detailed measurements are presented in Peyton Jones, Partain & Santos [1996].

7.1 Floating inwards

The floating-inward transformation is based on the following observation: *other things being equal, the further inward a binding can be moved, the better.* For example, consider:

```
let x = y+1
in case z of
        []     -> x*x
        (p:ps) -> 1
```

Here, the binding for `x` is used in only one branch of the `case`, so it can be moved into that branch:

```
case z of
  [] -> let x = y+1
         in x*x
  (p:ps) -> 1
```

Moving the binding inwards has at least three distinct benefits[9]:

√ *The binding may never be "executed".* In the example, `z` might turn out to be of the form `(p:ps)`, in which case the code which deals with the binding for `x` is not executed. Before the transformation a thunk for `x` would be allocated regardless of the value of `z`.

√ *Strictness analysis has a better chance.* It is more likely that at the point at which the binding is now placed it is known that the bound variable is sure to be evaluated. This in turn may enable other, strictness-related, transformations to be performed. In our example, instead of allocating a thunk for `x`, GHC will simply evaluate `y`, increment it and square the result, allocating no thunks at all (Section 6).

√ *Redundant evaluations may be eliminated.* It is possible that the RHS will "see" the evaluation state of more variables than before. To take a similar example:

```
let x = case y of (a,b) -> a
in
case y of
  (p,q) -> x+p
```

If the binding of `x` is moved inside the `case` branch, we get:

```
case y of
```

[9] We indicate advantages with √ and disadvantages with ×. The symbol □ indicates moot points.

```
  (p,q) -> let x = case y of (a,b) -> a
           in
           x+p
```

Now the compiler can spot that the inner case for y is in the RHS of an enclosing case which also scrutinises y. It can therefore eliminate the inner case to give:

```
case y of
 (p,q) -> p+p
```

The first two benefits may also accrue if a binding is moved inside the RHS of another binding. For example, floating inwards would transform:

```
let x = v+w
    y = ...x...x...
in
B
```

(where B does not mention x) into

```
let y = let x = v+w in ...x...x...
in
B
```

(The alert reader will notice that this transformation is precisely the opposite of that given at the start of Section 7, a point we return to in Section 7.3.) This example also illustrates another minor effect of moving bindings around:

□ Floating can change the size of the thunks allocated. Recall that in our implementation, each let(rec) binding allocates a heap object that has one slot for each of its free variables. The more free variables there are, the larger the object that is allocated. In the example, floating x into y's RHS removes x from y's free variables, but adds v and w. Whether y's thunk thereby becomes bigger or smaller depends on whether v and/or w were already free in y.

So far, we have suggested that a binding can usefully be floated inward "as far as possible"; that is, to the point where it can be floated no further in while still keeping all the occurrences of its bound variable in scope. There is an important exception to this rule: it is dangerous to float a binding inside a lambda abstraction, as we discussed in Section 4.1. If the abstraction is applied many times, each application will instantiate a fresh copy of the binding. Worse, if the binding contains a reducible expression the latter will be re-evaluated each time the abstraction is applied.

The simple solution is never to float a binding inside a lambda abstraction, and that is what our compiler currently does, although we plan in future to use guidance from linear-type information (see Section 4.2). But what if the binding is inside the abstraction to start with? We turn to this question next.

7.2 Full laziness

Consider the definition

```
f = \xs -> letrec
             g = \y -> let n = length xs
                       in ...g...n...
           in
             ...g...
```

Here, the length of xs will be recomputed on each recursive call to g. This recomputation can be avoided by simply floating the binding for n outside the \y-abstraction:

```
f = \xs -> let n = length xs
           in
           letrec
             g = \y -> ...g...n...
           in
           ...g...
```

This transformation is called *full laziness*. It was originally invented by Hughes (Hughes [1983]; Peyton Jones [1987]), who presented it as a variant of the supercombinator lambda-lifting algorithm. Peyton Jones & Lester [1991] subsequently showed how to decouple full laziness from lambda lifting by regarding it as an exercise in floating let(rec) bindings outwards. Whereas the float-in transformation avoids pushing bindings inside lambda abstractions, the full laziness transformation actively seeks to do the reverse, by floating bindings outside an enclosing lambda abstraction.

The full laziness transformation can save a great deal of repeated work, and it sometimes applies in non-obvious situations. One example we came across in practice is part of a program which performed the Fast Fourier Transform (FFT). The programmer wrote a list comprehension similar to the following:

```
[xs_dot (map (do_cos k) (thetas n)) | k<-[0 .. n-1]]
```

What he did not realise is that the expression (thetas n) was recomputed for each value of k! The list comprehension syntactic sugar was translated into the Core language, where the (thetas n) appeared inside a function body. The full laziness transformation lifted (thetas n) out past the lambda, so that it was only computed once.

A potential shortcoming of the full laziness transformation, as so far described, is this: it seems unable to float out an expression that is free in a lambda abstraction, but not let(rec) bound. For example, consider

```
f = \x -> case x of
            []     -> g y
            (p:ps) -> ...
```

Here, the subexpression (g y) is free in the \x-abstraction, and might potentially be an expensive computation which could be shared among all applications of f. It is simple enough, in principle, to address this shortcoming, by simply let-binding (g y) thus:

```
f = \x -> case x of
```

```
                []     -> let a = g y
                          in a
                (p:ps) -> ...
```

Now the binding for `a` can be floated out like any other binding.

