# Measuring the effectiveness of a simple strictness analyser

Simon Peyton Jones
Department of Computing Science, University of Glasgow
simonpj@dcs.glasgow.ac.uk

Will Partain
Department of Computing Science, University of Glasgow
partain@dcs.glasgow.ac.uk

**Abstract**

We describe a simple strictness analyser for purely-functional programs, show how its results are used to improve programs, and provide measurements of the effects of these improvements. These measurements are given both in terms of overall run-time, and in terms of internal operations such as allocations, updates, etc.

Despite its simplicity, the analyser handles higher-order functions, and non-flat domains provided they are non-recursive.

Ths paper appears in *Functional Programming, Glasgow 1993*, ed Hammond and O'Donnell, Springer Workshops in Computer Science, 1993, pp201-220.

## 1 Introduction

A lot has been written about strictness analysis for non-strict functional programs, usually in the hope that the results of the analysis can be used to reduce run-time. On the other hand, few papers present measurements of how well it works in practice. Usually, all that is presented are the run-times of a variety of (usually small) programs, with and without strictness analysis enabled (eg Smetsers et al. [1991]). The goal of this paper is to provide detailed quantitative insight about the effectiveness of a simple strictness analyser, in the context of a state-of-the art compiler, running serious application programs.

## 2 What might we hope to gain

Before going further, we need a clear model of what one might hope to gain from strictness analysis. The discussion of this section is intended to be generic, applying equally to any implementation technology. Accordingly, we refrain from introducing details about *how* we implement anything; instead we concentrate on *what* gains are possible.

## 2.1 The classic case: integers

Let's start with the standard example, a strict function with an `Int` argument. Suppose `f` is strict, and has type

```
f :: Int -> Int
```

Consider what happens for the call `(f (x+1))`. Without the knowledge that `f` is strict, we would have to build in the heap a thunk[1] for `(x+1)` and pass it to `f`.

If `f` is known to be strict, we can instead evaluate `(x+1)` before the call. Better still, *we can pass the argument unboxed*, since it is by now certainly an evaluated `Int`. We need a different version of `f`, which we call `fw` to abbreviate "`f`'s worker", with type

```
fw :: Int# -> Int
```

where `Int#` is the type of unboxed integers. Operationally, an `Int` is represented by a pointer to a heap object, which contains either an evaluated integer, or possibly an unevaluated thunk. An `Int#` is represented by the integer itself, not a pointer at all.

The big gain is that no thunk for `(x+1)` is ever created, which means that:

- We may save a heap-overflow check.
- We save the costs of writing the thunk into the heap memory.
- We save the costs of reading it back in again when it is evaluated in `f`.
- We save the costs of updating the thunk with its final value, and of the stack manipulations associated with updates[2].
- We save the amortised garbage collection costs of the thunk.

To give an idea of scale, these additional costs amount to around 25 instructions per argument, most of which are memory references. It is clearly a big win to replace these instructions with a single instruction to place the argument in a suitable register! The gains are compounded if `f` is recursive, because we save all these costs every time around the loop.

Is there *always* a gain, though? Suppose the argument to `f` is itself just an argument to the enclosing function. For example:

```
foo False x = 0
foo True  x = f x + 1
```

(Here, `foo` isn't strict in `x`, so `x` has to be passed boxed to `foo`.) In this case there is actually no gain from evaluating `x` before calling `f`, but there is no harm either. Even in cases like this we may still gain by avoiding redundant evaluations. Suppose instead that the second equation for `foo` was:

[1] We use the term "thunk" for an as-yet unevaluated closure; when evaluated it will usually be overwritten by its (weak) head normal form.

[2] A separate analysis might be able to discover that `f` would evaluate its argument at most once. In this case, the caller can mark the thunk as non-updatable, thus saving the cost of performing the update. But that's another story, and the costs of writing the thunk into the heap, and reading it back out, are unchanged.

```
foo True x = f x + f x
```

Now, it is possible to evaluate `x` *just once*, and pass the unboxed value to each of the two calls to `f`. Exactly the same number of thunks are built, but there is one fewer evaluation: we say that we have "eliminated a redundant EVAL".

To summarise, we may exploit strictness analysis in three ways:

1. by avoiding the creation and updating of thunks;
2. by manipulating unboxed values, instead of heap-allocated boxed values;
3. by eliminating redundant evaluations.

## 2.2 Other single-constructor types

The same idea extends to unboxing any algebraic data type with just one constructor[3]. A particularly common family of such data types is the tuple types. For example, if `g` is strict, and has type

```
g :: (a,a) -> a
```

then `g`'s worker, `gw` will take the *components* of the constructor:

```
gw :: a -> a -> a
```

As before, the pair is passed unboxed, as its two separate components. In a call `(g <arg>)`, where `<arg>` is an arbitrary expression, we can evaluate `<arg>` before the call, and pass the components to `gw` rather than building a thunk for `<arg>` which is later evaluated by `g`.

If we find a call with an explicit tuple argument, eg `(g (<arg1>,<arg2>))`, then we are laughing: we can just transform the call to `(gw <arg1> <arg2>)`.

Our implementation of Haskell adds extra *dictionary arguments* to overloaded functions. A dictionary is just a tuple of values, so many more functions end up with tuple arguments than those written explicitly by the programmer. (A separate pass, which specialises overloaded functions, may eliminate these dictionaries, however.)

## 2.3 Unboxing multi-constructor types

With multi-constructor types, such as lists, matters are rather murkier. Suppose we know that a function `h` is strict in its list argument, where `h` has type:

```
h :: [Int] -> Bool
```

How could we use this information? One possibility would be to have *two* workers for `h`, with types:

```
hwNil  :: Bool                   -- The Nil case
hwCons :: Int -> [Int] -> Bool   -- The Cons case
```

[3] As we'll see later, `Int` is an example of such a type.

Given a call `(h e)` we can evaluate `e` and call `hwNil` or `hwCons` depending on what it turns out to be. Unfortunately this scheme turns sour on us if the function has several arguments with multi-constructor types, because there's an exponential blowup in the number of versions of `h` required. We have not tried this option at all.

Another possibility would be to have some way of passing a variable-sized argument, along with a tag to say which constructor is meant. This is tricky territory for the code generator, and again we have not tried it.

In any case, it is far from clear what gains might be expected from either approach. If the data type is recursive, and `h` is recursive, then all we are doing is moving the *topmost* evaluation of the argument from `h` to the call site.

In short, unboxing strict arguments of multi-constructor type seems complex and the returns are debatable. There is one obvious exception to this pessimistic conclusion: enumeration types, such as `Bool`, have several constructors but all of them are nullary. Enumeration types have an obvious unboxed representation as a small integer tag. We plan to exploit this, but have not yet done so.

