Let-floating: moving bindings to give faster programs

Simon Peyton Jones, Will Partain, and André Santos
University of Glasgow
Email: {simonpj,partain,andre}@dcs.glasgow.ac.uk.

Abstract

Virtually every compiler performs transformations on the program it is compiling in an attempt to improve efficiency. Despite their importance, however, there have been few systematic attempts to categorise such transformations and measure their impact.

In this paper we describe a particular group of transformations — the "let-floating" transformations — and give detailed measurements of their effect in an optimising compiler for the non-strict functional language Haskell. Let-floating has not received much explicit attention in the past, but our measurements show that it is an important group of transformations (at least for lazy languages), offering a reduction of more than 30% in heap allocation and 15% in execution time.

1 Introduction

Consider the following expression:

```
let v = let w = <w-rhs>
        in Cons w Nil
in <body>
```

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

```
let w = <w-rhs>
in let v = Cons w Nil
in <body>
```

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 3.3.) A good compiler should transform the first expression into the second. However, the difference in efficiency is not large, and the transformation between the two is easy — none of the romance of strictness analysis here — and so 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.

The Glasgow Haskell Compiler (GHC) is an optimising compiler for the non-strict purely-functional language Haskell (Hudak et al. [1992]). Its guiding principle is that of *compilation by transformation* — that is, as much as possible of the compilation process is expressed as a series of correctness-preserving program transformations. As we developed the compiler we gradually discovered the importance of well-targeted let-floating. This paper reports on and quantifies this experience.

We make the following main contributions:

- We identify and describe three distinct kinds of let-floating transformations:
  - *Floating inwards* moves bindings as far inwards as possible (Section 3.1).
  - *The full laziness transformation* floats selected bindings outside enclosing lambda abstractions (Section 3.2)
  - *Local transformations* "fine-tune" the location of bindings (Section 3.3).
- We give detailed measurements of their effectiveness (Section 5). We do not simply measure bottom-line performance changes, but also quantify many of the effects predicted in Section 3, and explore the effects of some variants of the basic transformations.

  These measurements are made in the context of a state-of-the-art compiler and a set of substantial programs.

Our results are encouraging: on average, let-floating reduces heap allocation by over 30% and execution time by more than 15%. In the context of an optimising compiler — all the other optimisations are turned on for these measurements, so there are no easy pickings — this is a very worthwhile gain. For particular programs the gains can be spectacular; one program ran twice as fast with let-floating.

We believe that our results are not very GHC-specific, but probably are rather lazy-language-specific. Analogous transformations should be equally successful in other compilers for non-strict languages, but the whole issue is probably much less important for a strict language.

This paper appears in the proceedings of the 1997 International Conference on Functional Programming, Philadelphia (ICFP'97).

Program	*Prog*	$\rightarrow$	*Binding*$_1$; ... ; *Binding*$_n$	$n \geq 1$
Bindings	*Binding*	$\rightarrow$	*Bind*	
		|	`rec` *Bind*$_1$ ... *Bind*$_n$	
	Bind	$\rightarrow$	*var* `=` *Expr*	
Expression	*Expr*	$\rightarrow$	*Expr Atom*	Application
		|	*Expr ty*	Type application
		|	λ *var*$_1$ ... *var*$_n$ `->` *Expr*	Lambda abstraction
		|	Λ *ty* `->` *Expr*	Type abstraction
		|	`case` *Expr* `of` *Alts*	Case expression
		|	`let` *Binding* `in` *Expr*	Local definition
		|	*con var*$_1$ ... *var*$_n$	Constructor $n \geq 0$
		|	*prim var*$_1$ ... *var*$_n$	Primitive $n \geq 0$
		|	*Atom*	
Atoms	*Atom*	$\rightarrow$	*var*	Variable
		|	*Literal*	Unboxed Object
Literal values	*Literal*	$\rightarrow$	*integer* | *float* | ...	
Alternatives	*Alts*	$\rightarrow$	*Calt*$_1$; ... ; *Calt*$_n$; *Default*	$n \geq 0$
		|	*Lalt*$_1$; ... ; *Lalt*$_n$; *Default*	$n \geq 0$
Constr. alt	*Calt*	$\rightarrow$	*Con var*$_1$ ... *var*$_n$ `->` *Expr*	$n \geq 0$
Literal alt	*Lalt*	$\rightarrow$	*Literal* `->` *Expr*	
Default alt	*Default*	$\rightarrow$	`NoDefault`	
		|	*var* `->` *Expr*	

