Secrets of the Glasgow Haskell Compiler inliner

Simon Peyton Jones
Microsoft Research Ltd, Cambridge
simonpj@microsoft.com

Simon Marlow
Microsoft Research Ltd, Cambridge
simonmar@microsoft.com

September 1, 1999

Abstract

Higher-order languages, such as Haskell, encourage the programmer to build abstractions by composing functions. A good compiler must inline many of these calls to recover an efficiently executable program.

In principle, inlining is dead simple: just replace the call of a function by an instance of its body. But any compiler-writer will tell you that inlining is a black art, full of delicate compromises that work together to give good performance without unnecessary code bloat.

The purpose of this paper is, therefore, to articulate the key lessons we learned from a full-scale "production" inliner, the one used in the Glasgow Haskell compiler. We focus mainly on the algorithmic aspects, but we also provide some indicative measurements to substantiate the importance of various aspects of the inliner.

1 Introduction

One of the trickiest aspects of a compiler for a functional language is the handling of inlining. In a functional-language compiler, inlining subsumes several other optimisations that are traditionally treated separately, such as copy propagation and jump elimination. As a result, effective inlining is particularly crucial in getting good performance.

The Glasgow Haskell Compiler (GHC) is an optimising compiler for Haskell that has evolved over a period of about ten years. We have repeatedly been through a cycle of looking at the code it produces, identifying what could be improved, and going back to the compiler to make it produce better code. It is our experience that the inliner is a lead player in many of these improvements. No other single aspect of the compiler has received so much attention.

The purpose of this paper is to report on several algorithmic aspects of GHC's inliner, focusing on aspects that were not obvious to us — that is to say, aspects that we got wrong to begin with. Most papers about inlining focus on how to choose whether or not to inline a function called from many places. This is indeed an important question, but we have found that we had to deal with quite a few other less obvious, but equally interesting, issues. Specifically, we describe the following:

- A major issue for any compiler, especially for one that inlines heavily, is *name capture*. Our initial brute-force solution involved inconvenient plumbing, and we have now evolved a simple and effective alternative (Section 3).
- At first we were very conservative about inlining *recursive* definitions; that is, we did not inline them at all. But we found that this strategy occasionally behaved very badly. After a series of failed hacks we developed a simple, obviously-correct algorithm that does the job beautifully (Section 4).
- Because the compiler does so much inlining, it is important to get as much as possible done in each pass over the program. Yet one must steer a careful path between doing too *little* work in each pass, requiring extra passes, and doing too *much* work, leading to an exponential-cost algorithm. GHC now identifies *three* distinct moments at which an inlining decision may be taken for a particular definition. We explain why in Section 6.
- When inlining an expression it is important to retain the expression's *lexical* environment, which gives the bindings of its free variables. But at the inline site, the compiler might know more about the *dynamic* state of some of those free variables; most notably, a free variable might be known to be evaluated at the inline site, but not at its original occurrence. Some key transformations make use of this extra information, and lacking it will cause an extra pass over the code. We describe how to exploit our name-capture solution to support accurate tracking of both lexical and dynamic environments (Section 7).

None of the algorithms we describe is individually very surprising. Perhaps because of this, the literature on the subject is very sparse, and we are not aware of published descriptions of any of our algorithms. Our contribution is to abstract some of what we have learned, in the hope that we may help others avoid the mistakes that we made.

For the sake of concreteness we focus throughout on GHC, but we stress that the lessons we learned are applicable to any compiler for a functional language, and indeed perhaps to compilers for other languages too.

2 Preliminaries

We will assume the use of a pure, non-strict, strongly-typed intermediate language, called the *GHC Core language*. GHC is itself written in Haskell, so we define the Core language by giving its data type definition in Haskell:

```
type Program = [Bind]

data Bind = NonRec Var Expr
          | Rec [(Var, Expr)]

data Expr = Var   Var
          | App   Expr Expr
          | Lam   Var Expr
          | Let   Bind Expr
          | Const Const [Expr]
          | Case  Expr Var [Alt]
          | Note  Note Expr

type Alt      -- Case alternative
   = (Const, [Var], Expr)

data Const    -- Constant
   = Literal  Literal
   | DataCon  DataCon
   | PrimOp   PrimOp
   | DEFAULT
```

The Core language consists of the lambda calculus augmented with let-expressions (both non-recursive and recursive), case expressions, data constructors, literals, and primitive operations. In presenting examples we will use an informal, albeit hopefully clear, concrete syntax. We will feel free to use infix operators, and to write several bindings in a single non-recursive let-expression as shorthand for a sequence of let-expressions.

A program (`Program`) is simply a sequence of bindings, in dependency order. Each binding (`Bind`) can be recursive or non-recursive, and the right hand side of each binding is an expression (`Expr`). The constructors for variables (`Var`), application (`App`), lambda abstraction (`Lam`), and let-expressions (`Let`) should be self-explanatory. A constant application (`Const`) is used for literals, data constructor applications, and applications of primitive operators; the number of arguments must match the arity of the constant, and and the constant cannot be `DEFAULT`. (Likewise, the number of bound variables in a case alternative (`Alt`) always matches the arity of the constant; and the latter cannot be a `PrimOp`.) The `Note` form of `Expr` allows annotations to be attached to the tree. The only impact on the inliner is discussed in Section 7.6.

Case expressions (`Case`) should be self-explanatory, except for the `Var` argument to `Case`. Consider the following Core expression,

```
case (reverse xs) of ys {
  (a:as) -> ys
  []     -> error "urk"
}
```

The unusual part of this construct is the binding occurrence of "`ys`", immediately after the "`of`". The semantics is that `ys` is bound to the result of evaluating the scrutinee, `reverse xs` in this case, which makes it possible to refer to this value in the alternatives. This detail has no impact on the rest of this paper — indeed, we omit the extra binder in our examples — but we have found that it makes several transformations more simple and uniform, so we include it here for the sake of completeness.

GHC's actual intermediate language is very slightly more complicated than that given here. It is an explicitly-typed language based on System F_ω, and supports polymorphism through explicit type abstraction and application. It turns out that doing so adds only one new constructor to the `Expr` type, and adds nothing to the substance of this paper, so we do not mention it further. The main point is that this paper omits no aspect essential to a full-scale implementation of Haskell.

2.1 What is inlining?

Given a definition `x = E`, one can *inline* `x` at a particular occurrence by replacing the occurrence by `E`. (We use upper case letters, such as "E", to stand for arbitrary expressions, and "==>" to indicate a program transformation.) For example:

```
  let { f = \x -> x*3 } in f (a + b) - c
==>
  (a+b)*3 - c
```

We have found it useful to identify three distinct transformations that collectively implement what we informally describe as "inlining":

- *Inlining itself* replaces an occurrence of a `let`-bound variable by (a copy of) the right-hand side of its definition. Inlining `f` in the example above goes like this:

  ```
    let { f = \x -> x*3 } in f (a + b) - c
  ==> [inline f]
    let { f = \x -> x*3 } in (\x -> x*3) (a + b) - c
  ```

  Notice that not all the occurrences of `f` need be inlined, and hence that the original definition of `f` must, in general, be retained.

- *Dead code elimination* discards bindings that are no longer used; this usually occurs when all occurrences of a variable have been inlined. Continuing our example gives:

  ```
    let { f = \x -> x*3 } in (\x -> x*3) (a + b) - c
  ==> [dead f]
    (\x -> x*3) (a + b) - c
  ```

- *β-reduction* simply rewrites a lambda application `(\x->E) A` to `let {x = A} in E`. Applying β-reduction to our running example gives:

  ```
    (\x -> x*3) (a + b) - c
  ==> [beta]
    (let { x = a+b } in x*3) - c
  ```

The first of these is the tricky one; the latter two are easy. In particular, beta reduction simply creates a `let` binding.

In a lazy, purely functional language, inlining and dead-code elimination are both unconditionally valid, or meaning-preserving. (Neither is valid, in general, in a language permitting side effects, such as Standard ML or Scheme.) In particular, notice that inlining is valid, regardless of

- the number of occurrences of `x`,
- whether or not the binding for `x` is recursive,
- whether or not `E` has free variables (that is, inlining of nested definitions is perfectly fine), and
- the syntactic form of `E` (notably, whether or not it is a lambda abstraction).

Concerning the last of these items, notice that we (unconventionally) use the term "inline" equally for both functions and non-functions. Continuing the example, `x` can now be inlined, and then dropped as dead code, thus:

```
  (let { x = a+b } in x*3) - c
==> [inline x]
  (let { x = a+b } in (a+b)*3) - c
==> [dead x]
  (a+b)*3 - c
```

In this case, `x` is used exactly once, but we sometimes also inline non-functions that are used several times. Consider:

```
let x = (a,b)
in
...x...(case x of { (p,q) -> p+1 })...
```

By inlining `x` we can then eliminate the `case` to give

```
let x = (a,b)
in
...x...(a+1)...
```

In a similar way (when given bindings such as `x=y`), inlining subsumes copy propagation.

2.2 Factors affecting inlining

To say that inlining is *valid* does not mean that it is *desirable*. Inlining might increase code size, or duplicate work, so we need be careful about when to do it. There are three distinct factors to consider:

- *Does any code get duplicated,* and if so, how much? For example, consider

  ```
  let f = \v -> ...big... in (f 3, f 4)
  ```

  where "`...big...`" is a large expression. Then inlining `f` would not duplicate any work (`f` will still be called twice), but it will duplicate the code for `f`'s body. Bloated programs are bad (increased compilation time, lower cache hit rates), but inlining can often *reduce* code size by exposing new opportunities for transformations. GHC uses a number of heuristics to determine whether an expression is small enough to duplicate.