### 2.4 Early evaluation of multi-constructor types

Even if we have to pass a multi-constructor typed argument in boxed form, there might still be gains from evaluating it early. In particular, consider the call `(h <arg>)` where `<arg>` is not a variable or literal. Without strictness analysis, we have to build a thunk for `<arg>` and pass it to `h`. Knowing that `h` is strict, we could evaluate `<arg>` in-line, allocate a suitable list cell (*cons* or *nil*), and pass that to `h`. The saving here is that we don't need to write the thunk into the heap, read it back and then perform the update; instead we simply write the final value into the heap. Howe's paper in this proceedings concentrates on precisely this point (Howe & Burn [1993]).

If the argument of a call is a variable, eg `(h x)`, it depends on fine details of the implementation whether it is worth evaluating `x` before the call. Recall that in the case of single-constructor types the benefit was that the evaluation might be shared across two calls. In the multi-constructor case this benefit might also accrue *if evaluation and unpacking are two separate operations*, as they are in the G-machine. Then evaluation can be moved to the call site, while unpacking remains in the called function. In the STG-machine, evaluation and unpacking are performed simultaneously, so there is no benefit.

### 2.5 Improving `let` bindings

Consider the expression

```
let x = <x-rhs> in <body>
```

and suppose that strictness analysis tells us that `<body>` is sure to evaluate `x`. Can we make use of this information? Yes, of course. Just as in the case of a strict function, we evaluate `x` immediately instead of building a thunk for it, and waiting for `<body>` to evaluate it. For reasons which will become apparent, we call this the `let`-to-`case` transformation.

### 2.6 Non-flat domains

So far we have concentrated exclusively on "flat" strictness, which answers the question as to whether a function is strict in a particular argument position or not. Much work has been done on non-flat domains: can we take advantage of it? This question splits into three:

1. For *non-recursive types with just one constructor*, such as tuples, we should certainly be able to "see" inside the constructor to the strictness of its components. For example, we can do more with the function

   ```
   f (x,y) = x+1
   ```

   than to say that it is strict in its argument; it is obviously strict in the first component of the tuple.

2. For *non-recursive types with more than one constructor* it might be possible to do something similar, but it seems quite a bit more complicated. In our analyser we make no attempt to discover strictness in the components of multi-constructor types.

3. For *recursive types*, much theoretical work has been done relating the degree of evaluation performed on the result of a function to the degree of evaluation of its arguments (Burn [1990]; Wadler & Hughes [1987]). For example, the `append` function is strict in the spine of its arguments if it is called in a context which requires the spine of its result. We make no attempt to compute or exploit such information, for two reasons. First, it is a lot more work to compute the information. Second, the results of exploiting it are not always beneficial. For example, the version of the `append` function suitable for a spine-strict context will compute the whole of the result before returning any of it, which might have a terrible effect on the peak space requirement of the program.

   There is very little published work which attempts to measure the effects of exploiting strictness in recursive data types. Finne & Burn [1993] use "evaluation transformers" to evaluate lists more strictly (in the same spirit as the work described here), but for sequential implementations they find few performance benefits, and occasional large costs. In contrast, Hall [1993] uses strictness information over list types to guide a new transformation which uses a more efficient list representation where possible. The transformation gives substantial performance benefits where it is applicable.

## 3 A simple strictness analyser

Now that we know how we can exploit strictness analysis, we will describe our simple strictness analyser itself. Our general approach has been *to try to gain a large fraction of the possible winnings with a small fraction of the effort.* To this end, our strictness analyser is simple, but apparently quite effective. (It is of course hard to be quantitative about such a claim, since we have not implemented a more sophisticated one with which to compare it.)

The process of detecting and exploiting strictness is split into three stages:

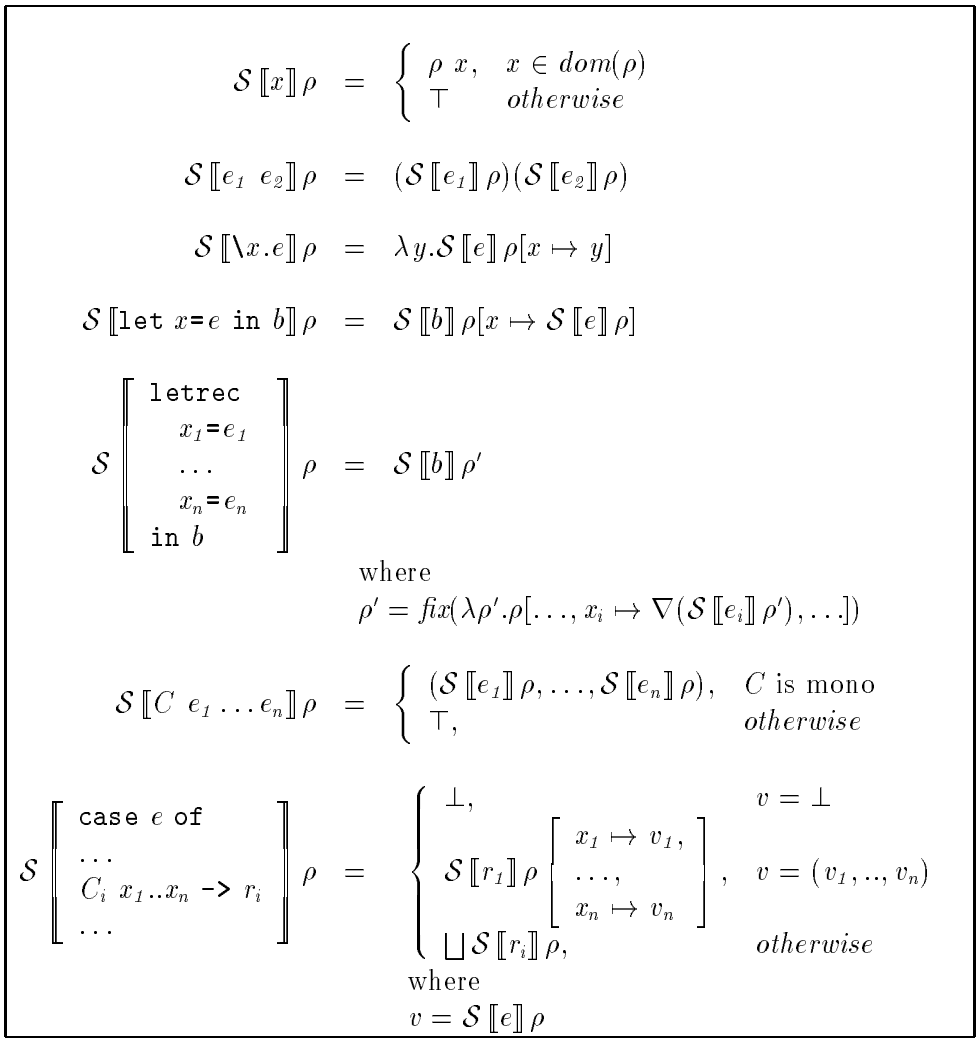


Figure 1: The abstract interpretation

1. First, a simple abstract interpretation is used to annotate the binding occurrence of each variable with (a) an indication of whether or not it is sure to be evaluated, and (b) in the case of `let(rec)` bound variables, an indication of its strictness properties.
2. Next, a simple pass uses these annotations to make local transformations of the program.
3. Finally, a general program-transformation system is applied to the resulting program, to propagate the effects globally.