The full laziness transformation may give rise to large gains, but at the price of making worse all the things that floating inwards makes better (Section 7.1). Hence, the full laziness transformation should only be applied when there is some chance of a benefit. For example, it should not be used if either of the following conditions hold:

1. *The RHS of the binding is already a value, or reduces to a value with a negligible amount of work.* If the RHS is a value then no work is saved by sharing it among many invocations of the same function, though some allocation may be saved.
2. *The lambda abstraction is applied no more than once,* information that should be made available by the linear type inference system (Section 4.2).

There is a final disadvantage to the full laziness which is much more slippery: it may cause a space leak. Consider:

```
  f = \x -> let a = enumerate 1 n in B
```

where `enumerate 1 n` returns the list of integers between `1` and `n`. Is it a good idea to float the binding for `a` outside the `\x`-abstraction? Certainly, doing so would avoid recomputing `a` on each call of `f`. On the other hand, `a` is pretty cheap to recompute and, if `n` is large, the list might take up a lot of store. It might even turn a constant-space algorithm into a linear-space one, or even worse.

In fact, as our measurements show, space leaks do not seem to be a problem for real programs. We are, however, rather conservative about floating expressions to the top level where, for tiresome reasons, they are harder to garbage collect (Section 7.5).

7.3 Local transformations

The third set of transformations consist of local rewrites, which "fine-tune" the placement of bindings. There are just three such transformations:

```
          (let v=R in B) A  ==>  (let v=R in B A)

case (let v=R in B) of {...}  ==>  let v=R
                                   in
                                   case B of {...}

     let x = let v=R1 in R2   ==>  let v=R1
     in B                          in
                                   let x=R2
                                   in B
```

Each of the three has an exactly equivalent form when the binding being floated outwards is a `letrec`. The third also has a variant when the outer binding is a `letrec`: in this case, the binding being floated out is combined with the outer `letrec` to make a larger `letrec`. Subsequent dependency analysis (see Section 7.4) will split up the enlarged group if it is possible to do so.

The first two transformations are always beneficial. They do not change the number of allocations, but they do give other transformations more of a chance. For example, the first (which we call *let-float-from-application*) moves a let outside an application. Doing so cannot make things worse and sometimes makes things better — for example, `B` might be a lambda abstraction which can then be applied to `A`. The second, *let-float-from-case*, floats a `let(rec)` binding outside a `case` expression, which might improve matters if, for example, `B` was a constructor application.

The third transformation, the *let-float-from-let* transformation, which floats a `let(rec)` binding from the RHS of another `let(rec)` binding, is more interesting. It has the following advantages:

√ *Floating a binding out may reveal a WHNF.* For example, consider the expression:

```
let x = let v = R in (v,v)
in B
```

When this expression is evaluated, a thunk will be allocated for `x`. When (and if) `x` is evaluated by `B`, the contents of the thunk will be read back into registers, its value (the pair `(v,v)`) computed, and the heap-allocated thunk for `x` will be overwritten with the pair.

Floating the binding for `v` out would instead give:

```
let v = R
    x = (v,v)
in B
```

When this expression is evaluated, a thunk will be allocated for `v`, and a pair for `x`. *In other words,* `x` *is allocated in its final form.* No update will take place when `x` is evaluated, a significant saving in memory traffic.

√ There is a second reason why revealing a normal form may be beneficial: `B` may contain a `case` expression which scrutinises `x`, thus:

```
...(case x of (p,q) -> E)...
```

Now that `x` is revealed as being bound to the pair `(v,v)`, this expression is easily transformed to

```
...(E[v/p,v/q])...
```

using the case-of-known-constructor transformation.

- √ *Floating* `v`*'s binding out may reduce the number of heap-overflow checks.* A "heap-overflow check" is necessary before each sequence of `let(rec)` bindings, to ensure that a large enough contiguous block of heap is available to allocate all of the bindings in the sequence. For example, the expression

  ```
  let v = R
      x = (v,v)
  in B
  ```

  requires a single check to cover the allocation for both `v` and `x`. On the other hand, if the definition of `v` is nested inside the RHS of `x`, then two checks are required.

These advantages are all very well, but the let-from-let transformation also has some obvious disadvantages: after all, it was precisely the reverse of this transformation which we advocated when discussing the floating-inward transformation! Specifically, there are two disadvantages:

- × If `x` is not evaluated, then an unnecessary allocation for `v` would be performed. However, the strictness analyser may be able to prove that `x` is sure to be evaluated, in which case the let-from-let transformation is always beneficial.

- × It is less likely that the strictness analyser will discover that `v` is sure to be evaluated. This suggests that the strictness analyser should be run before performing the let-from-let transformation.

Given these conflicting trade-offs, there seem to be four possible strategies for local let-floating:

Never — no local let-floating is performed at all.

Strict — bindings are floated out of strict contexts only; namely, applications, `case` scrutinees, and the right-hand-sides of strict `lets`[10].

WHNF — like "Strict", but in addition a binding is floated out of a `let(rec)` right-hand-side if doing so would reveal a WHNF.

Always — like "Strict", but in addition any binding at the top of a `let(rec)` right-hand-side is floated out.

We explore the practical consequences of these four strategies in Section 10.6.

7.4 Composing the pieces

We have integrated the three let-floating transformations into GHC. The full laziness and float-inwards transformations are implemented as separate passes. In contrast, the local let-floating transformations are implemented by the simplifier (Section 2). Among the transformations performed by the simplifier is dependency analysis, which splits each `letrec` binding into its

[10] A strict `let` is one whose bound variable is sure to be evaluated by the body of the `let`.

minimal strongly-connected components. Doing this is sometimes valuable because it lets the resulting groups be floated independently.

We perform the transformations in the following order.