Figure 1: Syntax of the Core language

2 Language framework

Before describing the transformations themselves, we must first introduce the language we use. Its grammar is given in Figure 1, and consists essentially of the (non-strict) second order lambda calculus augmented with `let(rec)` expressions, `case` expressions, and constructors. A program consists of a set of bindings. The value of the program is the value of the variable `main`.

A *value* is an expression in weak head normal form — a variable, literal, constructor application, or lambda abstraction.

An unusual feature of the Core language is that it is based on the polymorphic, second-order lambda calculus (Girard [1971]; Reynolds [1974]), featuring type abstractions (introduced by Λ) and type applications (denoted by simple juxtaposition). Its usefulness to us is that it provides a simple, well-understood way of attaching types to the program which can be maintained through substantial program transformations. We discuss this aspect in Peyton Jones et al. [1993], but here we tend to omit the type abstractions and applications, since they do not play an important role.

Throughout the paper we take a few liberties with the syntax: we allow ourselves infix operators (eg `<e1> + <e2>`), and special syntax for lists (`[]` for *nil* and infix `:` for *cons*). We allow multiple definitions in a single `let` expression to abbreviate a sequence of nested `let` expressions. We use the notation `<e>` to denote an arbitrary expression.

2.1 The operational reading

Another unusual feature of the Core language is that *it has a direct operational interpretation, as well as the conventional denotational semantics.* If we are to reason about the usefulness of a transformation — and this paper contains a great deal of such reasoning — we must have some model for how much it costs to execute it, so an operational interpretation is very desirable. The two most important operational intuitions are as follows:

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

```
let v = factorial 20
in
f v 3
```

   The operational understanding is "first allocate in the heap a thunk (or suspension) for `factorial 20`, and bind it to `v`, then call f passing it the parameters `v` and 3". The language is non-strict, so `v` is not evaluated before calling `f`. Rather, a heap-allocated *thunk* is built and passed to `f`. If `f` ever needs the value of `v` it will force the thunk which, when its evaluation is complete, will *update* (overwrite) itself with its value. If `f` needs the value of `v` again, the heap object now contains its value instead of the suspended computation.

   A `let(rec)` binding may also allocate a value rather than a thunk.

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

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

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

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

case expressions subsume conditionals, of course. The conditional if <cond> <e1> <e2> is written

```
case <cond> of
  True  -> <e1>
  False -> <e2>
```

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

```
f (factorial 20) 3
```

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

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

```
case (reverse xs) of
  []     -> <nil-case>
  (y:ys) -> <cons-case>
```

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

These informal operational notions help identify transformations that might be beneficial, but they do not constitue a formal model of "efficiency". We have started to develop such a model, but technical difficulties remain (Santos [1995, Chaper 9]).

2.2 The costs of a let(rec)-binding

Consider the expression

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

```
let x = f y
in ...x...x...
```

What, precisely, are the costs of the let-binding for x? The performance of modern processors is often limited by their memory bandwidth, so we treat interaction with memory as the major cost:

1. Allocation. The thunk for (f y) has to be allocated and initialised.
2. Evaluation. The first time x is evaluated, the contents of the thunk must be read from the heap into registers.
3. Update. When the evaluation x is started, an update frame must be stored on the stack; when its evaluation is complete, its final value must be written back into the heap, overwriting the thunk identified in the update frame. This value will either be a data value (such as an integer, character, or list cell) or a function value.
4. Further evaluations. Any subsequent evaluations of x will find the evaluated form, but each still entails reading the value from the heap into processor registers.

Sometimes, the right-hand side (RHS) of a binding is already manifestly a value, rather than a thunk:

```
let y = (p,q)
in ...y...y...
```

The let-binding for y is somewhat cheaper than that for x, because no update need be performed.