The first stage is described in the rest of this section, while the second two stages are described subsequently, in Section 4.

### 3.1 The abstract interpretation

Our strictness analyser uses abstract interpretation. The abstract domain contains top, bottom, functions, and finite products:

$$D = \mathbf{1} \; + \; (D \to D) \; + \; \sum_i D^i$$

(Here, $\mathbf{1}$ is the one-point domain, $+$ is lifted sum, and $D^i$ is ordinary (non-lifted) product.) The abstract interpretation itself is given in Figure 1. The syntax of the language is conventional: lambda calculus together with `let`, `letrec`, `case`, and constructors from algebraic data types. We assume that the source program is well typed by the time it reaches the strictness analyser.

There are several features worthy of note:

- The abstract interpretation is higher order, along the lines of Burn, Hankin & Abramsky [1986].
- The product space in the abstract domain is used only for constructors from data types with just one constructor (Figure 1 calls such constructors "mono"). For constructors from multi-constructor types we use the simple two point domain only.
- The interpretation of `case` expressions takes advantage of a product-space value. (In this case there can be only one alternative, of course.)
- `let` expressions are handled with no approximation at all, including the case where the variable has a functional type.
- The interpretation of `letrec` uses the fixpoint operator, of course, but it also uses a *widening operator*, $w$, which is the subject of the next section.

Using this abstract interpretation it is straightforward to annotate each `let`-bound or lambda-bound variable with its strictness properties.

### 3.2 Finding fixpoints

Our analyser uses a crude but fast approximation technique for finding fixpoints. The main payoff is that it converges in worst case time $O(N^2)$, where $N$ is the number of arguments to the function being fixpointed. The square law comes from the fact that at most $N$ iterations are required, each of which requires $N$ evaluations to compute.

Suppose we want to find the fixpoint of a functional $F$. The usual idea is to compute the chain of approximations $\bot, F(\bot), F^2(\bot), \ldots$, and stop when $F^k(\bot) = F^{k+1}(\bot)$, at which point $F^k(\bot)$ is the least fixpoint of $F$. The trouble is that computing equality between successive approximations is horribly expensive; a variety of solutions have been proposed, for example Hughes [1992], Peyton Jones & Clack [1986].

Our approach is simple. Instead of finding the fixpoint of $F$, we will find the fixpoint of $\nabla F$, where $\nabla F(x) = \nabla(F(x))$, and $\nabla$ is a *widening operator*. A widening operator makes things

bigger[4]; formally, $\nabla$ is a widening operator iff

$$\forall x.\ x \sqsubseteq \nabla(x)$$

The key idea is this: *we choose* $\nabla$ *so that it is easy to compare successive iterations,* $(\nabla F)^k(\bot)$ *and* $(\nabla F)^{k+1}(\bot)$ *for equality.* Why does this do us any good? Because of this theorem:

**Theorem 1:** *fix* $F \sqsubseteq$ *fix* $\nabla F$

That is, *the fixpoint of* $\nabla F$ *is a safe approximation (from the point of view of strictness analysis) to the fixpoint of* $F$. The proof is in the Appendix.

### 3.2.1 Choosing a widening operator

Now, how do we choose the widening operator $\nabla$? There are a variety of choices, but the one we use is to characterise a widened function by giving a "demand" for each argument, saying how much evaluation the function guarantees to perform on that argument. A "demand" is one of the following:

| | |
|---|---|
| L | Lazy; no evaluation of this argument is guaranteed. |
| S | Strict; the function is strict in this argument, but its type is a multi-constructor type, function, or type variable. |
| U($d_1 \ldots d_n$) | Unpack; a strict argument of a single-constructor type, to be passed unboxed. The demands which can be guaranteed for the components are given by $d_1 \ldots d_n$. |

For example, if `f` is defined like this:

```
f x y z = if z then x else y
```

then `f`'s "strictness signature", which gives s safe demand for each argument, is "LLS". We think of `f`'s strictness signature as specifying $\nabla$(`f`), a very crude uppper (and hence safe) approximation to the true abstract value of `f`:

$$\begin{array}{lcl} \nabla(\texttt{f})\ x\ y\ \bot & = & \bot \\ \nabla(\texttt{f})\ x\ y\ z & = & \top, \text{ if } z \neq \bot \end{array}$$

Notice that the "joint strictness" between `x` and `y` is lost by the widening process.

In general, suppose $f$ is a function of $n$ arguments. A *safe demand*, $d_i$, for argument $i$ has the property that:

$$\forall x \in \theta(d_i).f\ \top_1\ \top_2\ \ldots\ x\ \ldots\ \top_{n-1}\ \top_n = \bot$$

[4] The cognoscenti will know that widening operators have a slightly tighter definition than this, but this one is sufficient for our purposes.

where $\theta(d_i)$ is defined as follows:

$$\begin{array}{lcl} \theta(L) & = & \emptyset \\ \theta(S) & = & \{\bot\} \\ \theta(U(d_1, \ldots, d_n)) & = & \{\bot\} \cup \{(t_1, \ldots, t_n) \mid t_1 \in \theta(d_1) \vee \ldots \vee t_n \in \theta(d_n)\} \end{array}$$

Informally, if $d_i$ is a safe demand for argument $i$, then $\theta(d_i)$ is a set of values of argument $i$ which are certain to make the function diverge.

A strictness signature for `f` is a sequence of safe demands $d_1 d_2 \ldots d_n$. Then $\nabla(\texttt{f})$ is easy to define:

$$\nabla(\texttt{f}) = \lambda x_1 \ldots x_n . \mathit{if}\ (x_1 \in \theta(d_1))\ \vee\ \ldots\ \vee\ (x_n \in \theta(d_n))\ \mathit{then}\ \bot\ \mathit{else}\ \top$$

It is an easy theorem that $\nabla$ is indeed a widening operator.