1. Do the full laziness transformation.
2. Do the float-inwards transformation. This won't affect anything floated outwards by full laziness; any such bindings will be parked just outside a lambda abstraction.
3. Perform strictness analysis.
4. Do the float-inwards transformation again.

Between each of these passes, the simplifier is applied.

We do the float-inwards pass before strictness analysis because it helps to improve the results of strictness analysis. The desirability of performing the float-inwards transformation again after strictness analysis surprised us. Consider the following function:

```
f x y = if y==0
        then error ("Divide by zero: " ++ show x)
        else x/y
```

The strictness analyser will find `f` to be strict in `x`, because calls to `error` are equivalent to $\perp$, and hence will pass `x` to `f` in unboxed form. However, the `then` branch needs `x` in boxed form, to pass to `show`. The post-strictness float-inwards transformation floats a binding that re-boxes `x` into the appropriate branch(es) of any conditionals in the body of `f`, thereby avoiding the overhead of re-boxing `x` in the (common) case of taking the `else` branch.

The implementation of the float-in transformation and local let-floating is straightforward, but the full laziness transformation has a few subtleties, as we discuss next.

7.5 Implementing full laziness

We use a two-pass algorithm to implement full laziness:

1. The first pass annotates each let(rec) binder with its "level number"[11]. In general, level numbers are defined like this.
   - The level number of a let-bound variable is the maximum of the level numbers of its free variables, *and its free type variables.*
   - The level number of a letrec-bound variable is the maximum of the level numbers of the free variables of all the RHSs in the group, less the letrec-bound variables themselves.
   - The level number of a lambda-bound variable is one more than the number of enclosing lambda abstractions.

[11] Actually, all the other binders are also annotated, but they are never looked at subsequently.

- The level number of a case- or type-lambda-bound variable is the number of enclosing (ordinary) lambda abstractions.

2. The second pass uses the level numbers on let(rec)s to float each binding outward to *just outside the lambda which has a level number one greater than that on the binding.*

   Notice that a binding is floated out just far enough to escape all the lambdas which it can escape, and no further. This is consistent with the idea that bindings should be as far in as possible. There is one exception to this: bindings with level number zero are floated right to the top level.

   Notice too that a binding is not moved at all unless it will definitely escape a lambda.

This algorithm is largely as described by Peyton Jones & Lester [1991], but there are a few complications in practice. Firstly, type variables are a nuisance. For example, suppose that `f` and `k` are bound outside the following `\x`-abstraction:

```
\x -> ...(/\a -> ...let v = f a k in ...)
```

We'd like to float out the `v = f a k`, but we can't because then the type variable `a` would be out of scope. The rules above give `a` the same level number as `x` (assuming there are no intervening lambdas) which will ensure that the binding isn't floated out of `a`'s scope. Still, there are some particularly painful cases, notably pattern-matching failure bindings, such as:

```
fail = error a "Pattern fail"
```

We really would like this to get lifted to the top level, despite its free type variable `a`. There are two approaches: ignore the problem of out-of-scope type variables, or fix it up somehow. We take the latter approach, using the following procedure. If a binding `v = e` has free type variables whose maximum level number is strictly greater than that of the ordinary variables, then we abstract over the offending type variables, `a1..an`, thus:

```
v = let v' = /\a1..an -> e in v' a1 ... an
```

Now `v` is given the usual level number (taking type variables into account), while `v'` is given the maximum level number of the ordinary free variables only (since the type variables `a1..an` are not free in `v'`).

The reason this is a bit half baked is that some subsequent binding might mention `v`; in theory it too could be floated out, but it will get pinned inside the binding for `v`. (It's the binding for `v'` which floats.) But our strategy catches the common cases.

The second complication is that there is a penalty associated with floating a binding between two adjacent lambdas. For example, consider the binding

```
f = \x y -> let v = length x in ...
```

It would be possible to float the binding for `v` between the lambdas for `x` and `y`, but the result would be two functions of one argument instead of one function of two arguments, which is less efficient. There would be gain only if a partial application of `f` to one argument was applied many times. Indeed, our measurements[12] indicate that allowing lambdas to be split in this way resulted in a significant loss of performance. Our pragmatic solution is to therefore treat the

[12] See Santos [1995] for these figures; we do not present them here.

lambdas for `x` and `y` as a single "lambda group", and to give a single level number to all the variables bound by a group. As a result, lambda groups are never split.

The third complication is that we are paranoid about giving bindings a level number of zero, because that will mean they float right to the top level, where they might cause a space leak. (In our implementation, all top-level values are retained for the whole life of the program. It would be possible for the garbage collector to figure out which of them cannot be referred to again, and hence which could safely be garbage collected, but doing so adds complexity and slows both the mutator and the garbage collector.) We use several heuristics which sometimes decide (conservatively) to leave a binding exactly where it is. If this happens, rather than giving the binding level number zero, it is given a level number of the number of enclosing lambdas, so that it will not be moved by the second pass.

8 Other GHC transformations

We have focused so far on 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 detailed measurements of the frequency and usefulness of each transformation.

The specialiser uses partial evaluation to create specialised versions of overloaded functions, using much the same technique as that described by Jones [1994].

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:

$$\texttt{\textbackslash x -> R} \quad\Longrightarrow\quad \texttt{\textbackslash x a -> R a}$$

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 yet 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. This is a rather disappointing result, but we believe that there is potential for improving it considerably.

Deforestation also allows the desugaring of list comprehensions to be simplified considerably, moving a group of optimisations from the desugarer to the deforester. The details are in Gill, Launchbury & Peyton Jones [1993].

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.