- *Does any work get duplicated,* and if so, how much? For example, consider

  ```
  let x = foo 1000 in x+x
  ```

  where `foo` is expensive to compute. Inlining `x` would result in two calls to `foo` instead of one.

  Work can be duplicated even if `x` only appears once:

  ```
  let x = foo 1000
      f = \y -> x * y
  in ...(f 3)..(f 4)...
  ```

  If we inline `x` at its (single) occurrence site, `foo` will be called every time `f` is. The general rule is that we must be careful when inlining inside a lambda.

  It is not hard to come up with examples where a single inlining that duplicates work gives rise to an arbitrarily large increase in run time. GHC is therefore very conservative about work duplication. In general, GHC never duplicates work unless it is sure that the duplication is a small, bounded amount.

- *Are any transformations exposed by inlining?* For example, consider the bindings:

  ```
  f = \x -> E
  g = \ys -> map f ys
  ```

  Suppose we were to inline `f` inside `g`, thus:

  ```
  g = \ys -> map (\x -> E) ys
  ```

  No *code* is duplicated by doing so, but a small bounded amount of *work* is duplicated, because the closure for `(\x -> E)` would have to be allocated each time `g` was called. It is often worth putting up with this work duplication, because inlining `f` exposes new transformation opportunities at the inlining site. But in this case, nothing at all would be gained by inlining `f`, because `f` is not applied to anything.

These considerations imply that inlining is not an optimisation "by itself". The *direct* effects of careful inlining are small: it may duplicate code or a constant amount of work, and usually saves a call or jump (albeit not invariably — see the example in the last bullet above). It is the *indirect* effects that we are really after: the main reason for inlining is that it often exposes new transformations, by bringing together two code fragments that were previously separate. Thus, in general, inlining decisions must be influenced by context.

2.3 Work duplication

If `x` is inlined in more than one place, or inlined inside a lambda, we have to worry about work duplication. When will such work duplication be bounded? Answer: at least in the cases when `x`'s right hand side is:

- A variable.
- A constructor application.
- A lambda abstraction.

- An expression that is sure to diverge.

Constructor applications require careful treatment. Consider:

```
x = (f y, g y)
h = \z -> case x of
              (a,b) -> ...
```

It would plainly be a disaster, in general, to inline `x` inside the body of `h`, since that would duplicate the calls to `f` and `g`. Yet we want to inline `x` so that it can cancel with the `case`. GHC therefore maintains the invariant that every constructor application has only arguments that can be duplicated with no cost: variables, literals, and type applications. We call such arguments *trivial* expressions, so the invariant is called the *trivial-constructor-argument invariant*. Once established, this invariant is easy to maintain (see Section 7.2).

The last case, that of divergent computations, is more surprising, but it is useful in practice. Consider:

```
sump = \xs ->
       let
         fail = error ("sump" ++ show xs)
       in let rec
         go = \xs ->
              case xs of
                []     -> 0
                (x:xs) -> if x<0 then fail
                          else x + go xs
       in
       go xs
```

Here `error` is the standard Haskell function that prints an error message and brings execution to a halt. Semantically, its value is just $\perp$, the divergent value. In this example, `sump` adds up the elements of a list, but reports an error if any element is negative. As it stands, a closure for `fail` will be allocated every time `sump` is called. It is perfectly OK to inline `fail`, because if `fail` is ever called, execution is going to halt anyway, so there is no work-duplication issue. If we do that, no closure is allocated; instead, `error` is called directly if an element turns out to be less than zero.

GHC has a predicate `whnfOrBot` that identifies expressions that are in WHNF or are certainly divergent:

```
whnfOrBot :: Expr -> Bool
```

One could easily imagine extending `whnfOrBot` to cover cases where a small amount of work other than allocation is duplicated, such as a few machine instructions.

3 Name capture

It is well known that any transformation-based compiler must be concerned about *name capture* [Bar85]. Consider, for example:

```
let x = a+b in
   let a = 7 in
      x+a
```

It is obviously quite wrong to inline `x` to give:

```
let a = 7 in
     (a+b) + a
```

because the `a` that was free in `x`'s right hand side has been captured by the `let` binding for `a`.

3.1 The sledge hammer

Earlier versions of GHC used a sledge-hammer approach to avoid the name-capture problem: during inlining, GHC would simply rename, or *clone*, every single bound variable, to give:

```
let s796 = 7
in (a+b) + s796
```

This renaming made use of a supply of fresh names that, in this example, has arbitrarily renamed `a` to `s796`. This approach suffers from two disadvantages:

- It allocates far more fresh names than are actually necessary, and there is sure to be a compile-time performance cost to this.
- Plumbing the supply of fresh names to the places those names are required is sometimes very painful.

Why is there a compile-time performance cost to the sledge-hammer approach? Because a variable is a structure containing a name; to rename the variable we must copy the structure, inserting the new name. The substitution mapping old names to new names becomes larger. Finally, if the substitution is empty we can sometimes avoid looking at an expression or type at all — but if all names are cloned the substitution is never empty.

If the compiler were written in an impure language, fresh names could be allocated by side effect, but GHC is written in Haskell, which does not have side effects. Using the trees of [ARS94] is the best solution we know of, but it still involves plumbing a tree of fresh names everywhere they *might* be needed. Worse, the fresh names usually *aren't* needed, but the tree is nevertheless built. This is deeply irritating: loads of allocation for no purpose whatsoever. Finally, even if we were not worried about performance, it is sometimes extremely painful to get the name supply to where it is needed. For example, in a typed intermediate language it should be possible to have a function:

```
exprType :: Expr -> Type
```

that figures out the type of an expression. But suppose the expression is something like:

```
filter Int pred xs
```

The function `filter` has the polymorphic type

```
filter :: forall a. (a -> Bool) -> [a] -> [a]
```

So to figure out the type of the subexpression (`filter Int`) we must instantiate `filter`'s type, substituting `Int` for `a`. Oh no! Substitution! That can, in general, give rise to name capture. So we need to feed a name supply to `exprType`:

```
exprType :: NameSupply -> Expr -> Type
```

This "solution" is deeply unattractive, and the situation is only different in its cosmetics if the name supply is hidden in a monad. Something better is required.

3.2 The rapier

Suppose we write the call *subst* M $[E/x]$ to mean the result of substituting E for x in M. The standard rule for substitution [Bar85] when M is a lambda abstraction is:

$$\begin{array}{lll} subst\ (\lambda x.M)\ [E/x] & = & \lambda x.M \\ subst\ (\lambda x.M)\ [E/y] & = & \lambda x.(subst\ M\ [E/y]) \\ & & \text{if } x \text{ does not occur free in E} \end{array}$$

If the side condition does not hold, one must rename the bound variable x to something else. The brute-force solution does this renaming regardless.

Suppose that we lacked a name supply, but instead knew the free variables of E. Then we could test the side condition easily and, in the common case where there is no name capture, find that there was no need to rename x. But what if x *was* free in E? Then we need to come up with a fresh name for x that is not free in E. A simple approach is to try a variant of x, say "$x1$". If that, too, is free in E, try "$x2$", and so on.

When we finally discover a name, xn, that is not free in E, we can augment the substitution to map x to xn and apply this substitution to M, the body of the lambda. In general, then, we must simultaneously substitute for several variables at once.

To make this work at all, though, we need to know the free variables of E, or, more generally, the free variables of the range of the substitution. One way to find this is simply to compute the free variables directly from E, but if E is large this might be costly. However, it suffices to know any *superset* of these free variables. One obvious choice is *the set of all variables that are in scope*. If we made this choice, then we would end up renaming any bound variable for which there was an enclosing binding. We call this the *no-shadowing strategy*, for obvious reasons. The no-shadowing strategy will rename some variables when it is not strictly necessary to do so, but it has the desirable property of idempotence: a complete pass of the simplifier that happens to make no transformations will clone no variables. This is a good thing. Usually, some parts of the program being compiled are fully-transformed before others; the no-shadowing strategy reduces gratuitous "churning" of variable names.

Thus, we are led to a substitution algorithm that has three parameters, instead of two: the expression to which the substitution is applied, the substitution itself, ϕ, and the set of in-scope variables, θ:

$$\begin{array}{lll} subst\ (\lambda x.M)\ \phi\ \theta & = & \lambda x.subst\ M\ (\phi \setminus x)\ (\theta \cup \{x\}) \\ & & \text{if } x \notin \theta \\ subst\ (\lambda x.M)\ \phi\ \theta & = & \lambda y.subst\ M\ (\phi[x \mapsto y])\ (\theta \cup \{y\}) \\ & & \text{where } y \notin \theta \end{array}$$

Notice how conveniently the set of in-scope variables can be maintained. Almost all the time, it simply travels everywhere with the substitution; we shall see some interesting exceptions to this general rule in Section 7.1.

There is one other important subtlety in this algorithm: in the case where x is not in θ we must *delete* x from the substitution, denoted $\phi \setminus x$. How could x be in the domain of the substitution, but not be in scope? Perhaps because we are indeed substituting for x as a result of some enclosing inlining. It certainly happens in practice — we have the scars to show for it, though only in situations that are too convoluted to present here.

	Number of attempts				
	0	1	2	$3-9$	10+
Mean	93.2%	1.3%	0.7%	1.6%	3.2%
Min	0.94%	0%	0%	0%	0%
Max	100%	10%	6.13%	18.2%	94%

Figure 1: Cloning rates

Occasionally, the set of in-scope variables is not conveniently to hand when starting a substitution. In that case, it is easy to find the set of free variables of the range of the substitution, and use that to get the process started.