Given the abstract version of a function, it is easy to find a safe demand for each argument, by "probing". First, evaluate the function with that argument bound to $\bot$ and all the others bound to $\top$; if it returns $\bot$ then S is a safe demand, otherwise U is. If S is a safe demand and the argument is of a single-contructor type we can try probing with $(\bot, \top, \ldots, \top)$, $(\top, \bot, \top, \ldots, \top)$, and so on, to find a safe demand for each component. The strictness analyser can then widen a function by (a) probing it to compute its strictness signature and then (b) using the strictness signature to define the widened function.

So, finally, to find the fixpoint of a function, compute a chain of widened approximations, and stop when two iterations have the same strictness signature.

Working through this theory, which was done subsequent to the implementation, actually revealed several bugs in the implementation, one of which was in the widening process. Theory actually helps!

# 4 Exploiting the results of strictness analysis

The results of strictness analysis are often exploited by deeply mysterious processes within the code generator of the compiler. Our approach is instead:

- to express the results of strictness analysis by means of simple, local program transformations, and
- to propagate the effects of these transformations using the compiler's general program-transformation system which is required in any case.

To do this requires a language in which the evaluation of (for example) integers is explicit. We do this by making use of *unboxed types*; the ideas are sketched below, but full details can be found in Peyton Jones & Launchbury [1991].

### 4.1 Unboxed types

In Glasgow Haskell, unboxed types like `Int#` are (nearly) first-class citizens[5]. Indeed, the `Int` type is built from `Int#`, using an ordinary data type declaration:

```
data Int = I# Int#
```

Consider again the call `f (x+1)`. The idea of evaluating the `(x+1)` before the call, and passing it unboxed, can now be expressed like this:

```
case x of
I# x# -> case x# +# 1# of
           a# -> fw a#
```

The outer `case` evaluates `x`, and extracts its unboxed component `x#`. The inner `case` is the STG language's way of saying[6] "add `x#` to `1#` and call the result `a#`". Then `a#` is passed to `fw`.

### 4.2 Workers and wrappers

Suppose that the strictness analyser found that `f` is strict in its `Int` argument `x`, where `f` is defined like this:

```
f x = <rhs-of-f>
```

To express this fact, a local program transformation is made which splits `f` into two functions: a *wrapper* and a *worker*, thus:

```
f x   = case x of
          I# x# -> fw x#
fw x# = let x = I# x#
          in <rhs-of-f>
```

Now, the global program-transformation phase can unfold all calls to `f`, but not `fw`, which has precisely the effect of moving the argument evaluation to the call site.

Notice the somewhat curious re-construction of `x` from `x#` in the body of `fw`. Does this mean that the argument is taken apart by the wrapper only to be reconstructed by the worker? The answer is that such reconstruction does not usually take place. To see the common case, suppose `<rhs-of-f>` was `x+x`. Substituting this into `fw`, and unfolding the definition of `+`, gives

```
fw x# = let x = I# x#
          in case x of
             I# x1# -> case x of
                         I# x2# -> case x1# +# x2# of
                                     a# -> I# a#
```

[5] The only respect in which they are not first-class is that unboxed types cannot instantiate polymorphic type variables (Peyton Jones & Launchbury [1991]).

[6] We use `1#` rather than just `1`, and `+#` rather than `+` because the values and operations involved are the unboxed ones.

Now, both `case` expressions are scrutinising a variable which is directly bound to a `I#` constructor, so they can be eliminated in favour of binding `x1#` and `x2#` to `x#`. Now `x` is not mentioned at all, so it can be dropped, giving the satisfactory result:

```
fw x# =  case x# +# x# of
         a# -> I# a#
```

### 4.3 The `let` to `case` transformation

As remarked in Section 2.5 it should be possible to generate better code for a `let` binding which binds a variable which is sure to be evaluated in the body of the `let`. To express this, we can make the transformation:

```
let x = <rhs-of-x>      ===>      case <rhs-of-x> of
in <body>                         x -> <body>
```

This is called the `let`-to-`case` transformation. In our implementation, the original `let` would build a thunk for `<rhs-of-x>` while the `case` will instead evaluate it in-line[7]. A heap allocation does still take place: if `<rhs-of-x>` was a pair, for example, the pair would be allocated in the heap, and bound to `x`[8]. The gain is that the thunk does not need to be written out, read back in later, and updated.

The `let`-to-`case` transformation implements the optimisation discussed in Section 2.4. Given a function application `(h <arg>)`, for some non-atomic expression `<arg>`, we first transform to

```
let a = <arg> in h a
```

and then use `let`-to-`case` to obtain

```
case <arg> of
a -> h a
```

However, in the case of single-constructor types we can do better. For example, if `x` is a pair we can transform like this:

```
let x = <rhs-of-x>  ===>   case <rhs-of-x> of
in <body>                  (p,q) -> let x = (p,q) in <body>
```

Why is this better? Because the construction of `x` as a pair is then visible in `<body>` which often results in the elimination of one or more `case` expressions. A particularly important example of this `case`-elimination concerns lazy pattern matching. Suppose we start with the Haskell definition

```
f x y = if x then p else q
      where
```

[7] In Haskell, `case e of x -> b` is the same as `let x = e in b`, but not in our compiler's internal language, in which `case` is always strict.

[8] Incidentally, precisely because it needs to heap-allocate the value returned by `<rhs-of-x>`, our implementation will only handle a `case` with a single-variable alternative if the type of `x` is a statically determinable data type.

```
        (p,q) = h y
```

This translates naively to:

```
f x y = let t = h y
            p = case t of (p,q) -> p
            q = case t of (p,q) -> q
        in if x then p else q
```

The pattern binding for `(p,q)` gives rise to three thunks, one for each of `t`, `p` and `q`, which is horribly inefficient. Now, the strictness analyser will discover that `t` is sure to be evaluated, even though neither of `p` and `q` are. Using the `let`-to-`case` transformation gives:

```
f x y = case (h y) of
        (p1,q1) -> let p = case t of (p,q) -> p
                       q = case t of (p,q) -> q
                   in if x then p else q
```

Very simple automatic transformations can now eliminate the inner `case` expressions to give:

```
f x y = case (h y) of
        (p1,q1) ->  if x then p1 else q1
```

In effect, this composition of simple transformations implements the more complex rule "if any variable in a pattern binding is sure to be evaluated, then the pattern binding can be taken apart using `case` rather than the full lazy-pattern-binding mechanism". Using a `case` expression allocates no thunks, and (in our implementation) does not even build the pair `(p1,q1)` in the heap — the two components are simply returned in registers.