9 Separate compilation

GHC makes a serious attempt to propagate transformations across modules. When a module M is compiled, as well as producing M's object code, the compiler also emits M's *interface file* that contains, *inter alia*, three sorts of information:

- *Scope information*: the names and defining module of each class, type and value exported by M. This tells an importing module what names are exported by M, and hence are brought into scope by importing M.
- *Type information*: the type declarations of the classes, types, and values defined in M, *whether exported or not.* Haskell allows an exported value to have a type mentioning type constructors that are not exported, and the compiler needs access to the latter.
- *Implementation information*: for each value defined in M, the interface file gives:
  - Its definition, in the Core language, if it is smaller than some fixed threshold; this allows it to be inlined at call sites in other modules.
  - If it is overloaded, what specialised instances of the value have been compiled.
  - Its strictness information.
  - Its linearity information.
  - Its arity (how many arguments it takes).

Providing such implementation information allows an importing module to take advantage of detailed knowledge about M, but of course it also increases the coupling between modules. If M is recompiled, and some of this implementation information changes, then each module that imports M must be recompiled too. We leave the choice to the programmer: implementation information is written into interface files if and only if the `-O` flag is specified when invoking the compiler. In this way the programmer can trade compilation time for runtime efficiency.

10 Effect of the Transformations

Whilst every transformation we have described was motivated by particular examples, it is far from obvious that each will deliver measurable performance gains when applied to "average" programs. In this section we present quantitative measurements of some of the most important transformations described above.

10.1 Setup

We measured the effect of our transformations on a sample of between 15 and 50 programs from our `Nofib` test suite[13] (Partain [1993]). Many of these programs are "real" applications — that is, application programs written by someone other than ourselves to solve a particular problem. None was designed as a benchmark, and they range in size from a few hundred to a few thousand lines of Haskell. Our results are emphatically *not* best-case results on toy programs!

It is difficult to present the effect of many interacting transformations in a modular way. If we measure the effect of switching them *on* one at a time we risk being either over-optimistic (because an otherwise un-optimised program is a very soft target) or over-pessimistic (because one transformation relies on another to exploit its effects). Switching them *off* one at a time

[13] The number varied between different experiments, which were carried out over an extended period.

suffers from the opposite objections, but at least it faithfully indicates the cost of omitting that transformation from a production compiler.

Accordingly, *most results are given as percentage changes from the base case in which all transformations are enabled.* A value greater than 1 means "more than the base case", less than 1 means "less than the base case"; whether that is "good" or "bad" depends on what is being measured.

Space precludes listing the results for each individual program. Instead, we report just the average effect, where for "average" we use the *geometric* mean, since we are averaging performance *ratios* (Fleming & Wallace [1986]).

We concentrate on the following measures:

- *Instruction count* — how many instructions are taken to execute the program. This measure is independent of cache locality and paging, which in today's architectures can sometimes dominate all other effects put together. Nevertheless, it is a more portable measure, and our wall-clock-time measurements (which we made as a sanity check) mostly track the instruction-count measure in practice.

- *Heap allocation* — how much heap is allocated by the program during its execution. If an optimisation reduces allocation by a larger factor than instruction count, it is likely that a smaller proportion of instructions are memory cycles, and hence that execution time may decrease by a larger fraction than the instruction count. The reverse is also true, of course.

- *Maximum residency* — the maximum amount of live data during execution. This number directly affects the cost of garbage collection, and is the best measure of the space consumption of a program. The residency numbers were gathered by sampling the amount of live data at frequent intervals, using the garbage collector. Frequent sampling means that any "spikes" in live memory usage are unlikely to be missed.

- *Binary size* — the size of the compiled code, excluding symbol table.

- *Compile time.*

For each set of measurements we recompiled the standard prelude and libraries with the specified set of transformations enabled, so that the results reflect the effect on the *entire* program and not only on the "application" part of it.

Like all quantitative measurements, and especially average measurements, the numbers we present should not be read uncritically. In particular:

- The averages often conceal large individual variations. It is not uncommon to find that a particular transformation has a small effect on most programs, but a dramatic effect on a few.

- Gathering these measurements is a substantial exercise, taking gigabytes of disc space and weeks of CPU time. The set of benchmark programs was not the same in every case — hence the range of sample size — nor were all the measures collected in every experiment.

Nevertheless we believe that the figures present a reasonably truthful picture of the relative importance of the different transformations. The sources we cite elsewhere in this paper give much more detailed breakdowns of many of the figures we summarise here.