3.3 Choosing a new name

The other choice that must be made in the algorithm is to choose a fresh name, in the (hopefully rare) cases where that proves necessary. We could just try $x1$, $x2$, and so on, but there is a danger that once $x1 \ldots x20$ are in scope, then any new x will make 20 tries before finding $x21$. A simple way out is to compute some kind of hash value from the set of in-scope variables, and use that, together perhaps with the variable to be renamed, to choose a new name. Indeed, simply using the number of enclosing binders as the new variable name gives something not unlike de Bruijn numbers (see Section 3.5). The nice thing is that any old choice will do; the only issue is how many iterations it takes to find an unused variable.

3.4 Measurements

We made some simple measurements of the effectiveness of our approach. We compiled the entire `nofib` suite, some 370 Haskell modules, comprising around 50,000 lines of code in total [Par92]. The size of each module varied from a few dozen lines to a thousand lines or so.

Figure 1 summarises how many "tries" it took to find a variable name that was not in scope. The columns show what proportion of binders required zero, one, two, 3-9, and 10 or more attempts, to find a variable name that was not already in scope. We measured these proportions separately for each module, and then took the arithmetic mean of the resulting figures. The "min" (resp "max") rows show the smallest (resp largest) proportions encountered among the entire set of modules.

The zero column corresponds to the situation where the binder is not shadowed; as expected, this is the case for the vast majority (93%) of binders. Our hash function (we simply picked an arbitrary member of the in-scope set as a hash value) is obviously too simple, though: on average 3.2% of all binders required more than ten attempts to find a fresh name, and in one pathological module almost all binders required more than ten attempts. This pathological case suggests that there is plenty of room for improvement in the hash function.

3.5 Other approaches

Another well-known approach to the name-capture problem to use de Bruijn numbers [dB80]. Apart from being entirely unreadable, this approach suffers from the disadvantage that *when pushing a substitution inside a lambda, the entire range of the substitution must have its de Bruijn numbers adjusted.* That operation can be carried out lazily, to avoid a complexity explosion when pushing a substitution inside multiple lambdas, but that means yet more administration.

It is far from clear that using de Bruijn numbers gains any efficiency, and they carry a considerable cost in terms of the opacity of the resulting program. (Programmers will not care about this, but compiler writers do.)

There is one fairly compelling reason for using de Bruijn numbers. Precisely because they do discard the original variable names, many more common sub-expressions can arise. These CSEs increase sharing of the compiler's representation of the program; they do not in general represent run-time sharing. However this compile-time sharing can be particularly important when dealing with types, which can get large. Shao, for example, reports substantial savings when using de Bruijn numbers (for types) together with hash-consing [SLM98]. However, our types are smaller than his (we are not compiling SML modules) so type sizes only become an issue for deliberately pathological programs with exponential-sized types.

Another popular approach to the name-capture problem is this: establish the invariant that every bound variable is unique in the whole program. Then, since only inlining can duplicate an expression, we can maintain the invariant by *cloning all the locally-bound variables of an inlined expression.* There are three difficulties here. First, we found in practice that (in GHC at least) there were a quite a few transformations that had to do extra work to maintain the global-uniqueness invariant. Secondly, this strategy will do more cloning than is really necessary. Thirdly, cloning the local binders of an inlined expression implies a whole extra pass over that expression, prior to simplifying the expression in its new context. Our approach, of maintaining an in-scope set, combines the cloning pass with the simplification pass, and simultaneously reduces the amount of cloning that has to be done.

3.6 Summary

Our new substitution algorithm is a simple re-working of the standard algorithm in [Bar85]. What is interesting is that the resulting algorithm seems quite practical. Even if the compiler were written in a language where name-supply plumbing was not an issue, maintaining the set of in-scope variables makes it easy to reduce the amount of cloning that is done.

In GHC, a variable's name is actually a pair of a string and a unique number. The unique is used for comparisons, but the string is used when printing (optionally augmented with the unique if there is a danger of ambiguity). When we do need to clone a name, we invent a new unique, but keep the same print-name. This makes it possible to print dumps of intermediate code that still contain names that relate to the original source program.

4 Ensuring termination

Inlining, together with beta reduction, corresponds closely to compile-time evaluation of the program, so we must clearly be concerned about ensuring that the compiler terminates. We start from a secure base: it is a fact that F_ω is *strongly normalising.* This is a complicated way of saying that the process of reducing every reducible expression (redex) in a F_ω program will surely terminate. However, GHC's intermediate language extends F_ω. These extensions introduce non-termination in two distinct ways:

Recursive bindings. If a recursively-bound variable is inlined at one of its occurrences, that will introduce a new occurrence of the same variable. Unless restricted in some way, inlining could go on for ever.

Recursive data types. Consider the following Haskell definition for `loop`:

```
data T = C (T -> Int)

g = \y -> case y of
            C h -> h y

loop = g (C g)
```

Here, `g` is small and non-recursive, so when processing `g (C g)`, `g` will be inlined. But the inlined call very soon rewrites to `g (C g)`, which is just the expression we started with.

The problem here is that the data type `T` is recursive, and *it appears contravariantly in its own definition* [How96].

Of these two forms of divergence, the former is an immediate and pressing problem, since almost any interesting Haskell program involves recursion. The rest of this section focuses entirely on recursive definitions.

In contrast, the latter situation is rather rare, and (embarrassingly) GHC can still be persuaded to diverge by such examples. The most straightforward solution is to spot such contravariant data types, and disable the case-elimination transformation

```
  case (C g) of { C h -> ...h... }
==>
  ...g...
```

The question of spotting contravariant data types is complicated by the fact that Haskell data types can be parameterised and mutually recursive. The MLj compiler [BKR98] restricts data types declarations somewhat, but does perform the analysis for exactly this reason.

Before discussing recursive bindings, it is worth noting two other possible sources of divergence that a Haskell compiler does not have to deal with. Firstly, in an untyped setting (such as a Scheme compiler) one can easily construct terms such as

```
  (\x -> x x) (\x -> x x)
```

This expression is not explicitly recursive, but it nevertheless reduces to itself. However, the strong-normalisation theorem for F_ω tells us that such terms simply must be ill-typed.

Secondly, side effects (which Haskell lacks) can create a recursive structure. For example[1]:

```
(let ((foo a-special-value)
      (bar a-special-value))
  (begin
     (set! foo (lambda ..bar..))
     (set! bar (lambda ..foo..))
     body))
```

Here, `foo` and `bar` are mutable locations, each of which is updated to refer to the other.

4.1 The problem

From now on we focus our attention on recursive bindings. We call a group of bindings wrapped in `rec` a *recursive group*. Unrestricted inlining of non-recursive bindings is safe, but unrestricted inlining of recursive bindings might lead to non-termination. One obvious thing to do, therefore, is to ensure that each recursive group really *is* recursive. To discover this, we regard each variable in the group as a *node*, and we record an *edge* from `f` to `g` if `f`'s right hand side mentions `g` (so `f` depends on `g`). The resulting collection of nodes and edges describes a graph, called the *dependency graph*, whose strongly connected components are the smallest possible recursive groups [Pey87]. To exploit this observation, GHC constructs the dependency graph for each `let rec`, and analyses its strongly-connected components. If there is more than one component, the `let rec` is split into a nest of recursive and non-recursive `let`s. GHC performs this analysis regularly; quite often, groups that were mutually-recursive fall into separate strongly-connected components as a result of earlier transformations.

So much is well known. But what do we do when we are faced with a genuinely recursive group? The simplest thing to do is not to inline any recursively-bound variables at all, and that is what earlier versions of GHC did. But this conservative strategy loses obviously-useful optimisation opportunities. Consider a recursive group of bindings:

```
let rec
  f = \ x -> ...g...
  g = \ y -> ...f...
in
...f...
```

By convention, other variables of interest, such as `g` in this case, are assumed not to be free in `...f...`. Since only `f` is called outside the `rec`, we can inline `g` at its unique call site to give:

```
let rec
  f = \ x -> ...(...f...)...
in
...f...
```

Here, the gain is modest. But sometimes inlining in `recs` is critically important. Consider this:

[1]Thanks to Manuel Serrano for pointing this out.

```
let
  eq = ...
in
let rec
  d   = (eq,  neq)
  neq = \a b -> case d of
                  (e,n) -> not (e a b)
in
...
```

GHC generates code quite like this for an "`Eq` dictionary". A "dictionary" is a bundle of related "methods" for operating on values of a particular type. Here, the `Eq` dictionary, `d`, is a pair of methods (ordinary functions), `eq` and `neq`; the intention is that `eq` is a function that determines whether its arguments are equal, and `neq` determines whether they are unequal.

In this example, the `neq` method is specified by selecting the `eq` method from the dictionary `d`, calling it, and negating its result. You might think that it would be more straightforward to call `eq` directly, but this code is generated by the compiler from `class` and `instance` declarations in the Haskell source code. We found that it was very hard, in general, to call the appropriate method directly; it was much easier to allow the front end to generate naive code, and let the simplifier take care of the rest.

In this particular example, `d` and `neq` are genuinely mutually recursive. Yet, if `d` were inlined in the body of `neq`, the `case` would cancel with the pair constructor, leading to the following:

```
let
  eq  = ...
  neq = \a b -> not (eq a b)
  d   = (eq,  neq)
in
...
```

Now everything is non-recursive, the definition of `neq` is improved, and inlining opportunities in the rest of the program are improved.

This is not an isolated or artificial example. Compiling Haskell's type-class-based overloading, using the dictionary-passing encoding sketched above, gives rise to pervasive recursion through these dictionaries. Failing to unravel the recursion has a devastating effect on performance, because overloaded functions include equality, ordering, and all numeric operations, some of which show up in almost any inner loop. We originally went to great lengths in the front end to avoid generating unnecessary dictionary recursion but, no matter how hard we tried, some unnecessary `recs` *still* showed up. Our new approach uses a much simpler translation scheme, along with an inliner that does a good job of inlining `rec`-bound variables. This approach has the merit that it works equally well for complex recursions written by the programmer, though admittedly these are much less common.