### 4.4 Floating `case` out of `let`

There is yet another way in which we can exploit strictness information to reduce the construction of thunks and remove repeated evaluation. Consider the expression

```
let x = case y of
        <pat> -> <rhs>
in <body>
```

Furthermore, suppose that `x` is used strictly by `<body>`; that is, if `x` is bottom then so is `<body>`. Then `x` is sure to be evaluated, and hence so is `y`. This means that it is safe to transform to:

```
case y of
<pat> -> let x = <rhs>
          in <body>
```

Notice that the transformation works *regardless of the type of* `x` — for example, it does not have to be a single-constructor type, or even a data type. (If `x` were a single-constructor type, then the `let` to `case` transformation would be used.) Floating the `case` out of the `let` is a Good Thing to do for two reasons:

- It widens the scope of the `case`. For example, it may be that `y` is evaluated somewhere else in `<body>` (that is, there is a `case y of ...` in `<body>`). Now that `<body>` appears inside the `case y`, the second `case y` can be eliminated, saving an evaluation.
- It may be that `<rhs>` is a simple variable, in which case the `let` expression can now be eliminated entirely. Alternatively, `<rhs>` might be a constructor application, so now the `let` expression would allocate an immutable constructor, rather than a thunk which later has to be updated.

### 4.5 Absence

Consider this definition:

```
f t = case t of (a,b,c,d) -> b
```

The strictness of `f` is U(LLSL), which leads to the following worker/wrapper split (after some simplification):

```
f t = case t of (a,b,c,d) -> fw a b c d
fw a b c d = b
```

This is a bit silly, because the components of the tuple are passed to `fw` even though only one is used. It would be better if we could figure out when an argument is guaranteed *not* to be evaluated, so that we needn't pass it to the worker at all! Using "A" (absent) to indicate this, we would like `f`'s strictness to be U(AASA). In the absence of product types absence is rare, because programmers seldom write functions which ignore one of their arguments, but once products are added absence becomes quite common (notably in the form of selector functions).

The property of "guaranteed not to be evaluated" (that is, absence) is dual to that of "guaranteed to be evaluated" (that is, strictness), and a second abstract interpretation is needed to gather absence information. Why is a full abstract interpretation needed? Can we not simply use a syntactic criterion? Consider:

```
g x = fst (fst x)
```

We would like to get the strictness property `U(U(SA)A)` for `g`, even though the absence of the arguments is not syntactically apparent.

We have implemented absence analysis, using the same infrastructure as for the strictness analysis (abstract interpreter, fixpointing, etc), but the details are beyond the scope of this paper.

### 4.6 Modules

It is absolutely essential to convey strictness information across module boundaries. Like most compilers, we do this by adding strictness annotations to the module's interface file, which is imported by other modules using this one's resources. For example, the following might appear in an interface file:

```
f :: Int -> [Int] -> Bool -> Bool
{-# GHC_PRAGMA ... _S_ "U(LL)SL" ... #-}
```

In order to keep the strictness annotations small, they only encode widened functions, using the demand notation described in Section 3.2.1. The strictness of non-recursive functions is expressed in this way, as well as recursive ones; some details of their full abstract value is therefore lost across module boundaries.

# 5 Results

We exercised our strictness analyser on each of 17 programs, from the "Real" part of the (still-unreleased) `nofib` suite of Haskell programs. They cover a broad spread of application domains, including theorem proving, RSA encoding, geometric modelling, type inference, and data compression. They vary from 70 to 11,000 lines of Haskell source, with 500 lines being typical.

Each program was compiled with and without strictness analysis enabled, using a development version of GHC from November, 1993. All the other transformations used were identical, and in each case, the programs were linked with a standard Prelude which was compiled in the same way as they were.

All programs were compiled with "ticky-ticky" profiling: when run, they collect many dynamic counts, e.g., the number of three-word closures allocated, the number of updates, etc.

All the figures presented in the next section refer to *dynamic* counts. For example, the number of function calls which managed to call the worker of a strict function is the dynamic count, not the static number of such calls in the program text. The dynamic counts are what affects run-time.

There is one main caveat: too many programs have input data which leads to a runtime which is too small to measure reliably. This is partly a problem of success (the compiler has improved) and partly that they are run on a fast machine.

## 5.1 Run times

In a sense the runtime is the result that "really matters", although it doesn't give much insight in itself. Table 1 shows the change in runtime for each of the programs without strictness analysis ("None" column) and the percentage change when strictness analysis is enabled ("Strictly" column). "R" is real, wall-clock time, presented for completeness; "U+S" is user+system time, which is more informative. A dash (—) means "no discernible change". Each was run on a Sun SparcStation 10 with 48Mbytes of memory.

The performance gains are modest but consistent, ranging from near-zero to an improvement of 30% or more. `maillist`, whose execution time actually increases, is an unusual program, being totally dominated by input/output. We believe it is amplifying a small infelicity in the transformation system.

| | None | | Strictly | |
|---|---|---|---|---|
| | R | U+S | R | U+S |
| anna | 8.4 | 8.2 | –3.6% | –3.8% |
| veritas | 4.4 | 0.6 | +18.2% | –16.7% |
| gg | 7.3 | 4.3 | –2.7% | –7.0% |
| hidden | 412.0 | 395.1 | –31.3% | –32.1% |
| maillist | 94.2 | 15.2 | +7.3% | +5.9% |
| bspt | 6.1 | 3.6 | –13.1% | –19.4% |
| compress | 83.1 | 81.1 | –21.7% | –21.7% |
| infer | 18.8 | 16.7 | –9.6% | –9.6% |
| lift | 2.2 | 0.3 | +13.6% | — |
| parser | 15.2 | 12.0 | –13.8% | –14.2% |
| prolog | 2.3 | 0.7 | –8.7% | — |
| reptile | 5.9 | 3.7 | +23.7% | –2.7% |
| rsa | 27.3 | 25.6 | –9.2% | –9.0% |
| fluid | 7.5 | 3.5 | –1.3% | –22.9% |
| gamteb | 64.2 | 61.1 | –33.0% | –33.7% |
| pic | 13.0 | 9.6 | –37.7% | –44.8% |
| fulsom | 129.7 | 126.0 | –10.3% | –10.6% |

Table 1: Run times (in seconds [real; user+system])

### 5.2 Allocations saved

In Section 2 we identified the main saving from strictness analysis as the reduction in the number of thunks allocated. Table 2 quantifies this by counting how many thunks are allocated in a run of each program under each build. Just to reassure ourselves that we aren't making savings in one place only to incur costs in another, the "Others" column counts all allocation *other* than thunks; mostly allocation of head normal forms, such as data constructors and partial function applications. As before, the "Strictly" columns are given as percentage changes from the corresponding "None" column.