10.2 Overall gains from transformation

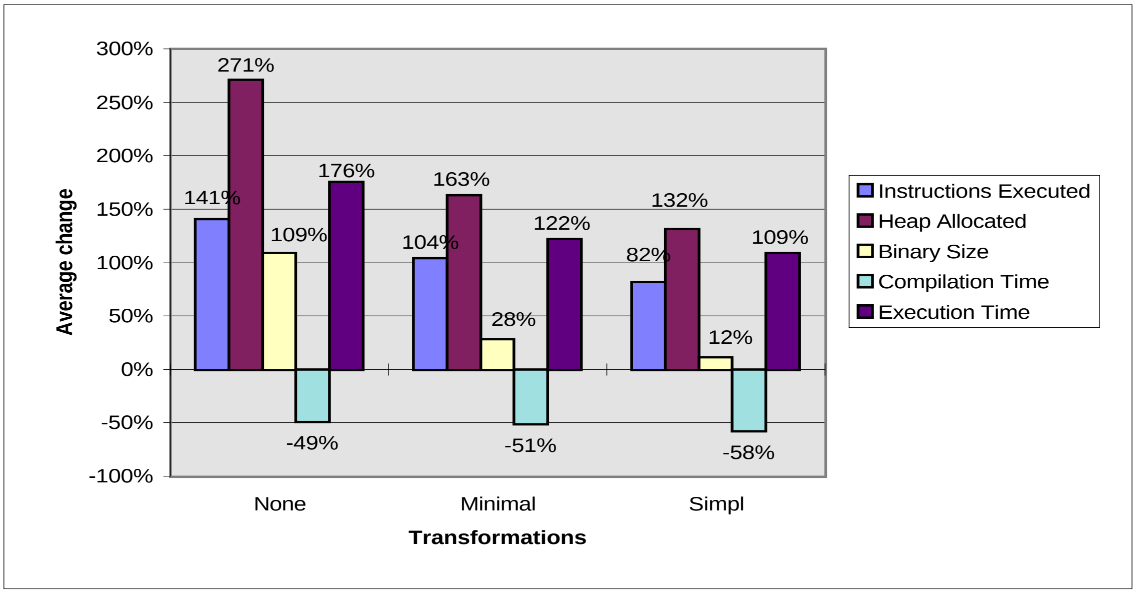

Figure 2: Overall effect of transformations

The overall gains from transformations are presented in Figure 2. The x axis represents various compilation options, each compared to a baseline in which all optimisations are enabled:

None — that is, no transformations at all. It might be argued that this makes the transformations look unreasonably effective, because the desugarer is written assuming that a subsequent simplification will clear up much of its "litter".

Minimal approximates the effect of a more plausible desugarer with no further transformations. We approximated this setup by performing a single non-iterative run of the simplifier, with most transformations disabled. The only important transformations that remain are beta reduction, and the inlining of trivial bindings (that is, ones that bind a variable to another variable or literal).

Simplifier only. Here all the global transformations are switched off, leaving only a full run of the simplifier (up to 4 iterations, although this limit was never reached).

Overall, switching off all transformations increases instruction count by around 140%. The fairer "minimal" case still increases instruction count by 100%, while switching off everything except the simplifier only increases instruction count by 80%. The minimum compilation time, less than half that of full optimisation, is achieved by the "simplifier only" case, so this is what GHC uses when compiling without the `-O` flag.

The following sections investigate individual transformations in more detail.

10.3 The simplifier

It does not make sense to measure the effect of switching the simplifier off while leaving all the other transformations on, since they all rely on the simplifier to clean up after them and exploit their effects. In effect, the simplifier is part of every transformation, so it cannot sensibly be measured in isolation.

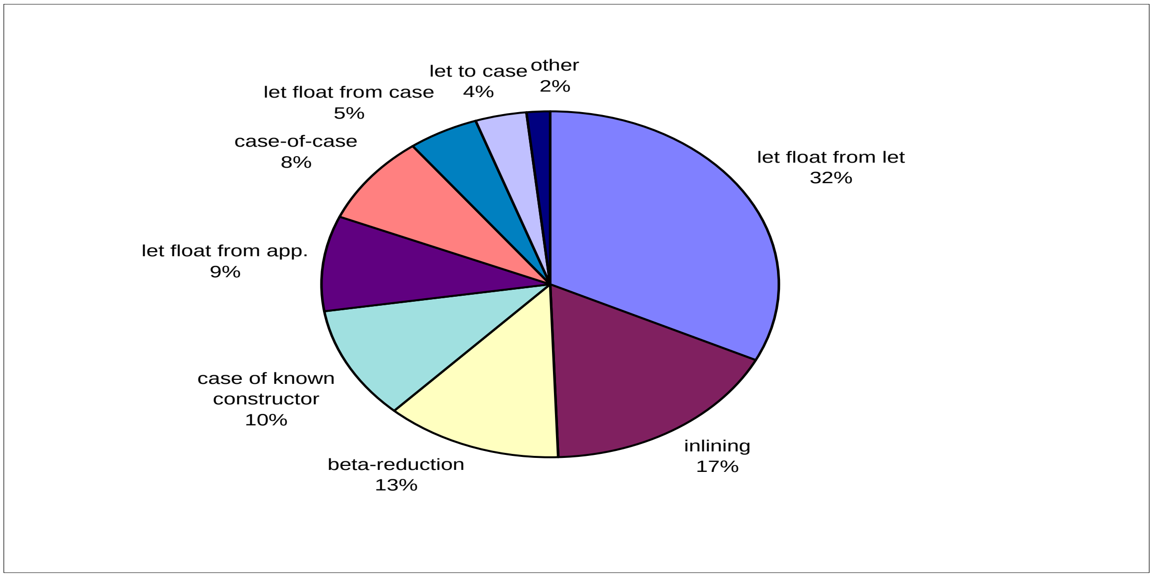

Figure 3: Frequency of transformations

The simplifier implements a large number of separate transformations, and it certainly makes sense to ask how often each is used. Figure 3 answers this question by giving the relative frequency of the most common transformations.

We did not include in the transformation counts the following two transformations, which would otherwise dominate the pie chart:

- Dead code elimination (unused bindings and unreachable case alternatives).
- Inlining for trivial bindings (ones that bind variables to other variables or literals).

These two transformations almost always occur as a byproduct of some other transformation, which is made easier to implement thereby, so their frequency is mostly a consequence of the implementation strategy of other transformations.