2.3 Strictness analysis

Strictness analysis is a static program analysis that identifies expressions which are sure to be evaluated. Though strictness analysis is not the subject of this paper[2], it helps to have some understanding of how the results of the analysis are used, because some let-floating transformations are designed to improve the effectiveness of strictness analysis. Suppose we start with the expression

```
let x = f y
in ...x...
```

If the strictness analyser is able to prove that x is sure to be evaluated, and annotates its binding to say so, then we can subsequently make the following transformation (which we call "let-to-case"):

```
case (f y) of
  x -> ...x...
```

The operational reading of the latter form is just as for any case expression: evaluate (f y), bind its value to x, and continue with the code for ...x.... In effect, this form encodes a strict let expression.

The second form is significantly cheaper to execute than the first. In effect, the first-evaluation and update costs of the thunk (items 2 and 3 in Section 2.2) are eliminated, quite a worthwhile saving.

[2] Peyton Jones & Partain [1993] presents detailed measurements of its effectiveness in the same spirit as this paper.

3 What we hope to gain

We are now ready to describe the three let-floating transformations mentioned earlier, and to say what we hope to gain by each. The details of how each is implemented, and what the actual gains achieved, are discussed subsequently.

3.1 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[3]:

- √ *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`, any decent compiler will simply evaluate `y`, increment it and square the result, allocating no thunks at all (Section 2.3).
- √ *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
   (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
  ```

[3] Throughout the paper, advantages are marked with √ and disadvantages with ×. □ indicates moot points.

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
<body>
```

(where `<body>` does not mention `x`) into

```
let y = let x = v+w in ...x...x...
in
<body>
```

(The alert reader will notice that this transformation is precisely the opposite of that given in the Introduction, a point we return to in Section 3.3.) 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 to "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.* Why? Because 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 (but see Section 7). But what if the binding is inside the abstraction to start with? We turn to this question next.

3.2 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 potentially 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 3.1). 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.* We are experimenting with a program analysis which detects some situations in which a lambda abstraction is applied only once (Turner, Wadler & Mossin [1995]).

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 <body>
```

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.

3.3 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=e in b) a          ⟶   (let v=e in b a)

case (let v=e in b) of     ⟶   let v=e
   alts                        in
                               case b of alts

let x = let v=e in b       ⟶   let v=e
in c                           in
                               let x=b
                               in c
```

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 3.4) will split the enlarged group up 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 moves a let outside an application, which 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 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-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 head normal form.* For example, consider the expression:

```
let x = let v = <v-rhs> in (v,v)
in <body>
```

When this expression is evaluated, a thunk will be allocated for `x`. When (and if) `x` is evaluated by `<body>`, 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 = <v-rhs>
    x = (v,v)
in <body>
```

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: <body> may contain a case expression which scrutinises x, thus:

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

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

```
...(<case-rhs>[v/p,v/q])...
```

(We call this the "known-branch" transformation, because it uses information about the scrutinee of a case expression to choose the correct branch of the case.)

√ *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 = <v-rhs>
    x = (v,v)
in <body>
```

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.

It is possible that the let-from-let transformation is worth while even if x is *not* sure to be evaluated. We explore various compromises in Section 5.3.

3.4 Composing the pieces

We have integrated the three let-floating transformations into the Glasgow Haskell Compiler. The full laziness and float-inwards transformations are implemented as separate passes. In contrast, the local let-floating transformations are combined with a large collection of other local transformations in a pass that we call the "Simplifier" (Peyton Jones & Santos [1994]; Santos [1995]). Among the transformations performed by the Simplifier is dependency analysis, which splits each letrec binding into its 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.

4 Implementing let-floating

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

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

1. The first pass annotates each let(rec) binder with its "level number"[4]. 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.*

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

- 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.
- 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 much 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[5] 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 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[6]. We use several heuristics which sometimes decide (conservatively) to leave a binding exactly where it is. If this happens, instead of 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).

5 Results

We measured the effect of our transformations on a sample of 15 "real" programs from our NoFib test suite[7] (Partain [1993]). By "real" we mean that each is an application program 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.