The number of thunks allocated always decreases, sometimes dramatically so (eg `hidden`). There are, unsurprisingly, more non-thunks allocated, but the total allocation is usually reduced. We would expect the total allocation to *always* be reduced; the cases where it is not (`maillist`, `bspt`, `pic`) show a rather large increase in non-thunk allocation). This deserves further investigation.

### 5.3 Updates saved

As well as reducing the number of thunks, we hope also to reduce the number of updates. Indeed, since every one of the thunks we do not allocate is one which would be entered, one might expect approximately one update to be saved for every thunk saved. This is confirmed by comparing the first two columns in Table 3. The figures are not exactly the same because other transformations are enabled or disabled by the effects of strictness analysis.

| | None | | | Strictly | | |
|---|---|---|---|---|---|---|
| | Thunks | Others | Total | Thunks | Others | Total |
| anna | 622,624 | 175,511 | 798,135 | –15.9% | +25.6% | –6.8% |
| veritas | 29,101 | 3,635 | 32,736 | –4.8% | +4.1% | –3.8% |
| gg | 335,104 | 151,972 | 487,076 | –14.9% | +18.8% | –4.4% |
| hidden | 61,178,224 | 6,449,232 | 67,627,456 | –46.4% | +13.1% | –40.7% |
| maillist | 144,555 | 313,380 | 457,935 | –19.0% | +69.9% | +41.8% |
| bspt | 390,280 | 48,979 | 439,259 | –26.9% | +246.3% | +3.6% |
| compress | 1,623,431 | 8,209,918 | 9,833,349 | –5.8% | — | –1.0% |
| infer | 619,273 | 274,155 | 893,428 | –5.7% | +7.9% | –1.5% |
| lift | 20,099 | 9,646 | 29,745 | –9.1% | +15.1% | –1.2% |
| parser | 1,276,925 | 614,789 | 1,891,714 | –34.8% | +26.4% | –14.9% |
| prolog | 44,189 | 22,426 | 66,615 | –30.4% | +43.0% | –5.7% |
| reptile | 474,845 | 82,681 | 557,526 | –22.2% | +29.7% | –14.5% |
| rsa | 553,934 | 414,186 | 968,120 | –91.6% | –47.7% | –72.8% |
| fluid | 268,514 | 83,288 | 351,802 | –30.3% | +58.6% | –9.2% |
| gamteb | 4,333,275 | 2,015,412 | 6,348,687 | –49.8% | +74.1% | –10.4% |
| pic | 599,024 | 369,784 | 968,808 | –72.9% | +139.6% | +8.2% |
| fulsom | 16,305,421 | 7,937,428 | 24,242,849 | –6.0% | –28.1% | –13.2% |

Table 2: Thunks (and other objects) allocated

| | Thunks | Updates | None | | Strictly | |
|---|---|---|---|---|---|---|
| | eliminated by strictness analysis | | without update analysis | with | without update analysis | with |
| anna | 99,010 | 90,096 | 535,148 | –11.5% | 445,052 | –5.5% |
| veritas | 1,397 | 1,366 | 28,466 | –3.2% | 27,100 | –0.4% |
| gg | 49,858 | 49,566 | 330,051 | –3.8% | 280,485 | –0.5% |
| hidden | 28,362,677 | 14,034,948 | 39,774,327 | –6.9% | 25,739,379 | –5.2% |
| maillist | 27,509 | 27,509 | 104,139 | –8.7% | 76,630 | — |
| bspt | 104,889 | 102,461 | 382,558 | –9.4% | 280,097 | –1.5% |
| compress | 94,872 | 94,872 | 1,623,318 | — | 1,528,446 | — |
| infer | 35,389 | 35,370 | 443,570 | –9.0% | 408,200 | –1.5% |
| lift | 1,821 | 1,871 | 19,228 | –7.6% | 17,357 | –1.8% |
| parser | 443,965 | 290,793 | 720,170 | –1.5% | 429,377 | –1.9% |
| prolog | 13,448 | 13,423 | 36,908 | –32.4% | 23,485 | –1.3% |
| reptile | 105,313 | 105,900 | 474,142 | –16.9% | 368,242 | –2.1% |
| rsa | 507,463 | 455,299 | 501,469 | –38.5% | 46,170 | — |
| fluid | 81,310 | 78,438 | 238,861 | –6.2% | 160,423 | –0.7% |
| gamteb | 2,156,738 | 2,053,226 | 4,175,334 | –2.3% | 2,122,108 | –0.9% |
| pic | 436,633 | 427,670 | 578,972 | –0.2% | 151,302 | — |
| fulsom | 970,177 | 859,845 | 12,191,166 | –7.1% | 11,331,321 | –6.6% |

Table 3: Updates

| | None | | | Strictly | | |
|---|---|---|---|---|---|---|
| | data-vals | thunks | others | data-vals | thunks | others |
| anna | 1,875,349 | 522k | 2,187k | –5.1% | –16.9% | –20.8% |
| veritas | 11,293 | 27k | 37k | –6.2% | –3.7% | –8.1% |
| gg | 657,597 | 322k | 1,144k | –33.3% | –15.2% | –23.8% |
| hidden | 79,661,596 | 38,842k | 104,354k | –21.3% | –35.3% | –37.1% |
| maillist | 1,072,984 | 101k | 1,077k | –26.6% | –26.7% | –41.3% |
| bspt | 475,832 | 373k | 799k | –20.5% | –26.8% | –32.5% |
| compress | 36,689,810 | 1,585k | 24,311k | –5.7% | –5.9% | –64.3% |
| infer | 7,177,846 | 433k | 6,339k | — | –8.1% | –38.1% |
| lift | 22,368 | 18k | 37k | –2.7% | –11.1% | –10.8% |
| parser | 2,041,658 | 703k | 3,347k | –4.0% | –40.4% | –14.6% |
| prolog | 98,091 | 36k | 157k | –6.6% | –38.9% | –29.9% |
| reptile | 449,386 | 463k | 929k | –50.3% | –22.5% | –33.2% |
| rsa | 1,032,357 | 489k | 1,678k | –63.6% | –90.8% | –40.9% |
| fluid | 589,827 | 233k | 864k | –42.9% | –33.0% | –49.8% |
| gamteb | 11,096,188 | 4,077k | 15,683k | –59.1% | –49.2% | –54.6% |
| pic | 2,281,123 | 565k | 2,915k | –50.1% | –74.0% | –65.4% |
| fulsom | 13,816,368 | 11,905k | 26,017k | –24.6% | –7.1% | –32.3% |

Table 4: Number of enters

The picture is complicated slightly by *update analysis*, which tries to infer when a thunk can only be entered at most once, and hence does not need to be updated (Marlow [1993]). An interesting question is: *does strictness analysis eliminate exactly those thunks which update analysis identifies as single-entry, or does update analysis find some more beside*? The "with" columns on the right side of Table 3 quantify the answer. It says how many updates happened even with the update analyser. On some numerically-intensive programs (`rsa`, `compress`) the update analyser can't find much to do after the strictness analyser has done its stuff. However, there are often handy winnings still to be had (eg `anna`, `hidden`).