4.2 The solution

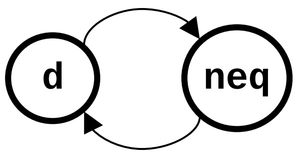
The real problem with recursive bindings is that they can make the inliner fall into an infinite loop. The key insight is this:

- *The inliner cannot loop if every cycle in the dependency graph is broken by a variable that is never inlined.*

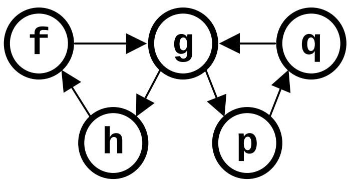
The conservative scheme works by never inlining *any* recursively-bound variable, but that is over-kill, as we saw in the example in Section 4.1:

```
rec
  d   = (eq,  neq)
  neq = \a b -> case d of
                 (e,n) -> not (e a b)
```

we obtained much better results by inlining `d` (but not `neq`) than by inlining neither. The dependency graph for this group forms a circle, thus:



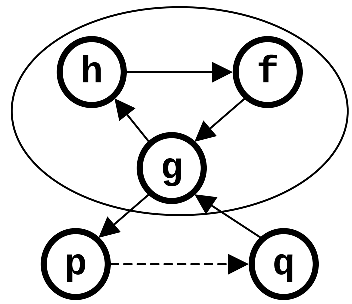
To prevent the inliner diverging, it suffices to choose *either* of `d` or `neq`, and refrain from inlining it. In a more complicated situation, however, it might not be at all obvious which variable(s) suffice to break all the loops. For example, consider this more complex dependency graph:



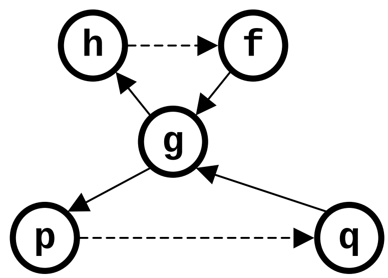
In this graph, we can break all the loops by picking `g` alone, or `f` and `q`, or `h` and `p`, or a variety of other pairs. To exploit this idea, we enhance the standard `rec`-breaking dependency analysis described above, in the following way. For each `rec` group, we construct its dependency graph, and then execute the following algorithm:

1. Perform a strongly-connected component analysis of the dependency graph.
2. For each strongly-connected component of the graph, perform the following steps, treating the components in topologically-sorted order; that is, deal first with the component that does not refer to any of the other components, and so on.
   (a) If the component is a singleton that does not depend on itself, do nothing.
   (b) Otherwise, choose a single variable, the *loop-breaker*, that will not be inlined. This choice is made using a heuristic we discuss shortly (Section 4.3).
   (c) Take the dependency graph of the component (a subset of the original graph), and delete all the edges in this graph that terminate at the loop-breaker.
   (d) Repeat the entire algorithm for this new dependency graph, starting with Step 1.

The result of the algorithm is an ordered list of bindings with the following property: *the only forward references are to loop-breakers.* The bindings are still, of course, mutually recursive, but all the non-loop-breakers can be treated *exactly like non-recursive* `lets` so far as the inliner is concerned: their definition occurs before any of their uses, and inlining them cannot cause non-termination. For example, consider the five-node dependency graph given above. It forms a single strongly-connected component. Suppose we pick `q` as a loop breaker; we delete arcs leading to it and perform the strongly-connected component analysis again. The reduced dependency graph has three strongly-connected components, namely {`p`}, {`f`, `g`, `h`}, and {`q`}



(We use dashed arcs for the arcs that are deleted in step (c).) Suppose now that we choose `f` as the loop breaker. Now we have no strongly connected components left in the reduced graph:



Notice that the only forward arcs are the dashed arcs leading to loop breakers. Reconstructing the recursive group in topologically sorted order (left to right in the diagrams) gives:

```
rec
  p  = ...q...
  h  = ...f...
  g  = ...h...
  f* = ...g...
  q* = ...g...
```

The "*" indicates the loop breakers. Only the loop breakers are referred to in the group earlier than they are defined, considering the definitions top to bottom. This is a wonderful property. As we shall see later (Section 6), inlining even non-recursive `let`-bound variables is far from straightforward, and having to worry about recursion would only make it worse. The beauty of the loop-breaking algorithm means that recursive `lets` can be treated essentially identically to non-recursive `lets`, thereby factoring the problem into two independent pieces: first cut the loops, and then treat recursive and non-recursive bindings uniformly.

4.3 Selecting the loop breaker

There are two criteria that one might use to select a loop breaker:

- Try not to select a variable that it would be very beneficial to inline.
- Try to select a variable that will break many loops.

GHC currently uses only the first of these criteria. The second is a bit tricky to predict, and we have not explored using it. To evaluate the first criterion, GHC crudely "scores" each variable by how keen GHC is to inline it. Specifically, we pick the first of the following criterion that applies to the binding in question:

Score = 3, if the right hand side is just a constant or variable. In this case the binding will certainly be inlined.

Score = 3, if the variable occurs just once (counting both the right hand sides of the `rec` itself and the body of the `let`). The variable is likely to be inlined if it occurs only once.

Score = 2, if the right hand side is a constructor application. Thus, we avoid selecting "`d`" in the example in Section 4.1, because its right hand side is a pair.

Score = 1, if the variable has rewrite rules or specialisations attached to it. Details of this are beyond the scope of this paper.

Score = 0, otherwise.

Then we pick a loop breaker by arbitrarily choosing one of the variables with lowest score. While this scoring mechanism is very crude, it seems adequate. In practice, we have never come across a `rec` in which a different choice of loop breaker would have made a significant difference. This amounts to anecdotal evidence only; we have not tried systematically to measure the effectiveness of loop-breaker choice.

4.4 Other approaches

A much more common approach to termination, taken by both [Ser97] and [WD97], is to bound both the *effort* that the inliner is prepared to invest, and the *size* of the expression it is prepared to build, when inlining a particular call. If either limit is exceeded, the inliner abandons the attempt to inline the call. Bounding effort deals with expressions, such as `(\x->x x)(\x->x x)`, that do not grow, but do not terminate either. The effort bound is typically set quite high, to allow for cascading transformations, so an effort bound alone might produce very large residual programs; that is why the size bound is necessary as well.

A variant of the approach retains a stack of inlinings that have been begun but not completed. When examining a call, the function is not inlined if an inlining of that same function is already in progress, or "pending". In effect, that function becomes the loop breaker, but it is chosen dynamically rather than statically.

This approach has the very great merit that it deals readily with all forms of non-termination: recursive functions, recursive data types, untyped languages and side effects, for example, all cause no problems. The difficulty with this approach in our setting is that the simplifier is applied repeatedly, a dozen times or more, between applying other transformations (strictness analysis, let-floating, etc). If each iteration accepts a given amount of code growth, or effort applied, then *each iteration might unroll a recursive function further.* The effort/size bound mechanism uses an auxiliary parameter (the effort/size budget) that is not recorded in the tree between successive iterations of the simplifier; it records the state of the inliner itself.

Our approach does not have this problem — successive applications of the simplifier will eventually terminate. However, our more static analysis required that recursive functions and recursive data types be handled differently, which is undesirable. And yet more would be needed in an untyped or impure setting.

A quite separate, complementary, approach to inlining recursive functions is variously described by [App94] ("loop headers"), [Ser97] ("labels-inline"), [DS97] ("lambda-dropping"), and [San95] ("the static argument transformation"). The common idea is to turn a recursive function definition into a non-recursive function containing a local, recursive definition. Thus we can, for example, transform the standard recursive definition of `map`:

```
map = \f xs -> case xs of
                  [] -> []
                  (x:xs) -> f x : map f xs
```

into the following non-recursive definition:

```
map = \f xs ->
      let mp = \xs -> case xs of
                         []     -> []
                         (x:xs) -> f x : mp xs
      in mp xs
```

With the original definition, inlining would simply unroll a finite number of iterations of `map`. With the new definition, inlining `map` creates a new, specialised function definition for `mp` into which the particular `f` used at the call site can be inlined, perhaps resulting in better code — claimed benefits range from 1% to 10%. The overall effect is much better than that achieved by simply unrolling the original definition of `map`; unrolling a loop reduces the overheads of the loop itself, whereas creating a specialised function, `mp`, reduces the cost the computation in each iteration of the loop.

The static argument transformation may indeed be useful, but it is orthogonal to the main thrust of this paper. It is best considered as a separate transformation, performed on `map` before inlining is begun, that enhances the effectiveness of inlining.

4.5 Results

It is hard to offer convincing measurements for the effectiveness of the loop-breaker algorithm, because GHC is now built in the expectation that `rec`s that can be broken will be. Nevertheless, Figure 2 gives some indicative results. It shows the the effect of switching the loop-breaker algorithm

Allocations	No libs	Libs too
Mean	+23%	+78%
Min	−15%	0%
Max	+200%	+1125%

Figure 2: Effect on total allocation of switching off the loop-breaker algorithm

off, by marking *every* `rec`-bound variable as a loop breaker. The "Mean" row shows the *geometric* mean of the ratio between the switched-off version and the baseline version — we use a geometric mean because we are averaging ratios [FW86]. The "Min" and "Max" rows show the most extreme ratios we found.