Table 5 gives the absolute performance numbers for each program, compiled with `ghc-0.26 -O`, which includes all the let-floating transformations of this paper. The SPARC instruction counts were collected with SpixTools, kindly provided by Bob Cmelik, then at Sun; we also report how many of the instructions were memory loads and stores, and how many were used by the garbage collector. Instruction-count changes tend to understate execute-time changes, so performance changes might be a bit better In Real Life. The "time" numbers (from a single use of `/bin/time`) are inherently fuzzy; we provide them mainly as a sanity check for the instruction-count numbers. "Allocs" gives the number of words in heap-allocated objects, a widely-used measure which is surprisingly poorly correlated with execution time; we place little faith in it.

"Resid" (residency) gives the average and maximum amount of live data during execution, a number that 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

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

[6] 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.

[7] The results reported in this paper are by no means all that we collected. We pored over results for all 70-ish NoFib programs, not just the "real" ones used here. First we built 14 special versions of prelude libraries (145MB worth). Then we built each NoFib program 16 ways (the extra 2 ways used standard prelude libraries), taking up a total of 1,415MB in disk space. Compile time and run time for all the tests?—we'd rather not think about it!

Program	Allocs. (Kwords)	Resid.(Kwords) Avg.	Max.	Instructions (M) total	%mem	%gc	Time (secs.)
`real/HMMS`	74,233	1,008	1,885	3293.0	35.5%	20.5%	67.8
`real/anna`	5,873	329	542	252.0	46.4%	10.5%	6.4
`real/bspt`	1,015	288	365	26.6	41.0%	18.6%	0.7
`real/compress`	34,735	158	164	918.9	50.4%	14.0%	19.8
`real/fulsom`	51,488	1,177	3,328	1142.0	43.9%	20.2%	25.9
`real/gamteb`	20,053	279	528	594.4	26.7%	16.1%	10.9
`real/gg`	1,628	215	369	47.5	34.2%	17.0%	1.0
`real/hidden`	93,494	211	315	1937.0	38.6%	4.7%	37.3
`real/hpg`	13,440	355	597	344.7	33.6%	24.7%	11.3
`real/infer`	2,531	961	1,935	201.1	46.3%	32.9%	4.8
`real/parser`	2,465	470	860	110.9	41.1%	29.0%	2.5
`real/pic`	1,208	208	296	38.7	32.1%	12.4%	0.9
`real/reptile`	1,108	488	606	28.5	41.3%	15.8%	0.7
`real/rsa`	7,315	3	9	1122.1	8.3%	1.8%	12.9
`real/symalg`	46,960	23,210	46,160	7449.2	3.3%	1.1%	109.1

Table 1: Base case: absolute numbers for what happens with `-O`

sampling means that any "spikes" in live memory usage are unlikely to be missed.

The following sections give the results of measuring the effect of the three transformations (floating inwards, floating outwards, and local floating) separately, followed by measurements of their combined effect. In each case, we try to quantify the effects predicted in Sections 3.1–3.3, as well as measuring the overall effect on execution time and space.

Most results will be given as percentage changes from these `-O` numbers, which we treat as the base case. A positive value means "more than the base case", negative means "less than the base case"; whether that is "good" or "bad" depends on what is being measured. Changes to percentages, e.g. "percentage of instructions in garbage-collection", are given as raw numbers; if the base number was 35.1% and the change is given as -0.7, then the percentage in the changed case is 34.4%. In all tables, a dash indicates a zero figure.

Space precludes listing the results for each individual program. Instead, we report the minimum, maximum, median and mean figures for the whole set. The 'Means' listed are *geometric* means, since they are the means of performance ratios (Smith [1988]). While the mean improvements we find are often modest, it is important to bear in mind the "outliers", reflected in the 'Min' and 'Max' columns. A production-quality compiler should have a lot bullets in its gun; each individual bullet may only target a small class of programs, but it may make a large difference to that class.

5.1 Floating inwards

In this section, we quantify the effects of the floating-inwards transformation. Table 2 shows the effects of switching floating-inwards off. In most of the table, all the other optimisations are left on, except for the second block of the table where strictness analysis is also switched off.

The first block gives the "bottom line". Without floating inwards, we write about 6% more words into memory, and execute somewhat under 1% more instructions. Residency is not affected significantly. The largest improvement came from `fulsom`, with 55.7% more allocations, and `infer`, with 3.9% more instructions.