### 5.4 Evaluations saved

A side benefit of strictness analysis is that sometimes a thunk may be evaluated just once instead of twice (cf Section 2.1). The second enter will encounter an evaluated data value, so with strictness analysis one would expect the number of data-value enters to drop. And so it does (Table 4). The reduction in the number of data-value enters is often modest (`veritas`, `infer`) but sometimes very substantial (`reptile`, `gamteb`) — the numerical programs seem to be the ones which work well, unsurprisingly.

The number of thunks entered also drops, which is unsurprising since fewer thunks are allocated.

### 5.5 Function calls

How many function calls are able to exploit strictness at all? Table 5 gives the answers. It splits all function calls into three groups: the "non-wrapperised" ones, which have no

| | Non-wrapperised | Workers | Wrappers |
|---|---|---|---|
| anna | 1,309,080 | 76,303 | 7,964 |
| veritas | 31,292 | 1,394 | 128 |
| gg | 472,077 | 158,189 | 67,687 |
| hidden | 47,582,515 | 11,840,378 | 3,796,296 |
| maillist | 452,020 | 150,096 | 12,129 |
| bspt | 322,805 | 143,250 | 19,290 |
| compress | 8,535,426 | 126,507 | 8 |
| infer | 2,963,582 | 12,763 | 2,630 |
| lift | 23,875 | 3,455 | 1,902 |
| parser | 2,398,474 | 100,531 | 28,363 |
| prolog | 98,359 | 6,253 | 191 |
| reptile | 518,598 | 96,612 | 12,454 |
| rsa | 67,227 | 729,738 | 18,882 |
| fluid | 279,668 | 83,186 | 22,691 |
| gamteb | 2,350,999 | 2,977,282 | 805,196 |
| pic | 267,546 | 442,348 | 167,144 |
| fulsom | 11,054,384 | 2,308,330 | 318,209 |

Table 5: Number of function calls to... ("Strictly" build)

exploitable strict arguments; the "workers" which are the function calls to a strict function where the worker was called directly; and the "wrappers" which are the calls to strict functions where the wrapper was called. Why are there any calls in the last category? Because data abstraction means that (currently) we may not be able to unpack the argument(s) at the call site even though they are known to be strict.

### 5.6 Argument distribution

In the discussion of Section 2 we distinguished between various argument types: `Int`, single-constructor, tuples, multi-constructor, and so on. Table 6 tells the distribution of these argument types over all calls. The proportions are given as percentages of all strict argument positions, so that non-strict arguments don't appear at all.

The first group of columns are all single-constructor types that could "unpack" effectively: c=`Char`, i=`Int`, j=`Integer`, f=`Float`, d=`Double`, t=tuple, s=other single-constructor type. Column U gives the percentage of single-constructor arguments that we *failed* to unpack, because of data abstraction.

The most unexpected measurement is the prevalence of the "s" column, indicating that generalising the unboxery to types other than numeric ones and tuples is quite worth while.

Lastly, Table 7 addresses the question of the number of arguments which are passed to workers. In principle, a worker might take very many more arguments than its wrapper, since several arguments may be unpacked. In extreme cases this might be quite counter-productive. Table 7 is reassuring: a worker seldom takes many more arguments than its wrapper.

| | c | i | j | f | d | t | s | U |
|---|---|---|---|---|---|---|---|---|
| anna | 17.39 | 32.73 | | | | 6.38 | 28.85 | 9.35 |
| veritas | 1.39 | 7.37 | | | | 73.35 | 4.03 | 8.38 |
| gg | 6.60 | 16.62 | 38.37 | 0.01 | 0.18 | 7.54 | 0.27 | 29.99 |
| hidden | 0.33 | 1.27 | 1.98 | 0.01 | 0.00 | 1.44 | 71.68 | 23.29 |
| maillist | | 41.59 | | | | 41.08 | 6.54 | 6.54 |
| bspt | | 36.61 | | | | 5.94 | 44.31 | 11.69 |
| compress | | 99.98 | | | | 0.01 | 0.00 | 0.00 |
| infer | 24.13 | 43.34 | | | | 0.18 | 17.29 | 15.00 |
| lift | 21.47 | 19.05 | 2.14 | | | 9.37 | 13.11 | 34.77 |
| parser | 60.38 | 7.86 | | | | 4.99 | 0.20 | 20.51 |
| prolog | 76.49 | 14.75 | | | | 1.16 | 0.54 | 2.96 |
| reptile | 1.46 | 90.11 | | | | 0.03 | 0.00 | 8.33 |
| rsa | | 0.22 | 96.90 | | | 0.47 | 0.00 | 2.40 |
| fluid | 8.61 | 33.01 | 25.06 | 5.07 | 0.14 | 4.20 | 0.01 | 18.85 |
| gamteb | 0.00 | 35.67 | 33.04 | | 0.87 | 8.39 | 0.24 | 21.79 |
| pic | 0.00 | 40.70 | 1.69 | | 1.65 | 27.67 | 0.00 | 28.29 |
| fulsom | 0.00 | 28.06 | 1.34 | | 28.03 | 24.86 | 0.37 | 14.77 |

Table 6: Wrappers' strict arguments were...

