Secrets of the GHC inliner

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

Abstract

This is the **first complete draft** *of a paper I plan to submit to the WAAAPL workshop (deadline 16 June).*

I would be hugely grateful for constructive feedback before, or even after, then.

I have tried to summarise the literature in the related-work section at the end, but I have not spent enough time on it yet. If you know of anything I don't, or think I am misrepresenting anyone's work (esp yours), please tell me.

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 Glagow 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.

As a result of this attention, the inliner has been rewritten several times. 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. Specifically, we describe the following:

- 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 3.1).
- A major issue for any compiler, especially for one that inlines heavily, is *name capture*. Our inital brute-force solution involved inconvenient plumbing, and we have now evolved a simple and effective alternative (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, anddoing 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. Yet, 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. When discussing inlining, the most obvious question is how to choose whether to inline a function, in the hope of exposing new transformations. This is indeed an important question, and one that has absorbed some of our efforts, but we have found other algorithmic aspects of inlining far more challenging. 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, so that they are free to make new mistakes of their own.

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 langauges too.

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

Figure 1: Syntax of the Core language

2 Preliminaries

We will assume the use of a pure, non-strict, typed intermediate language, consisting of the lambda calculus agumented with `let`, `letrec`, `case`, constructors, literals, and primitive operations (Figure 1). The details of this language are not important to this paper. In presenting examples we often omit braces, and write operators infix, where the meaning is clear. We also write several bindings in a single non-recursive `let` expression as shorthand for a squence of `let` expressions.

Our intermediate language is a variant of System $F\omega$, and supports type abstraction and application, but these aspects raise no new issues, and we do not dicuss them here except in passing.

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 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 replaces `(\x->E) A` by `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.

We use the term "inline" equally for 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 a non-strict language, inlining is unconditionally valid (meaning-preserving) regardless of

- the number of occurrences of `x`,
- whether or not the binding for `x` is recursive, and
- the form of `E` (e.g. whether or not it is a lambda abstraction).

Of course, 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 (Section 6).

Inlining subsumes copy propagation:

```
  let x = y in ...x...x...
==>
  ...y...y...
```

We informally use "`...x...`" to stand for an expression that has `x` as a free variable. In this example, we simply inline `x` at each of its occurrences.

3 Recursive bindings

It is, in general, dangerous to inline a recursive definition.

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

If `f` is a recursive function, then if we inline `f` at one of its call sites, we now have a new occurrence of `f`; if we inline that, we get a new occurrence, and so on. This is essentially loop unrolling; it may be a good thing, but there must be some limit.

So recursive groups are bad, and we should try to get rid of them at every opportunity. One obvious thing to do is to ensure that the 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.

3.1 The problem

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 `rec`s is critically important. Consider this:

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

GHC generates code quite like this for an `Eq` dictionary. The pair `t` is a pair of functions, `eq` and `neq`. The `neq` function is defined by selecting the `eq` function from `t` and negating its result. Thus `t` and `neq` are genuinely mutually recursive. Yet, if `t` 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)
  t   = (eq,  neq)
in
...
```

Now everything is non-recursive, the definiton of `neq` is improved, and inlining opportunities in rest of the program are improved. It turns out that one gets dreadful code for overloaded Haskell programs if one does not somehow succeed in unravelling such apparently-recursive definitions.

You may say that GHC is silly to generate such strange `rec`s, and indeed we originally went to some lengths to generate the final form directly. But we found that more and more effort was being invested in avoiding creating unnecessary `rec`s, and *still* some of them showed up. The Right Thing To Do is to do a better job of inlining `rec`-bound variables; that involves a bounded amount of work in a single place.

3.2 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. For example, consider:

```
let rec
   f = ...g...
   g = ...h...
   h = ...f...
in
...f...
```

The dependency graph for the `rec` forms a circle, thus: To prevent the inliner diverging, it suffices to choose *any one* of `f`, `g`, `h`, and refrain from inlining it. In a more compliciated situation, however, it might not be at all obvious which variable(s) suffice to break all the loops. For example, in this dependency graph, we can break all the loops by picking `g`, 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 dependency 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 3.3).
   (c) Take the dependency graph of the component (a subset of the original graph), and delete all the edges in the graph that terminate at the loop-breaker.
   (d) Repeat the entire algorithm for this new dependency graph.

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.

This is a wonderful property. As we shall see later (Section 6), inlining for `let`-bound variables is far from straightforward, so it is excellent that the loop-breaking algorithm means that recursive `lets` can be treated essentially identically to non-recursive `lets`.

We should mention here that our initial claim is not correct. Even in the absence of recursive functions, it is possible to cause the inliner to diverge, using a standard categorical trick Consider the following definition for `urk`:

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

g = \y -> case y of
                C h -> h y
urk = 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)`, and round we go. This way of using a recursive data type to synthesise a recursive function reveals that our original claim that identifying loop-breakers prevents divergence is false if arbitrary data types are allowed. We know of no elegant way to identify when this occurs without penalising perfectly innocent (and important) transformations. One solution is simply to place an upper bound on the number of functions the inliner will inline in any one pass. GHC currently takes the simpler strategy of diverging; no real program has so far elicited this behaviour.

3.3 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:

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 "`t`" in the example in Section 3.1, 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. We no longer find programs with `recs` that can easily be broken.

3.4 The static argument transformation

A 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 defintion 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 `map`. With the new definition, inlining `map` creates a new local 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%.

This transformation may indeed be useful, but it is orthogonal to the main thrust of this paper. It is best considered as a separate tranformation, performed on `map` before inlining is begun.

3.5 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 strutures, 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.

4 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
    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`.

4.1 The sledge hammer

Earlier versions of GHC used a sledge-hammer approach: during inlining, GHC would simply rename 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 performance cost to this.
- Plumbing the supply of fresh names to the places those names are required is sometimes very painful.

If the compiler were written in an impure language, fresh names can 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 are *not* 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`. 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.

4.2 De Bruijn numbers

Another well-known approach is 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.

4.3 Using the in-scope set

The standard rule for substitution [Bar85] in a lambda abstraction is

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

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 sustitution to map x to xn and apply this substitution to M, the body of the lambda. In general, then, we will 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 gather the free variables directly, 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 is still conservative, but it has the merit of being idempotent: once it is established, if no transformations take place, then no gratuitous renaming will take place either.

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:

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

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.

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.

4.4 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. The nice thing is that any old choice will do; the only issue is how many iterations it takes to find an unused variable.

So far, choosing new names has not proved a bottleneck, so we have not tried very hard to get the number of iterations down.

4.5 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.

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. 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. However, we will not need to consider these aspects here.

The substitution and in-scope set perform precisely the roles described in Section 4, 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 some arguments, or it is the scrutinee of a `case` expression). This context information is important when making inlining decisions.

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.