	Mean	Min.	Median	Max.
Allocs.	6.1%	-0.5%	0.1%	55.7%
Avg. resid.	-2.2%	-29.6%	-	7.6%
Max. resid.	-2.7%	-38.0%	-0.1%	13.3%
Insns.	0.6%	-1.9%	-	3.9%
%mem	-0.1	-	-	-
%gc	+0.1	-	-0.5	-1.8
`noSA-FI` vs `noSA-noFI`				
Alloc'd	5.0%	-	0.2%	54.9%
AvgSize	-0.1%	-2.4%	-	0.8%
Strict bindings found				
`-O`	37.7%	24.0%	37.9%	69.4%
`noFI`	-0.7	-0.4	-0.5	-
Effect on number of enters				
Enters:	0.2%	-0.3%	0.1%	1.6%

Table 2: The effects of turning off floating inwards

Next, we try to quantify the effects predicted in Section 3.1.

- *How many heap objects are not allocated at all as a result of floating inwards, and how is the size of allocated object affected?* There is a complication here: floating inwards helps strictness analysis, and successful strictness analysis also reduces allocation. Hence, to measure the reduction in allocation due only to floating inwards we turned off strictness analysis (`noSA`), and compared how many objects were allocated with and without floating inwards (`FI` and `noFI`).

  The results are given in the second block of Table 2. About 5% fewer objects are allocated when floating inwards is on, and there is essentially no effect on object size (-0.1%).

- *How much is the strictness analyser helped by floating inwards?* The third block of Table 2 shows the proportion of binders that are identified by the strictness analyser as sure to be evaluated in the base case (`-O`), and how this proportion changes when floating inwards is switched off.

  With floating inwards, strictness analysis manages to tag 37.7% of binders as "certain to be demanded".

	Mean	Min.	Median	Max.
Allocs.	10.9%	-	6.4%	66.8%
Avg. resid.	0.1%	-35.8%	0.4%	23.9%
Max. resid.	-0.3%	-7.8%	-0.1%	11.0%
Insns.	7.3%	-0.2%	3.3%	90.5%
%mem	-	-	-0.4	+0.2
%gc	+1.3	-	+0.3	+1.3
Updates:	7.0%	-0.2%	2.9%	60.1%

Table 3: Effect of turning off the full laziness transformation

Things are slightly worse without floating inwards (-0.7%). So floating inwards does help the strictness analyser, but not much.

- *How many evaluations are eliminated by floating inwards?* The final block of Table 2 concerns the number of times a heap closure (be it a thunk, data value or function value) was evaluated — or "entered" — during the execution of the program. Again, floating inwards has a small but beneficial effect (mean 0.2%, max 1.6%).

5.2 Full laziness

Next we turn our attention to the full laziness transformation. The results are summarised in Table 3, which shows the effect of switching off full laziness while leaving all other optimisations on.

Overall, full laziness buys a reduction of 10% in allocation, and 7% in instructions executed. We would expect the number of updates to decrease, because of the extra sharing of thunks caused by full laziness, and indeed it does go down, by about 7%.

As the "Max" column shows, one program (`hidden`, a hidden-line removal algorithm) is dramatically improved by full laziness. Leaving bindings unnecessarily stuck inside an inner loop is a Really Bad Idea.

Does full laziness affect a very few expressions in each program, or is it widely applicable? Initially we guessed the former, but the measurements in Table 4 contradicts this guess. The table counts how many bindings were floated at all by full laziness. In order to make comparable the figures for different programs, we (somewhat arbitrarily) normalised them to the number of lambda groups in the program. We also identify separately bindings which float to the top level — these are just constant expressions — and those that float to some other place. For example, in `HMMS`, on average 1.4 constant bindings and 0.3 non-constant bindings floated past each lambda group.

As you would expect, floated constant bindings are about 5 times as common as floated non-constant bindings. Overall, we find the figures in Table 4 surprisingly high, and we believe that there is probably scope for increased selectivity in the full laziness transformation.