The effects are dramatic. The column headed "No libs" has the loop-breaking algorithm switched off when compiling the application, but not when compiling the standard libraries. The column "Libs too" shows the effect of switching off the loop-breaking algorithm when compiling the standard libraries as well. The importance of the libraries is that they contain implementations of arithmetic over basic types; if that is compiled badly then performance suffers horribly. (We are investigating the strange −15% figure, which suggests that switching off loop breakers improved at least one program.)

4.6 Summary

In retrospect, the algorithm is entirely obvious, yet we spent ages trying half-baked hacks, none of which quite worked, before finally biting the bullet and finding it quite tasty. It is more likely to be important for compilers for lazy languages than for strict ones, because only non-strict languages allow recursive data structures, and it is there that the most important performance implications show up. However, as our first example demonstrated, even where no data structures are involved, useful improvements can be had.

All of this is entirely orthogonal to the question of loop unrolling. A loop breaker could be inlined a fixed number of times to gain the effect of loop unrolling.

5 Overall architecture

The GHC inliner tries to do as much inlining as possible in a single pass. Since inlining often reveals new opportunities for further transformations, the inliner is actually part of GHC's *simplifier*, which performs a large number of local transformations [PJS98]. In this section we give an overview of the simplifier to set the scene for the rest of the paper.

5.1 The simplifier

The simplifier takes a substitution, a set of in-scope variables, an expression, and a "context", and delivers a simplified expression:

```
simplExpr :: Subst -> InScopeSet
          -> InExpr -> Context
          -> OutExpr
```

The real simplifier's type is a bit more complicated than this: it takes an argument that enables or disables individual transformations; it gathers statistics about how many transformations are performed; and it takes a name supply, to use when it has to conjure up a fresh name not based on an existing name[2]. However, we will not need to consider these aspects here.

The substitution and in-scope set perform precisely the roles described in Section 3, but, as we shall see, they both have further uses. The *context* tells the simplifier something about the context in which the expression appears (e.g. it is applied to some arguments, or it is the scrutinee of a `case` expression). This context information is important when making inlining decisions (Section 7.5).

We refer to an un-processed expression as an "in-expression", and an expression that has already been processed as an "out-expression", and similarly for variables. The reasons for making these distinctions will become apparent (Section 6.2).

```
type InVar  = Var
type InExpr = Expr
type InAlt  = Alt

type OutVar  = Var
type OutExpr = Expr
type OutAlt  = Alt
```

As indicated in Section 2, the simplifier treats an entire Haskell module (which GHC treats as a compilation unit) as a sequence of bindings, some recursive and some not. It deals each of these bindings in turn, just as if they were in a nested sequence of `let`s.

5.2 The occurrence analyser

It is clear that whether to inline `x` depends a great deal on how often `x` occurs in `E`. Before each run of the simplifier, GHC runs an *occurrence analyser*, a bottom-up pass that annotates each binder with an indication of how it occurs, chosen from the following list:

`LoopBreaker.` The occurrence analyser executes the dependency-graph algorithm we discussed in Section 4.1, marking loop breakers, and sorting the bindings in each `rec` so that only loop breakers are referred by an earlier definition in the sequence. Building the dependency graph uses precisely the information that the occurrence analyser is gathering anyway, namely information about where the bound variables of the `rec` occur.

`Dead.` The binder does not occur at all. For a `let` binder (whether recursive or not), the binding can be discarded, and the occurrence analyser does so immediately, so that it does not need to analyse the right hand side(s).

`OnceSafe.` The binder occurs exactly once, and that occurrence is not inside a lambda, nor is a constructor ar-

[2] We could certainly do without this name supply, by conjuring up names based on an arbitrary base name, but it turns out that it can conveniently piggy-back on the (monadic) plumbing for the other administrative arguments.

gument. Inlining is unconditionally safe; it duplicates neither code nor work. Section 2.2 explained why we must not inline an arbitrary expression inside a lambda, and also described the trivial-constructor-argument invariant.

`MultiSafe`. The binder occurs at most once in each of several distinct case branches; none of these occurrences is inside a lambda. For example:

```
case xs of
      []     -> y+1
      (x:xs) -> y+2
```

In this expression, `y` occurs only once in each case branch. Inlining `y` may duplicate code, but it will not duplicate work.

`OnceUnsafe`. The binder occurs exactly once, but inside a lambda. Inlining will not duplicate code, but it might duplicate work (Section 2.2).

`MultiUnsafe`. The binder may occur many times, including inside lambdas. Variables exported from the module being compiled are also marked `MultiUnsafe`, since the compiler cannot predict how often they are used.

Notice that we have three variants of "occurs once" (`OnceSafe`, `MultiSafe`, and `OnceUnsafe`). We have found all three to be important.

Some lambdas are certain to be called at most once. Consider:

```
let x = foo 1000
    f = \y -> x+y
in case a of
      []     -> f 3
      (b:bs) -> f 4
```

Here `f` cannot be called more than once, so no work will be duplicated by inlining `x`, even though its occurrence is inside a lambda. Hence, it would be better to give `x` an occurrence annotation of `OnceSafe`, rather than `OnceUnsafe`.

We call such lambdas *one-shot lambdas*, and mark them specially. They certainly occur in practice — for example, they are constructed as join points by the case-of-case transformation (for details see [PJS98]). We are (still) working on a type-based analysis for identifying one-shot lambdas [WP99]. Details of this analysis are beyond the scope of this paper, but our point here is that they are beautifully easy to exploit: the occurrence analyser simply ignores them when it is computing its "inside-lambda" information.

5.3 Summary

The overall plan for GHC's simplifier is therefore as follows:

```
while something-happened && iterations < 4
do
      perform occurrence analysis
      simplify the result
end
```

The simplifier alternates between occurrence analysis and simplification, until the latter indicates that no transformations occurred, or until some arbitrary number (currently 4) of iterations has occurred. This entire algorithm is applied between other major passes, such as specialisation, strictness analysis [PP93], or let-floating [PPS96].

GHC is capable of wholesale inlining across module boundaries. Whenever GHC compiles a module `M` it writes an "interface file", `M.hi`, that contains GHC-specific information about `M`, including the full Core-language definitions for any top-level definitions in `M` that are smaller than a fixed threshold. (This threshold is chosen so that few, if any, larger functions could possibly be inlined, regardless of the calling context.) When compiling any module, `A`, that imports `M`, GHC slurps in `M.hi`, and is thereby equipped to inline calls in `A` to `M`'s exports. Since the definition of function exported from `M` might refer to values *not* exported from `M`, GHC dumps into `M.hi` the transitive closure of all (sufficiently small) functions reachable from `M`'s exports. Values that are not exported from `M` may not be mentioned directly by the programmer, but may nevertheless be inlined by the inliner.

The consequence of all this is that `A` may need to be recompiled if `M` changes. There is no avoiding this, except by disabling cross-module inlining (via a command-line flag). GHC goes to some trouble to add version stamps to every inlining in `M.hi` so that it can deduce whether or not `A` *really* needs to be recompiled.

6 The three-phase inlining strategy

After considerable experimentation, GHC now makes an inlining decision about a particular `let` bound variable at no fewer than three distinct moments. In this section we explain why. Consider again the expression:

```
let x = E in B
```

PreInlineUnconditionally. When the simplifier meets the expression for the first time, it considers whether to inline `x` unconditionally in `B`. It does so if and only if `x` is marked `OnceSafe` (see Section 5.2). In this case, the simplifier does not touch `E` at all; it simply binds `x` to `E` in its current substitution, discards the binding completely, and simplifies `B` using this extended substitution. This is the main use of the substitution beyond dealing with name capture, but it needs a little care, as we discuss in Section 6.2.

Notice, crucially, that *the right hand side of the definition is processed only once*, namely at the occurrence site. It turns out that this is very important. If the right hand side *is* processed when the `let` is encountered, and then again at the occurrence of the variable, the complexity of the simplifier becomes exponential in program size. Why? Because the right hand side is processed twice; and it might have a `let` whose right hand side is then processed twice *each time*; and so on. In retrospect this is obvious, but it was very puzzling at the time!

PostInlineUnconditionally. If the pre-inline test fails, the simplifier next simplifies the right hand side, `E`, to produce `E'`. It then again considers whether to inline `x` unconditionally in `B`. It decides to do so if and only if

- `x` is not exported from this module (exported definitions must not be discarded), and
- `x` is not a loop breaker, and
- `E'` is *trivial* – that is, a literal or variable[3]. Neither work nor code is is duplicated if a trivial expression is inlined.

If so, then again the binding is dropped, and `x` is mapped to `E'` in the substitution.

This case is quite common; it corresponds to copy propagation in a conventional compiler. It often arises as a result of β-reduction. For example, consider the definitions:

```
f = \x -> E
t = f a
```

If `f` is inlined, we get a β redex, and thence

```
f = \x -> E
t = let x = a in E
```

The interesting question is why we do not make this test at the PreInlineUnconditionally stage, something we discuss below.

CallSiteInline. If neither of the above holds, GHC retains the `let` binding, adds `x` to the in-scope set. While processing `B`, at every *occurrence* of `x`, GHC considers whether to inline `x`. This decision is based on a fairly complex heuristic, that we discuss in Section 7. If the decision is "Yes", then GHC needs to have access to `x`'s definition; this can be achieved quite elegantly, as we discuss in Section 6.3.

6.1 Why three-phase?

An obvious question is this: why not combine PostInlineUnconditionally with PreInlineUnconditionally? That is, before processing `E`, why not look to see if it is trivial (e.g. a variable), and if so inline it unconditionally? Doing so is a huge, but rather subtle, mistake.