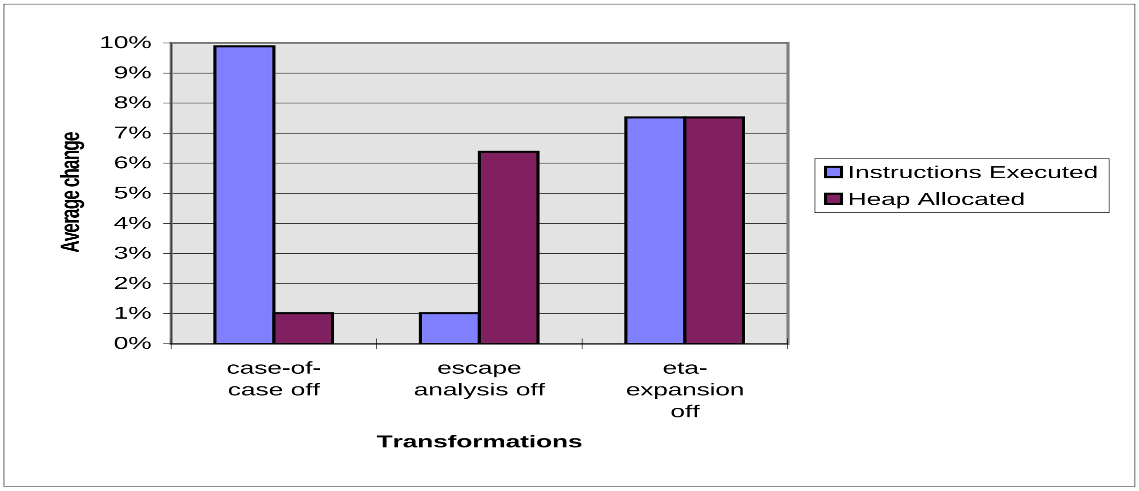

Figure 4: Turning off individual transformations

We also measured the effect of turning off individual transformations, one at a time, relative as always to the full-optimisation base case. Figure 4 shows three interesting cases (let-floating is dealt with in Section 10.6):

- Switching off the case-of-case transformation increases instruction count by a substantial 10%.
- Section 5.1 described how to use `let` bindings to describe join points, asserting that a simple analysis in the code generator suffices to identify these special join-point bindings. The second column in Figure 4 shows that the analysis is not in fact very important: switching it off gives only a 1% increase in instruction count, albeit with a larger increase in heap allocation.
- Eta expansion (Section 8) has a substantial individual effect: switching it off costs some 8% in both instructions and allocation.

10.4 Inlining

The effect of inlining is summarised in Figure 5. The x axis is calibrated by the following inlining strategies:

Off. Inlining is turned off, except for trivial bindings of variables to variables or literals.

One occ. Trivial bindings, and variables or functions that occur only once, in a $\mathcal{W}$-safe context, are inlined.

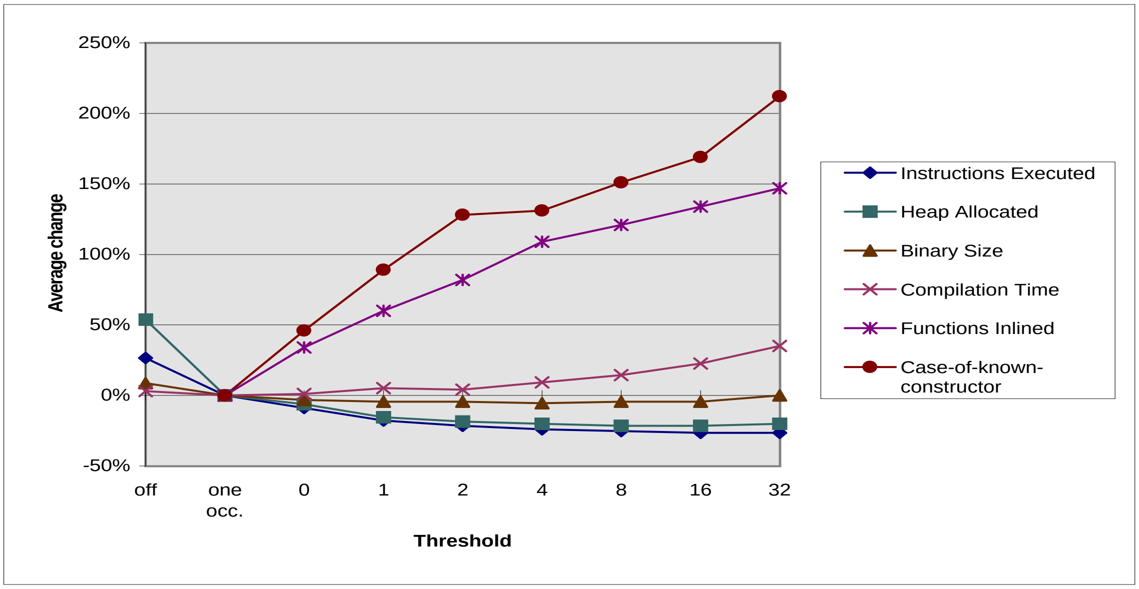

Figure 5: Effect of Inlining Strategies

Threshold(n). Any non-recursive binding is inlined if it is $\mathcal{W}$-safe to do so, and either it occurs just once, or its space penalty (Section 4.1) is less than the given threshold.

As well as the usual measures (instruction count, heap allocation, compilation time) we also show the number of functions that are actually inlined. These graphs show the following effects:

- The bigger effects come directly from inlining variables and functions with one occurrence, and then reasonable gains come up to threshold 3, where the gains start to be minimal.
- Although we get many more functions inlined with larger thresholds, this is not reflected on the number of instructions executed, i.e. we quickly get to a point where more inlining is (almost) useless.
- Binary size remains virtually unaltered, which means that most of the (larger) functions being inlined do not occur many times in the program.
- Compilation time actually decreases initially, since we end up with less to do in later phases of the compiler. With the highest inline threshold we measured (32), compilation takes about 35% longer than the "one occurrence" case.