5.3 Local transformations

Next, we compare four different strategies for local let-floating:

Float outwards:	Lambda groups	Top-level ratio	Inner ratio
`real/HMMS`	435	1.4	0.3
`real/anna`	2,221	0.7	0.1
`real/bspt`	290	0.8	0.1
`real/compress`	19	0.3	-
`real/ebnf2ps`	487	1.3	0.4
`real/fluid`	468	1.0	0.2
`real/fulsom`	217	1.0	0.3
`real/gamteb`	52	2.1	-
`real/gg`	363	0.9	0.3
`real/grep`	173	0.5	0.1
`real/hidden`	269	0.3	0.1
`real/hpg`	310	0.6	0.3
`real/infer`	258	0.6	0.3
`real/lift`	151	0.6	0.2
`real/maillist`	37	0.6	0.2
`real/parser`	424	0.9	0.3
`real/pic`	124	1.0	0.6
`real/prolog`	185	0.6	0.2
`real/reptile`	246	1.5	0.1
`real/rsa`	24	1.9	0.1
`real/symalg`	148	1.8	-
`real/veritas`	1,047	2.0	0.2
MEAN		0.9	0.2
MIN		0.3	-
MEDIAN		0.9	0.2
MAX		2.1	0.6

Table 4: Details of full laziness floats

None. Do no local let-floating at all.

Strict. Bindings are floated out of strict contexts only; namely, applications, `case` scrutinees, and the RHSs of *strict* `lets`. These floats cannot increase the number of closures allocated, so "strict" should out-perform "none".

Base case (`-O`). Like "strict", but in addition a binding is floated out of a `let(rec)` RHS if doing so would reveal a value.

Always. Like "strict", but any binding at the top of a `let(rec)` RHS is floated out.

Table 5 shows the effects of these four strategies; as always we report percentage changes from the base case (`-O`). Overall, the base case consistently out-performs the three other variants (which is, of course, why we make it the default choice). "None" is, unsurprisingly, terrible (15% more allocation, 8% more instructions, 8% worse peak residency). "Strict" is relatively dire (5% more allocation, 6% more instructions, 6% worse peak residency); "always" has more gentle effects (e.g., only 0.5% more instructions) but cannot be said to justify its more aggressive floating strategy.

Note also that "strict" is prone to very unpleasant "outliers" (e.g., 70%+ residency degradation); moreover, these "outlier effects" are spread across a range of programs (it isn't just one program being hit very badly).

We now compare the three defensible strategies for floating bindings out of `let(rec)` RHSs, using the effects predicted in Section 3.3. (Since "none" is so obviously terrible we don't consider it further.)

	Mean	Min.	Median	Max.
Allocs.:				
always	3.4%	0.1%	1.3%	15.5%
strict	5.3%	0.1%	3.2%	16.1%
none	15.7%	0.2%	10.1%	118.2%
Avg. resid.:				
always	-0.8%	-30.8%	0.4%	29.0%
strict	3.9%	-19.9%	1.0%	75.8%
none	5.7%	-19.9%	4.1%	50.9%
Max. resid.:				
always	-0.5%	-38.5%	0.5%	19.1%
strict	6.1%	-19.8%	0.8%	71.1%
none	8.3%	-19.8%	4.0%	69.0%
Insns.:				
always	0.5%	-2.1%	-	3.2%
strict	6.0%	-	4.7%	18.6%
none	8.7%	0.1%	5.7%	35.3%
%mem:				
always	+0.7	+1.2	-	+0.1
strict	-	-	+0.3	-0.2
none	+0.4	+0.6	+0.2	-0.2
%gc:				
always	+1.2	+0.7	-0.5	-
strict	+0.6	-	-0.3	-2.4
none	+1.6	+0.7	+0.3	-2.3
Updates:				
always	2.4%	-12.8%	0.5%	33.7%
strict	28.2%	2.0%	26.9%	103.1%
none	30.7%	3.1%	30.5%	103.1%
AvgSize:				
always	-2.3%	-7.5%	-1.2%	0.1%
strict	1.0%	-3.6%	0.7%	6.6%
none	-1.3%	-12.7%	-0.6%	5.7%
Heap Chks:				
always	-0.3%	-14.7%	-0.2%	19.3%
strict	14.2%	0.7%	10.1%	35.7%
none	25.5%	0.8%	16.8%	96.6%
Known branches:				
always	-3.8%	-14.8%	-3.2%	-
strict	-2.5%	-20.0%	-0.9%	9.3%
none	-7.5%	-36.7%	-5.5%	-
Enters:				
always	1.5%	-0.1%	0.4%	8.0%
strict	2.5%	-1.1%	1.4%	16.9%
none	6.5%	-0.8%	4.5%	32.0%