The mistake is to do with the correctness of the precomputed occurrence information. Suppose we have:

```
let
  a = ...big...
  b = a
in
...b...b...b...
```

`a` will be marked `OnceSafe`, and hence will be inlined unconditionally. But if PreInlineUnconditionally now sees that `b`'s right-hand side is just `a`, and inlines `b` everywhere, `a` now effectively occurs in many places. This is a disaster, because `a` is now inlined unconditionally in many places.

The cause of this disaster is that `a`'s occurrence information was rendered invalid by our decision to inline `b`. Several solutions suggest themselves — for example, provide some mechanism for fixing `a`'s occurrence information; or get the occurrence analyser to propagate `b`'s occurrences to `a` — and we tried some of them. They are all complicated, and the result was a bug farm.

We finally discovered the three-phase inline mechanism we have described. It is simple, and obviously correct. The PreInlineUnconditionally phase only inlines a variable `x` if `x` occurs once, not inside a lambda. That means that *the occurrence information for any variable,* `y`*, free in* `x`*'s right hand side is unaffected by the inlining.*

On the other hand, once the right hand side has been processed, if `y` is going to be inlined unconditionally, then that will have happened already. In our example, PreInlineUnconditionally will decide to inline `a`. Now the simplifier moves on to the binding for `b`. PreInlineUnconditionally declines to inline, so the right hand side of `b` is processed; `a` is inlined, and (a processed version of) `...big...` is produced. This is not trivial, so PostInlineUnconditionally declines too.

Another obvious question is whether PostInlineUnconditionally could be omitted altogether, leaving CallSiteInline to do its work. Here the answer is clearly "yes"; PostInlineUnconditionally is just an optimisation that allows trivial bindings to be dropped a little earlier than would otherwise be the case. To summarise, the key feature of our three-phase inlining strategy is that it allows the use of simple, pre-computed occurrence information, while still avoiding the exponential blowup that can occur if PreInlineUnconditionally is omitted.

6.2 The substitution

As we mentioned at the start of Section 6, the simplifier carries along (a) the current substitution, and (b) the set of variables in scope. But since the simplifier is busy transforming the expression and cloning variables, we have to be more precise:

- The domain of the substitution is *in-variables.*
- The in-scope set consists of *out-variables.*

(We discussed in-variables and out-variables in Section 5.) But what is the *range* of the substitution? When used for cloning or PostInlineUnconditionally the range was an *out-expression*, but when used in PreInlineUnconditionally the range was an *in-expression.* But watch out! Since we are, in effect, deferring the simplification of the in-expression, we must also record the substitution appropriate to the original site of the expression. Thus we are led to the following definition for the substitution:

```
type Subst     = FiniteMap InVar SubstRng
data SubstRng = DoneEx OutExpr
              | SuspEx InExpr Subst
```

A `DoneEx` is straightforward, and is used both by the name-cloning mechanism, and by PostInlineUnconditionally. A `SuspEx` (`Susp` for "suspended") is used by PreInlineUnconditionally, and pairs an in-expression with the substitution appropriate to its `let` binding; you can think of it as a suspended application of `simplExpr`. Notice that we do *not*

[3] Or, in the real compiler, a type application.

capture the in-scope set as well. Why not? Because we must use the in-scope set appropriate to the occurrence site — Section 7.1 amplifies this point.

6.3 The in-scope set

We mentioned earlier (Section 6) that the simplifier needs access to a `let`-bound variable's right-hand side at its occurrence site(s). All we need is to turn the in-scope set into a finite mapping:

```
type InScopeSet = FiniteMap OutVar Definition
data Definition = Unknown
                | BoundTo OutExpr OccInfo
                | NotAmong [DataCon]
```

Whether or not a variable is in scope can be answered by looking in the domain of the in-scope set (we still call it a "set" for old times sake). But the range of the mapping records what value the variable is bound to:

`Unknown` is used for variables bound in lambda and case patterns. We don't know what value such a variable is bound to.

`BoundTo` is used for `let` bound variables (both recursive and non-recursive), and records the right-hand side of the definition and the occurrence information left with the binding by the occurrence analyser. The latter is needed when making the inlining decision at occurrence sites.

`NotAmong` is described shortly.

The in-scope set is also a convenient place to record information that is valid in only *part* of a variable's scope. Consider:

```
\x -> ...(case x of (a,b) -> E)...
```

When processing `E`, but not in the "`...`" parts, `x` is known to be bound to `(a,b)`. So, when processing the alternative of a `case` expression whose scrutinee is a variable, it is easy for the simplifier to modify the in-scope set to record `x`'s binding. Why is this useful? Because `E` might contain another `case` expression scrutinising `x`:

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

By inlining `(a,b)` for `x`, we can eliminate this `case` altogether. This turns out to be a big win [PJS98].

The `NotAmong` variant of the `Definition` type allows the simplifier to record negative information:

```
case x of
  Red     -> ...
  Blue    -> ...
  Green   -> ...
  DEFAULT -> E
```

The `DEFAULT` alternative matches any constructors other than `Red`, `Blue`, and `Green`. GHC supports such `DEFAULT` alternatives directly, rather than requiring `case` expressions to be exhaustive, which is dreadful for large data types. Inside `E`, what is known about `x`? What we know is that it is *not* bound to `Red`, `Blue`, or `Green`. This can be useful; if `E` contains a `case` expression that scrutinises `x`, we can eliminate any alternatives that cannot possibly match. Similarly, the expression `` x `seq` F `` inside `E` can be transformed to just `F`, since `NotAmong` implies that `x` is evaluated[4]. Even the value `NotAmong []` is useful: it signals that the variable is evaluated, without specifying anything about its value.

	Pre	Post	CallSite
Mean	47.4%	17.4%	35.2%
Min	0.25%	0.92%	0.72%
Max	80%	95%	98%

Figure 3: Relative frequency of inlining

The in-scope set, extended to be an in-scope mapping, plays the role of a *dynamic environment*. It records knowledge of the value of each in-scope variable, including knowledge that may be true for only part of that variable's scope. The nice thing is that this dynamic knowledge can elegantly be carried by the in-scope set, which we need anyway. The details of the transformations that exploit that dynamic knowledge are beyond the scope of this paper.

Almost all the time, the substitution and in-scope set travel together. But that is not always the case, as we discuss in Section 7.1.

6.4 Measurements

Figure 3 gives some simple measurements of the relative frequency of each form of inlining. We used the same set of benchmark programs in in Section 3.4, gathered statistics on how often each sort of inlining was used, and averaged these separately-calculated proportions. We took *arithmetic* means of the percentages, because here we are averaging "slices of the pie", so the "Mean" line should still sum to 100%.

The figures indicate that on average, each sort of inlining is actually used in practice, and that each dominates in some programs.

6.5 Summary

We can summarise the binding-site effects on the substitution and in-scope set as follows. Suppose that we encounter the binding `x = E` with a substitution `subst`, and an in-scope set `in-scope`.

PreInlineUnconditionally. The substitution is extended by binding `x` to `SuspEx E subst`. The in-scope set is not changed.

PostInlineUnconditionally. The substitution is extended by binding `x` to `DoneEx E'`, where `E'` is the simplified version of `E`. The in-scope set is not changed.

Otherwise. If `x` is not already in scope, the substitution is not changed, but the in-scope set is extended by binding `x` to `E'`. If `x` *is* already in scope, then a new variable name `x'` is invented (Section 3.3); the substitution is

[4] The expression `` E1 `seq` E2 `` evaluates `E1`, discards the result, and then evaluates and returns `E2`.

extended by binding `x` to `DoneEx x'`, and the in-scope set is extended by binding `x'` to `E'`.

This concludes the discussion of what happens at the binding site of a variable. Now we consider what happens at its occurrence(s).

7 Occurrences

When the simplifier finds an occurrence of a variable, it first looks up the variable in the substitution (Section 7.1), and then decides whether to inline it (Section 7.2).

7.1 Looking up in the substitution

When the simplifier encounters the occurrence of a variable, the latter (being an `InVar`) must be looked up in the substitution:

```
simplExpr sub ins (Var v) cont
  = case lookup sub v of
      Nothing          -> considerInline ins v cont
      Just (SuspEx e s) -> simplExpr s ins e cont
      Just (DoneEx e)   -> simplExpr empty ins e cont
```

The variable might not be in the substitution at all – for example, it might be a variable that did not need to be renamed. In that case, the next thing to do is to consider inlining it. The substitution can be discarded at this point, because the inlining (if any) is already an out-expression. Incidentally, notice that the variable we previously thought of as an `InVar` is now an `OutVar`. This is one reason that `InVar` and `OutVar` are simply synonyms for `Var`, rather than being truly distinct types.

If the substitution maps the variable to a `SuspEx`, then the simplifier is (tail) called again, passing the captured substitution, and the *current* in-scope set. The substitution and the in-scope set usually travel together, but here they do not. We must use the in-scope set from the *occurrence site* (because that describes what variables are in scope there), and the substitution from the *definition site*.

The third case is when the variable maps to `DoneEx e`. In this case you might think we were done. But suppose `e` was a variable. Then we should consider inlining it, given the current context `cont`, which differs from that at the variable's definition site. What if `e` was a partial application of a function? Again, the context might now indicate that the function should be inlined. So the simple thing to do is simply to pass `e` to `simplExpr` again. But notice that we give it the empty substitution! Consider this example:

```
\x -> let
         f = x
      in
      \x -> ...f..f...
```

When the binding for `f` is encountered, PostInlineUnconditionally will extend the substitution, binding `f` to `DoneEx x`. When the `\x` is encountered, the substitution will again be extended to bind `x` to `DoneEx x1`, because `x` is already in scope. Now, when we replace the occurrence of `f` by `x`, we must not apply the same substitution again, which would replace `x` by `x1`! The right thing to do is to continue with the empty substitution.