10.5 Strictness Analysis

The effect of strictness analysis is shown in Figure 6, which shows that if we disable strictness analysis we will increase execution time by about 18%, and the number of objects allocated in the heap is also a lot higher, since we will have many more `lets`.

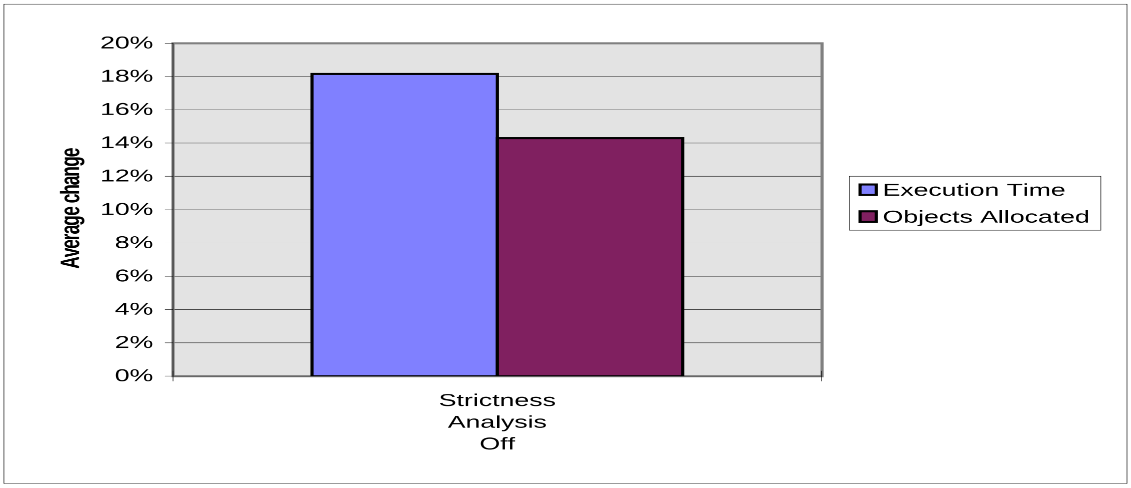

Figure 6: Strictness Analysis

10.6 Code motion

As mentioned in Section 7 we identify three kinds of let-floating: floating inwards, full laziness, and local let floating. Figure 7 presents the average effects of these let floating transformations, relative to the fully-optimised base case:

FI off presents the effect of turning off the float inwards transformation; this increases instruction count by a modest 0.6%.

FL off presents the effect of turning off the full laziness transformation; results here are somewhat variable, but average to around 7%.

Local LF off presents the effect of turning off all *local* `let` floating; this costs around 9%.

All Floating Off presents the effect of turning off all `let` floating, i.e. floating inwards, full laziness and all kinds of local `let` floating. The total penalty in instruction count is (perhaps surprisingly) about equal to the sum of the three individual effects, a substantial 16%. The execution-time sanity check bears this out, with an 18% improvement from let-floating.

Figure 7 also shows the effect of the four variants of local let-floating described in Section 7.3. In the baseline case (full optimisation) we use the “WHNF” strategy, because that appears to give the best results. The other three strategies (“Never”, “Strict”, and “Always”) are presented in the columns “Local LF off”, “Local LF strict”, and “Local LF always”. All three are indeed worse than the baseline strategy; “Strict” and “Never” are very bad (6% and 9% worse respectively), while “Always” is only a little worse (0.5%).

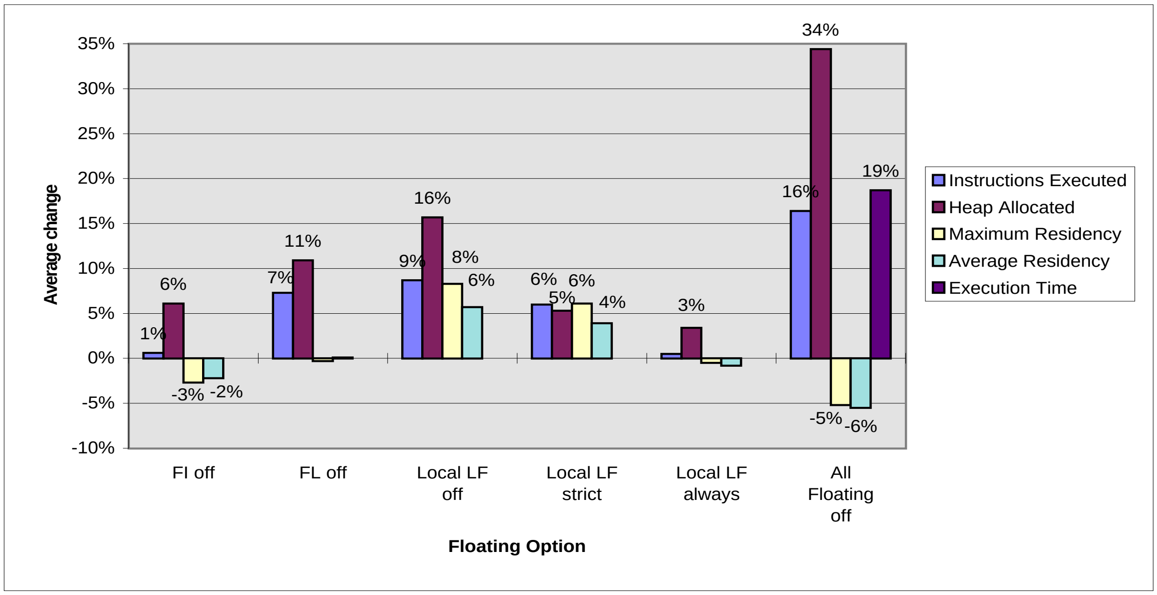

Figure 7: Let Floating

11 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]). Monads allow us to express imperative algorithms in a purely functional setting. In particular, the compiler can freely use its entire armoury of transformations on stateful computations expressed using monads; in contrast, optimising compilers for Lisp or ML must perform some kind of effects analysis to infer which "functions" are pure and which may have side effects; many optimsiations must be disabled for the latter. Since monads simply make explicit the otherwise-implicit flow dependencies it is not clear that we get better code than the analyse-and-disable approach, but we certainly get a simpler compiler.
- 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. All this really only applies to compiler *writers* however; almost all compiler *users* will be content to use the bundle of flags that are conjured up by using the standard `-O` ("please optimise") or `-O2` ("please optimise a lot") flags.