Table 5: Effects of local let-floating

- *How many extra allocations are performed? Is the average closure size increased or decreased?* Allocation is up 3% in the "always" case, which is unsurprising, because floating a binding out of a `let(rec)` RHS will cause that binding to be allocated when it might not otherwise be.

  It is more surprising that allocation also rises in the "strict" case, which is *less* aggressive than the base case. The reason turns out to be that the strictness analyser is foxed by definitions like this one:

  ```
  f = let x = <x-rhs> in \y -> <f_body>
  ```

  With the "strict" strategy the binding for `x` may not be floated out of that for `f`. Whilst the strictness analyser spots that `f` is strict, it does not exploit that fact because doing so naively would involve recomputing `<x-rhs>` on each call of `f` (Peyton Jones & Launchbury [1991]). Since our default `-O` floating strategy dominates "strict" in other ways, and solves this difficulty by floating `x` out one level, we have not made the strictness analyser able to deal with it directly.

- *How many updates are saved?*

  It is no surprise that updates are more common (28%) with the less aggressive "strict" strategy, because fewer `let(rec)` RHSs are values which require no update.

  It is slightly surprising that "always" seems to make updates increase again (2.4%). Why? Perhaps because it undoes floating inwards, and hence gives less good strictness analysis and hence more updates.

- *How many known-branch transformations are eliminated?* Both "strict" and "always" reduce the number of known-branch transformations. Since this transformation is a guaranteed win, this reduction is undesirable. It is easy to explain why "strict" offers fewer known-branch opportunities, because the non-floated bindings may hide a constructor (Section 3.3). We do not yet understand why "always" has the same effect; one would expect the reverse.

- *How many fewer heap-allocation checks are performed?*

  The "always" strategy does indeed "clump" lets so that we do fewer heap checks, but it is a mere 0.3% improvement over the base case. The downside of the "strict" strategy is quite a bit worse (14.2%).

5.4 Overall results

Table 6 summarises the total effect of switching all three floating transformations off (using "strict" as the "no local floating" case). The total reduction in allocation due to floating is 34.4%, which is close to the sum of the gains measured for each transformation separately ($6.1 + 10.9 + 15.7 = 32.7\%$).

The reduction in instruction count is 16.4% (with measured-but-fuzzy time savings of 18.7%). This, too, is not far from the sum of the gains for each transformation independently ($0.6 + 7.3 + 8.7 = 16.6\%$).

We have more than once found that the effects of an optimisation are drastically reduced when it is done along with a

Program	Allocs.	Resid. Avg.	Max.	Instructions total	%mem	%gc	Time
`real/HMMS`	37.4%	-1.3%	-1.2%	7.5%	-0.4	+1.8	11.9%
`real/anna`	32.3%	7.3%	3.8%	12.0%	-0.8	+4.3	12.5%
`real/bspt`	33.9%	-27.9%	-34.2%	13.2%	+0.6	+1.9	14.3%
`real/compress`	24.8%	-18.5%	-18.5%	14.1%	-	-0.7	15.2%
`real/fulsom`	55.5%	5.3%	10.9%	31.0%	-1.8	+2.7	33.6%
`real/gamteb`	27.7%	-8.1%	-7.9%	6.4%	-	+0.1	6.4%
`real/gg`	47.9%	-41.7%	-5.5%	19.8%	+1.1	+2.1	30.0%
`real/hidden`	90.2%	25.4%	0.1%	110.3%	-	+6.4	107.2%
`real/hpg`	24.7%	-12.8%	-14.4%	11.4%	+0.6	+1.1	4.4%
`real/infer`	66.7%	1.3%	3.0%	4.1%	+0.1	-2.2	4.2%
`real/parser`	62.4%	9.3%	10.8%	23.4%	-1.2	+6.2	20.0%
`real/pic`	26.7%	2.4%	-9.2%	13.6%	+1.0	+0.2	33.3%
`real/reptile`	14.7%	0.2%	-1.1%	7.6%	-0.8	-1.4	14.3%
`real/rsa`	1.3%	-1.0%	-3.8%	0.1%	-	-	0.8%
`real/symalg`	0.2%	-0.2%	0.1%	1.4%	+0.6	+1.3	2.7%
MEAN	34.4%	-5.5%	-5.2%	16.4%	+0.3	+2.2	18.7%
MIN	0.2%	-41.7%	-34.2%	0.1%	+0.6	+0.7	0.8%
MEDIAN	32.3%	-0.2%	-1.2%	12.0%	-	+0.1	14.3%
MAX	90.2%	25.4%	10.9%	110.3%	-	+2.3	107.2%