The code is simple enough, but it took us a long time before the interplay between the substitution and the in-scope set became as simple and elegant as it now is.

7.2 Inlining at an occurrence site

Once the simplifier has found a variable that is not in the substitution (and hence is an `OutVar`), we need to decide whether to inline it (CallSiteInline from Section 6). The first thing to do is to look up the variable in the in-scope set:

```
considerInline ins v cont
 = case lookup ins v of
    Nothing                -> error "Not in scope"

    Just (BoundTo rhs occ) |  inline rhs occ cont
                           -> simplExpr empty ins rhs cont

    Just other             -> rebuild (Var v) cont
```

If the dynamic information is `BoundTo`, and the predicate `inline` says "yes, go ahead", we simply tail-call the simplifier, passing the in-scope set and the empty substitution (as in the `DoneEx` case of the substitution). In all other cases we give up on inlining. The function `rebuild`, which we do not discuss further here, simply combines the variable with its context.

The `inline` predicate is the interesting bit. It looks first at the variable's occurrence information:

```
inline :: OutExpr -> OccInfo -> Context -> Bool
inline rhs LoopBreaker cont = False
inline rhs OnceSafe cont    = error "inline: OnceSafe"
inline rhs MultiSafe cont   = inlineMulti rhs cont

inline rhs OnceUnsafe cont  = whnfOrBot rhs &&
                              not (veryBoring cont)

inline rhs MultiUnsafe con  = whnfOrBot rhs &&
                              inlineMulti rhs cont
```

The `LoopBreaker` case is obvious. The `OnceSafe` case should never happen, because PreInlineUnconditionally will have already inlined the binding.

The `OnceUnsafe` case uses the `whnfOrBot` predicate (Section 2.2), to ensure that inlining will not happen if there is any work duplication. However, as noted in Section 2.2, even if the variable occurs just once, it is not always a good idea to inline it. The `veryBoring` predicate has type

```
veryBoring :: Context -> Bool
```

It examines the context, returning `False` if there is anything at all interesting about it, namely if and only if:

- The variable is applied to one or more arguments.
- The variable is the scrutinee of a `case`.

Notice that if a variable is the argument of a constructor, it is in a `veryBoring` context, and so it will not be inlined, thus maintaining the trivial-constructor-argument invariant (Section 2.2).

The `MultiSafe` and `MultiUnsafe` cases deal with the situation where there is more than one occurrence of the variable. Both make use of `inlineMulti` to do the bulk of the work; in addition, `MultiUnsafe` uses `whnfOrBot` to avoid work duplication.

Incidentally, since `whnfOrBot rhs` depends only on `rhs`, it is actually (lazily) cached in the `BoundTo` constructor rather than being re-calculated at each occurrence site.

7.3 Inlining multiple-occurrence variables

Now we are left with the case of inlining a variable that occurs many times.

```
inlineMulti :: OutExpr -> Context -> Bool
inlineMulti rhs cont
 | noSizeIncrease rhs cont = True
 | boring rhs cont         = False
 | otherwise               = smallEnough rhs cont
```

The third case of `inlineMulti` is the function that every inliner has: is the function small enough to inline? The first two cases are less obvious. The second case deals with the situations like this:

```
let
  f = \x -> E
in
... (let g = \y z -> (f y, f z) in ...) ...
```

There is very little point in inlining `f` at these two sites, because we can guarantee that no new transformations (beyond those already performed on `f` itself) will be enabled by doing so; the only saving is the call to `f`, and there is a code duplication cost to pay. How do we know that no transformations will be enabled? Because: (a) the arguments `y` and `z` are lambda-bound and hence uninformative; and (b) the result of both calls are simply stored in a data structure.

The predicate `boring` takes an expression (the one we are considering inlining) and a context (in which it would be inlined).

```
boring :: Expr -> Context -> Bool
```

Corresponding to our example above, `boring` returns `True` if both

(a) All the arguments to which the function is applied are types, or variables that have dynamic information of `Unknown`; and

(b) After consuming enough arguments from the context to satisfy the lambdas at the top of the function, the remaining context is `veryBoring`.

Even if the context is `boring`, however, it is still worth while inlining the function if the result of doing so is no bigger than the call [App92]. That is what the predicate `noSizeIncrease` tests. Again, one might expect this case to be rare, but it isn't. For example, Haskell data constructors are curried functions, but in GHC's intermediate language constructor applications are saturated (Section 2). We bridge this gap by producing a function definition for each constructor such as:

```
cons = \x xs -> Cons {x,xs}
```

where the `Cons {x,xs}` is the saturated constructor application. (In reality there are a few type abstractions and applications too, but the idea is the same.) These definitions also make a convenient place to perform argument evaluation (and perhaps unboxing) for strict constructors. For the simple definitions, such as `cons`, it is clearly better to inline the definition, even if the context is `boring`.

Notice that the first case is required even though `smallEnough` is sure to return `True` if `noSizeIncrease` does. Why? Because otherwise the second case might decide that the context is `boring` and decline to inline.

7.4 Size matters

We have now finally arrived at the `smallEnough` predicate, the main aspect of this paper for which there is a reasonable (albeit small) literature. We do not claim any new contribution here, though (unlike some proposals) `smallEnough` is context-sensitive:

```
smallEnough :: Expr -> Context -> Bool
```

For the record, however, the algorithm is as follows. We compute the size of the function body (having first split off its formal parameters, namely the lambdas at the top). From this size we subtract:

- The size of the call.
- An *argument discount* for each argument (extracted from the context) that (a) has dynamic information other than `Unknown`, and (b) is scrutinised by a `case`, or applied to an argument, in the function body.
- A *result discount* if the context is not `boring` and the function body returns an explicit constructor or lambda.

If the result of this computation is smaller than the *inline threshold* then we inline the function. The argument discount, result discount, and inline threshold are all settable from the command line. Santos gives more details of GHC's heuristics [San95, Section 6.3].

7.5 The context

It should by now be clear that the *context* of an expression plays a key role in inlining decisions. For a long time we passed in a variety of *ad hoc* flags indicating various things about the context, but we have now evolved a much more satisfactory story. The context is a little like a continuation, in that it indicates how the result of the expression is consumed. But this continuation *must not be represented as a function* because we must be able to ask questions of it, as the earlier sub-sections indicate.

So GHC's contexts are defined by the following data type:

```
data Context
  = Stop
  | AppCxt InExpr Subst Context
  | CaseCxt  InVar [InAlt] Subst Context
  | ArgCxt   (OutExpr -> OutExpr)
  | InlineCxt  Context
```

The `Stop` context is used when beginning simplification of a lazy function argument, or the right hand side of a `let` binding. The `AppCxt` context indicates that the expression under consideration is to be applied to an argument. The argument is as yet un-simplified, and must be paired with its substitution. Similarly, the `CaseCxt` context is used when simplifying the scrutinee of a `case` expression.

`simplExpr` simply recurses into the expression, building a context "stack" as it goes. Here, for example, is what `simplExpr` does for `App` and `Case` nodes:

```
simplExpr sub ins (App f a) cont
  = simplExpr sub ins f (AppCxt a sub cont)

simplExpr sub ins (Case e b alts) cont
  = simplExpr sub ins e (CaseCxt b alts sub cont)
```

We have already seen how useful it is to know the context of a variable occurrence. The context also makes it easy to perform other transformations, such as the case-of-known-constructor transformation:

```
  case (a,b) of { (p,q) -> E }
==>
  let {p=a; q=b} in E
```

`simplExpr` just matches a constructor application with a `CaseCxt` continuation.

The next case, `ArgCxt`, is used when simplifying the argument of a strict function or primitive operator. Here, a genuine, functional continuation is used, because no more needs to be known about the continuation.

The `InlineCxt` context is discussed in the next subsection. In practice, GHC's simplifier has another couple of constructors in the `Context` data type, but they are more peripheral so we do not discuss them here.

7.6 `INLINE` pragmas

Like some other languages, GHC allows the programmer to specify that a function should be inlined at all its occurrences, as a *pragma* in the Haskell source language:

```
{-# INLINE f #-}
f x = ...
```

GHC also allows the Haskell programmer to ask the compiler to inline a function at a particular call site, thus:

```
...(inline f a b)...
```

The function `inline` has type $\forall\alpha.\alpha$ `->` α, and is semantically the identity function. Operationally, though, it asks that `f` be inlined at this call site. Such per-occurrence inline pragmas are less commonly offered by compilers [Bak92].

Both these pragmas are translated to constructors in the `Note` data type, which itself can be attached to an expression (Section 2):

```
data Note = ...
          | InlineMe       -- {-# INLINE #-}
          | InlinePlease   -- inline
```

If they are so similar in the Core language, why do they appear so different in Haskell? Haskell allows functions to be defined by pattern-matching, using multiple equations, so there is no convenient syntactic place to ask for `f` to be inlined everywhere. At an occurrence site, however, it is natural just to use a pseudo-function.

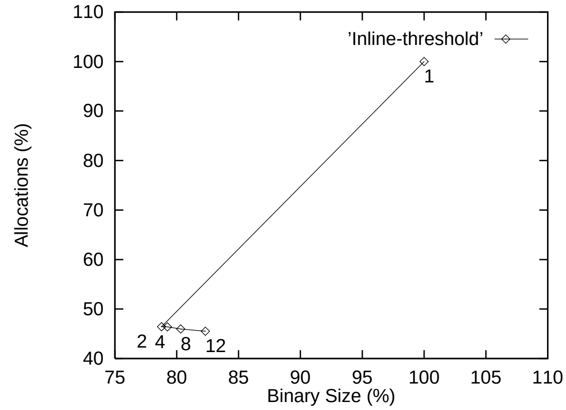


Figure 4: Effect of inlining threshold

The effects of `InlineMe` and `InlinePlease` are as follows:

- The effect of `InlineMe` is to make the enclosed expression look very small, which in turn makes the `smallEnough` predicate reply `True`. When `simplExpr` finds an `InlineMe` in a non-`boring` context, it drops the `InlineMe`, because its work is done.
- The effect of `InlinePlease` is to push an `InlineCxt` onto the context stack. The `smallEnough` predicate returns `True` if it finds such a context, regardless of the size of the expression.