It is hard to estimate the cost of this plug-and-play approach. Would the compiler be faster if several passes were amalgamated into a single giant pass? In one case, namely the simplifier, we have indeed combined many small transformations into a single pass. For the larger transformations the benefits are probably modest: each does a substantial task, so the cost reduction from eliminating the intermediate data structure is probably small and it would come at a high programming cost. It might be more attractive to develop *automatic* techniques for fusing successive passes together, perhaps by generalising the short-cut deforestation technique mentioned above to arbitrary data structures, and explicitly-recursive functions (Launchbury & Sheard [1995]).

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 desugaring, 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 8) 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[14]. On the other hand it is sometimes indispensable to know the type of an expression, notably during strictness analysis. Maintaining types throughout compilation is becoming more popular (Shao & Appel [1995]; Tarditi et al. [1996]).

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 stupidest 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 highly desirable that inlining, strictness analysis, specialisation, and so on, work between modules. For example, many abstract data types export very small functions that would probably be implemented as macros in C. With cross module inlining they can be inlined at every call site, gaining (most of) the advantages of macros without the burden of macro processors' strange semantics. 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".

[14] This is true for the transformations we have implemented. Some transformations, notably closure conversion, require more work, and indeed a more sophisticated type system than the second order lambda calculus (Harper & Morrisett [1995]).

Acknowledgements

The Glasgow Haskell Compiler was built by many people, including Will Partain, Jim Mattson, Kevin Hammond, Andy Gill, Patrick Sansom, Cordelia Hall, and Simon Marlow. I'm very grateful to Sigbjorn Finne, Hanne Nielson, Will Partain, Patrick Sansom, and Phil Trinder, and three anonymous referees, 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/`

References

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

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

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

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

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

DF Bacon, SL Graham & OJ Sharp [1994], "Compiler transformations for high-performance computing," *ACM Computing Surveys* 26(4), Dec 1994, .

B Calder, D Grunwald & B Zorn [1994], "Quantifying behavioural differences between C and C++ programs," *Journal of Programming Languages* 2(4), Dec 1994, .

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

CD Clack & SL Peyton Jones [1985], "Generating parallelism from strictness analysis," Internal Note 1679, Department of Computer Science, University College London, Feb 1985.

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

C Flanagan, A Sabry, B Duba & M Felleisen [1993], "The essence of compiling with continuations," *SIGPLAN Notices* 28(1), June 1993, .

PJ Fleming & JJ Wallace [1986], "How not to lie with statistics - the correct way to summarise benchmark results," *CACM* 29(3), March 1986, .

P Fradet & D Le Metayer [1991], "Compilation of functional languages by program transformation," *ACM Transactions on Programming Languages and Systems* 13(1), Jan 1991, .

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

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

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, 1971, .

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

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 [1992], "Report on the functional programming language Haskell, Version 1.2," *SIGPLAN Notices* 27(5), May 1992.

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

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, 1985, .

MP Jones [1994], "Dictionary-free overloading by partial evaluation," in *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), Orlando, Florida*, ACM, June 1994.

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

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

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

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, 1986.

J Launchbury [1993], "A natural semantics for lazy evaluation," in *20th ACM Symposium on Principles of Programming Languages (POPL'93), Charleston*, ACM, Jan 1993, .

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

J Launchbury & T Sheard [1995], "Warm fusion," in *Proc Functional Programming Languages and Computer Architecture, La Jolla*, ACM, June 1995.

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

R Morrison, A Dearle, RCH Connor & AL Brown [1991], "An ad hoc approach to the implementation of polymorphism," *ACM Transactions on Programming Languages and Systems* 13(3), July 1991, .

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

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, 1993, .

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

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

SL Peyton Jones [1996], "Compilation by transformation: a report from the trenches," in *European Symposium on Programming (ESOP'96), Linköping, Sweden*, Springer Verlag LNCS 1058, Jan 1996, .

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

SL Peyton Jones & J Launchbury [1991], "Unboxed values as first class citizens," in *Functional Programming Languages and Computer Architecture (FPCA'91), Boston*, Hughes, ed., LNCS 523, Springer Verlag, Sept 1991, .

SL Peyton Jones & D Lester [1991], "A modular fully-lazy lambda lifter in HASKELL," *Software – Practice and Experience* 21(5), May 1991, .

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, 1993, .

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

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, 1994, .

SL Peyton Jones & PL Wadler [1993], "Imperative functional programming," in *20th ACM Symposium on Principles of Programming Languages (POPL'93), Charleston*, ACM, Jan 1993, .

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

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

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

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

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

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

D Tarditi, G Morrisett, P Cheng, C Stone, R Harper & P Lee [1996], "TIL: A Type-Directed Optimizing Compiler for ML," in *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'96), Philadelphia*, ACM, May 1996.

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

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

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

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