Table 6: Bottom line: how "no floating" compares with `-O`

slew of others, because several transformations were hitting the same targets. In this case, however, the fact that the three let-floating transformations "add up" reasonably well means that they are hitting genuinely different targets.

We made some measurements of the effect on compile time of the floating transformations. Generally, compile times are a few percent *worse* with no floating at all, presumably because other parts of the compiler (such as the code generator) have to work harder. Certainly, none of the floating transformations cause a noticeable increase in compile time.

All these bottom-line figures should be taken with a pinch of salt. Since the rest of the compiler was written in the expectation that at least the more basic let-floating transformations were implemented, the figures probably overstate the penalty for turning them off.

6 Related work

Using correctness-preserving transformations as a compiler optimisation is, of course, 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]).

Perhaps because it seems such a modest transformation, however, there are few papers about let-floating, except in the context of hoisting invariants out of loops. Appel's work on "let-hoisting" in the context of ML is the only substantial example we have uncovered (Appel [1992, Chapter 8]). He identifies both floating inwards ("hoisting downwards") and floating outwards ("hoisting upwards"). Because ML is strict, though, floating outwards is only sure to save work if the loop is guaranteed to execute at least once, which restricts its applicability. The local let-floating transformations are done automatically by the CPS transform — because the language is strict all three local strategies coincide. Appel reports some outline results that show instruction-count improvements on the order of 1% for hoisting down and 2% for hoisting up.

7 Contributions

We have described a group of three related transformations that each attempt to improve the location of `let(rec)` bindings in a purely-functional program. We found it very helpful to identify three independent flavours of let-floating. Our results suggest that they really are independent: they aren't just various ways to get the same optimisation benefits.

We have measured the effects of the transformations, both on the "bottom line" and on more insightful internal measures. The improvements we obtain are modest but significant. Any serious compiler for a non-strict language should implement (a) local floating to expose values (the less aggressive "strict" strategy has all sorts of unfortunate effects); (b) floating out of constants. The benefits of the other transformations — namely floating inwards and complete full-blown full laziness — are more modest.

One lesson that we learned repeatedly is that it is very hard to predict the interactions between transformations. A major benefit of performing all these measurements is that they threw up many individual cases where a usually-useful transformation was counter-productive. Investigating these cases led us to some new transformations, and a considerable amount of fine-tuning of the existing one. So far as let-floating goes, the net result is fairly good: collectively, the let-floating transformations never increase instruction the count, and seldom do so individually.

We are now adding a linear-type inference pass to GHC, to spot lambda abstractions that are guaranteed only to be applied once (Turner, Wadler & Mossin [1995]). This information can increase opportunities for floating inwards, and reduce unnecessary floating outwards. It is also useful for other reasons, such as arity expansion and inlining (Gill

[1996]).

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

SL Peyton Jones & J Launchbury [Sept 1991], "Unboxed values as first class citizens," in *Functional Programming Languages and Computer Architecture, Boston*, Hughes, ed., LNCS 523, Springer Verlag, 636–666.

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

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

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

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

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

JE Smith [Oct 1988], "Characterising computer performance with a single number," *Communications of the ACM* 31, 1202–1207.

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