There is an important subtlety, however. Consider

```
g = \a b -> ...big...
{-# INLINE f #-}
f = \x -> g x y
```

and suppose that this is the only occurrence of `g`. Should we inline `g` in `f`'s right hand side? By no means! The programmer is asking that `f` be replicated, but not `g`! The right thing to do is to switch off all inlining when processing the body of an `InlineMe`; when `f` is inlined, then (and only then) `g` will get its chance.

7.7 Measurements

As mentioned in Section 7.4, our implementation makes use of an "inline threshold" to determine whether a given expression is small enough to inline. Figure 4 shows the effect of varying this threshold on (the geometric mean of) binary size and allocation. We use allocation instead of run-time because allocation is easy to measure repeatably, and is a somewhat reliable proxy for run-time, with the notable exception of some very small programs.

The actual values for the threshold are fairly arbitrary, and are affected by some of the other parameters: discounts for evaluated arguments and so on. What is more interesting is the *shape* of the graph. As expected, beyond a certain point, binary sizes increase without having any dramatic effect on the efficiency of the program. The graph also shows that setting the threshold too low (i.e. less than 2) has a dramatic effect on both binary size and run-time. Essentially very little call-site inlining is being performed below this threshold, and even less inter-module inlining is happening (because this is covered by call-site inlining only; we can't see the binding).

The jump between threshold values 1 and 2 is caused by the fact that even functions marked `{-# INLINE #-}` are not inlined at a threshold of 1. The "wrapper" functions generated by strictness analysis are of this form, and if these wrappers are not inlined performance drops dramatically. Making measurements is very instructive: we were surprised by the rather small performance increases as the threshold is increased beyond 2, and plan to investigate this further.

8 Related work

There is a modest literature on inlining applied to imperative programming languages, such as C and FORTRAN — some recent examples are [DH92, CMCH92, CHT91, CHT92]. In these works the focus is exclusively on *procedures* defined at the *top level*. The benefits are found to be fairly modest (in the 10-20% range), but the cost in terms of code bloat is also very modest. Considerable attention is paid to the effect on register allocation of larger basic blocks, which we do not consider at all.

It seems self-evident that the benefits of inlining are strongly related to both language and programming style. Functional languages encourage the use of abstractions, so the benefits of inlining are likely to be greater. Indeed, Appel reports benefits in the range 15-25% for the Standard ML of New Jersey compiler [App92], while Santos reports average benefits of around 40% for Haskell programs [San95]. Chambers reports truly dramatic factors of 4 to 55 for his SELF compiler [Cha92]; SELF takes abstraction very seriously indeed!

The most detailed and immediately-relevant work we have found is for two Scheme compilers. Waddell and Dybvig reports performance improvements of 10-100% in the *Chez Scheme* compiler [WD97], while Serrano found a more modest 15% benefit for the Bigloo Scheme compiler [Ser95, Ser97]. Both use a dynamic, effort/size budget scheme to control termination. The *Chez Scheme* inliner uses an explicitly-encoded context parameter that plays exactly the role of our `Context` (Section 7.5).

A completely different approach to the inlining problem is discussed by [AJ97]. In this paper the focus is on inlining functions that are called precisely once, something that we have been very concerned with. Appel and Jim show that this transformation, along with a handful of others (including dead-code elimination), are normalising and confluent, a very desirable property. Their focus is then on finding an efficient algorithm for applying the transformations exhaustively. Their solution involves adjusting the results of the occurrence analysis phase as transformations proceed. Their initial algorithm has worst-case quadratic complexity, but they also propose a more subtle (and unimplemented) linear-time variant. We too are concerned about efficient application of transformation rules, but our set of transformations is much larger, and includes general inlining, so their results are not directly applicable to our setting. Nevertheless, it is a unique and inspiring approach.

Copious measurements of many transformations in GHC (not only inlining) can be found in Santos's thesis [San95]; although these measurements are now several years old, we believe that the general outlines are unlikely to have changed dramatically. [PJS98] contains briefer, but more up-to-date, measurements.

9 Conclusion

This paper has told a long story. Inlining seems a relatively simple idea, but in practice it is complicated to do a good job. The main contribution of the paper is to set down, in sometimes-gory detail, the lessons that we have learned over nearly a decade of tuning our inliner. Everyone who tries to build a transformation-based compiler has to grapple with these issues but, because they are not crisp or sexy, there is almost no literature on the subject. This paper is a modest attempt to address that lack.

Acknowledgements

We warmly thank Nick Benton, Oege de Moor, Andrew Kennedy, John Matthews, Sven Panne, Alastair Reid, Julian Seward, and the four IDL Workshop referees, for comments on drafts of this paper. Special thanks are due to Manuel Chakravarty, Manuel Serrano, Oscar Waddell, and Norman Ramsey, for their particularly detailed and thoughtful remarks.

References

[AJ97] AW Appel and T Jim. Shrinking lambda-expressions in linear time. *Journal of Functional Programming*, 7(5):515–541, September 1997.

[App92] AW Appel. *Compiling with continuations.* Cambridge University Press, 1992.

[App94] AW Appel. Loop headers in lambda-calculus or CPS. *Lisp and Symbolic Computation*, 7:337–343, 1994.

[ARS94] L Augustsson, M Rittri, and D Synek. On generating unique names. *Journal of Functional Programming*, 4(1):117–123, January 1994.

[Bak92] HG Baker. Inlining semantics for subroutines which are recursive. *ACM Sigplan Notices*, 27(12):39–49, December 1992.

[Bar85] HP Barendregt. *The lambda calculus: its syntax and semantics.* Number 103 in Studies in Logic. North Holland, 1985.

[BKR98] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In ICFP98 [ICF98], pages 129–140.

[Cha92] C. Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* Technical report STAN-CS-92-1240, Stanford University, Departement of Computer Science, March 1992.

[CHT91] KD Cooper, MW Hall, and L Torczon. An experiment with inline substitution. *Software Practice and Experience*, 21:581–601, June 1991.

[CHT92] K. Cooper, M. Hall, and L. Torczon. Unexpected Side Effects of Inline Substitution: A Case Study. *ACM Letters on Programming Languages and Systems*, 1(1):22–31, 1992.

[CMCH92] PP Chang, SA Mahlke, WY Chen, and W-M Hwu. Profile-guided automatic inline expansion for C programs. *Software Practice and Experience*, 22:349–369, May 1992.

[dB80] N de Bruijn. A survey of the project AUTOMATH. In JP Seldin and JR Hindley, editors, *To HB Curry: essays on combinatory logic, lambda calculus, and formalism*, pages 579–606. Academic Press, 1980.

[DH92] JW Davidson and AM Holler. Subprogram inlining: a study of its effects on program execution time. *IEEE Transactions on Software Engineering*, 18:89–102, February 1992.

[DS97] O Danvy and UP Schultz. Lambda-dropping: transforming recursive equations into programs with block structure. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97)*, volume 32 of *SIGPLAN Notices*, pages 90–106, Amsterdam, June 1997. ACM.

[FW86] PJ Fleming and JJ Wallace. How not to lie with statistics - the correct way to summarise benchmark results. *CACM*, 29(3):218–221, March 1986.

[How96] BT Howard. Inductive, co-inductive, and pointed types. In ICFP96 [ICF96].

[ICF96] *ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, Philadelphia, May 1996. ACM.

[ICF98] *ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, Balitmore, September 1998. ACM.

[Par92] WD Partain. The `nofib` benchmark suite of Haskell programs. In J Launchbury and PM Sansom, editors, *Functional Programming, Glasgow 1992*, Workshops in Computing, pages 195–202. Springer Verlag, 1992.

[Pey87] SL Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[PJS98] SL Peyton Jones and A Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47, September 1998.

[PP93] SL Peyton Jones and WD Partain. Measuring the effectiveness of a simple strictness analyser. In K Hammond and JT O'Donnell, editors, *Functional Programming, Glasgow 1993*, Workshops in Computing, pages 201–220. Springer Verlag, 1993.

[PPS96] SL Peyton Jones, WD Partain, and A Santos. Let-floating: moving bindings to give faster programs. In ICFP96 [ICF96].

[San95] A Santos. *Compilation by transformation in non-strict functional languages.* Ph.D. thesis, Department of Computing Science, Glasgow University, September 1995.

[Ser95] M. Serrano. A fresh look to inlining decision. In *4th International Computer Symposium (ICS'95)*, Mexico city, Mexico, November 1995.

[Ser97] M Serrano. Inline expansion: *when* and *how*? In *International Symposium on Programming Languages Implementations, Logics, and Programs (PLILP'97)*, September 1997.

[SLM98] Z Shao, C League, and S Monnier. Implementing typed intermediate languages. In ICFP98 [ICF98], pages 313–323.

[WD97] O Waddell and RK Dybvig. Fast and effective procedure inlining. In *4th Static Analysis Symposium*, number 1302 in Lecture Notes in Computer Science, pages 35–52. Springer Verlag, September 1997.

[WP99] K Wansbrough and SL Peyton Jones. Once upon a polymorphic type. In *26th ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 15–28, San Antonio, January 1999. ACM.