| | −3 | −2 | −1 | 0 | +1 | +2 | +3 | +4 | +5 |
|---|---|---|---|---|---|---|---|---|---|
| anna | | 0.00 | 0.30 | 86.78 | 8.29 | 4.32 | 0.00 | 0.31 | |
| veritas | 0.14 | 1.43 | 2.08 | 25.90 | 42.97 | 25.61 | 0.36 | | 1.51 |
| gg | | | | 41.72 | 0.31 | 19.72 | 0.26 | 37.86 | 0.14 |
| hidden | | | 0.02 | 93.97 | 1.42 | 2.35 | 0.02 | 2.22 | 0.00 |
| maillist | | | | 7.18 | 88.78 | | 4.03 | | |
| bspt | | | | 48.60 | 8.91 | 39.06 | 1.14 | | 2.29 |
| compress | | | | 99.99 | 0.01 | | | | |
| infer | | | 0.01 | 99.74 | 0.23 | 0.02 | | | |
| lift | | | | 81.22 | 16.12 | 0.61 | | 2.00 | 0.06 |
| parser | | | 0.10 | 92.77 | 0.04 | 0.28 | 6.82 | | |
| prolog | | | | 98.74 | 1.23 | | 0.03 | | |
| reptile | | | 0.10 | 99.85 | 0.02 | 0.03 | | | |
| rsa | | | | 1.25 | 0.00 | 22.86 | | 75.89 | 0.00 |
| fluid | | | 1.15 | 63.88 | 7.01 | 3.92 | 0.40 | 23.62 | 0.01 |
| gamteb | | | 0.00 | 46.72 | 14.24 | 4.94 | 0.17 | 33.40 | 0.52 |
| pic | | | 0.00 | 26.31 | 70.95 | 1.05 | 0.01 | 1.67 | 0.00 |
| fulsom | | 0.00 | 0.00 | 55.88 | 0.00 | 0.98 | 41.69 | 1.45 | 0.00 |

Table 7: Worker functions' # of arguments vs. their wrappers' # (percentages)

# 6 Conclusion

Our main conclusion is this: strictness analysis on large, realistic programs leads to solid but modest improvements in execution speed. The best speedup we observed was about 30% (ie the runtime dropped to about 70% of its previous value), but 10–20% is more typical. This contrasts with other papers which describe dramatic speedups, but these are usually measured on small programs which spend a lot of time in an optimisable loop.

A second general conclusion is that program behaviours really do vary widely, even in large programs — see, for example, the distribution of strict argument types in Table 6. There is no "silver bullet" — a good compiler has to be more like a shotgun, with many transformations targetted at many situations.

A third conclusion is that making detailed quantitative measurements of the internal behaviour of programs, as we have done, often shows up compiler "performance bugs"; that is, places where the compiler generates significantly (but not drastically) less good code than it could. These can lie (and in our experience have lain) undiscovered for a very long time, since they do not cause programs to fail, until uncovered by making measurements.

There is plenty of scope for refining our measurements. In particular, each outcome (eg number of thunks allocated) is the effect of a combination of causes (eg `let`-to-`case`, unboxing, etc); it would be nice to isolate the effect of each of these causes.


# Acknowledgements

Many thanks to John Launchbury for keeping us honest with the maths; to André Santos whose transformation system exploits and propagates the results of strictness analysis; and to Andy Gill, who wrote the first version of this strictness analyser.


# Appendix: proofs

Our theorem was:

**Theorem 1:** If $w$ is a widening operator, then $\mathit{fix}\ F \sqsubseteq \mathit{fix}\ wF$

That is, the fixpoint of $wF$ is a safe approximation (from the point of view of strictness analysis) to the fixpoint of $F$. The proof is in two steps.

$$\begin{array}{lcll} F(\mathit{fix}\ wF) & \sqsubseteq & wF(\mathit{fix}\ wF) & \text{Since } w \text{ is a widening operator} \\ & = & \mathit{fix}\ wF & \text{By definition of } \mathit{fix} \end{array}$$

In the jargon, $\mathit{fix}\ wF$ is a *post-fixpoint* of $F$. In general, $z$ is a post-fixpoint of $F$ if $F(z) \sqsubseteq z$. The following theorem holds for post-fixpoints:

**Theorem 2:** a post-fixpoint of $F$ is greater than $F^n(\bot)$ for all $n$. More precisely, if $z$ is a post-fixpoint of $F$, then $F^n(\bot) \sqsubseteq z$.

The proof is a simple induction on $n$. Certainly $\bot \sqsubseteq z$, which is the base case. Assuming $F^n(\bot) \sqsubseteq z$, applying $F$ to both sides gives $F^{n+1}(\bot) \sqsubseteq F(z)$; but since $z$ is a post-fixpoint of $F$, $F(z) \sqsubseteq z$, and hence $F^{n+1}(\bot) \sqsubseteq z$.□

**Corollary 3:** if $z$ is a post-fixpoint of $F$, then *fix* $F \sqsubseteq z$. This follows immediately from Theorem 2.□

**Corollary 4:** since $F(\textit{fix } wF) \sqsubseteq \textit{fix } wF$, once we have found *fix* $wF$ we can get a better (or at least no worse) approximation to *fix* $F$ by computing $F^n(\textit{fix } wF)$, for some arbitrarily chosen $n$. No tests for equality are required here, because $\textit{fix } F \sqsubseteq F^n(\textit{fix } wF)$ for any $n$.

## References


GL Burn [April 1990], "The evaluation transformer model of reduction and its correctness," in *TAPSOFT 91, Brighton*.

GL Burn, CL Hankin & S Abramsky [Nov 1986], "Strictness analysis for higher order functions," *Science of Computer Programming* 7, 249–278.

S Finne & G Burn [June 1993], "Assessing the evaluation transformer model of reduction on the spineless G-machine," in *Proc Functional Programming Languages and Computer Architecture, Copenhagen*, ACM, 331–340.

CV Hall [1993], "A framework for optimising abstract data types," in *Functional Programming, Glasgow 1993*, K Hammond & JT O'Donnell, eds., Workshops in Computing, Springer Verlag.

DB Howe & GL Burn [1993], "Using strictness in the STG machine," in *Functional Programming, Glasgow 1993*, K Hammond & JT O'Donnell, eds., Workshops in Computing, Springer Verlag.

RJM Hughes [Sept 1992], "A loop-detecting interpreter for lazy higher-order programs," Department of Computer Science, Chalmers University.

S Marlow [1993], "Update avoidance analysis using abstract interpretation," in *Functional Programming, Glasgow 1993*, K Hammond & JT O'Donnell, eds., Workshops in Computing, Springer Verlag.

SL Peyton Jones & CD Clack [1986], "Finding fixpoints in abstract interpretation," in *Abstract Interpretation of Declarative Languages*, C Hankin & S Abramsky, eds., Ellis Horwood, Chichester, 246–265.

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.

S Smetsers, E Nocker, J van Groningen & R Plasmeijer [Sept 1991], "Generating efficient code for lazy functional languages," in *Functional Programming Languages and Computer Architecture, Boston*, Hughes, ed., LNCS 523, Springer Verlag.

PL Wadler & John Hughes [Sept 1987], "Projections for strictness analysis," in *Functional Programming Languages and Computer Architecture*, G Kahn, ed., Springer Verlag LNCS 274.