```
type InVar  = Var
type InExpr = Expr

type OutVar  = Var
type OutExpr = Expr
```

As indicated in Figure 1, 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.

5.1 Factors affecting inlining

When making inlining decisions, there are three distinct factors to consider:

- *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. Since it is possible to make a program arbitrarily slower in this way, GHC is very conservative about work duplication. In general, GHC never duplicates work unless it is sure that the duplication is a small, bounded amount.

- *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. Code size is a fuzzier issue than work duplication. Bloated programs are bad (increased compilation time, less locality at runtime), 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.

- *Is anything gained by inlining?* For example, in the expression

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

  nothing at all would be gained by inlining `f`, thus:

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

  Indeed, if there were many calls to `g`, there would be a small loss, because the closure for `(\x -> E)` would have to be allocated each time `g` was called.

How can we figure out whether any work or code is duplicated? Consider the expression

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

We can analyse the expression to discover (a) how `x` is used in `B` (many times? once? never?), and (b) what `E` is like (is it big? is it a value?). We study these two analyses in the next two subsections.

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 3.1, marking loop breakers, and ordering the bindings in each `rec` so that only loop breakers are referred in an earlier definition in the sequence.

`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 annotate the right hand side(s).

`OnceSafe.` The binder occurs exactly once, and that occurrence is not inside a lambda. Inlining is unconditionally safe; it duplicates neither code nor work.

`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. For example:

```
let x = foo 1000
    f = \y -> x + y
in (f 1, f 2, f 3)
```

If we inline `x` inside the `\y`-abstraction, the call to `foo` will be made three times, once for each call of `f`.

`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. We call such lambdas *one-shot lambdas*, and mark them specially. They certainly occur in practice.

We are (still) working on a type-based analysis for identifying one-shot lambdas [WJ99], but it turns out that some lambdas are one-shot by construction, notably those constructed as join points by the case-of-case transformation (for details see [JS98]. Details of identifying one-shot lambdas is 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 The form of the right hand side

Consider our expression

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

If `x` is inlined in more than one place, or inlined inside a (non-one-shot) lambda, we have to worry about work duplication. When will work *not* be duplicated? When `E` is:

- A variable.
- A partial application; that is, the application of a function to fewer arguments than the function requires.
- A constructor application.
- A lambda abstraction
- An expression that is sure to diverge.

The first four are collectively called a weak head normal form (WHNF). The last is more surprising, but it is very useful in practice. Consider:

```
fail = error "Urk"
f    = \x -> if x==0 then fail else ...
```

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. It is perfectly OK – indeed, it is an excellent thing – to inline `fail`, because if `fail` is ever called, execution is going to halt anyway, so there is no work-duplication issue.

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

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

5.4 Summary

GHC's simplifier is the most complex of the passes that are applied to the intermediate language. The simplifer 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 analyis, or let-floating [JS98]. Because it is used so much, the simplifier has to be pretty efficient.

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 our standard 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 (in which case its definition must not be discarded)
- `x` is not a loop breaker
- `E'` is *trivial* – that is, a literal, variable, or type application. 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 pre-computed 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.

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*.

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! An in-expression is meaningless without the substitution appropriate to the original site of the expression. The substitution plays the role of the *lexical environment* of an expression: it gives meaning to the free variables of the expression, and an unevaluated expression (or in-expression, in our present context) must always be paired with its lexical environment.

Thus we are led to the following definition for the substitution:

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

A `SuspEx` (`Susp` for "suspended") is used by PreInlineUnconditionally, and pairs an in-expression with the substitution appropriate to its `let` binding. A `DoneEx` is used by the name-cloning mechanism and by PostInlineUnconditionally.

The `DoneTy` case just like `DoneEx`, except that it is used for types.

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 that is needed 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 occurence sites.

`NotAmong` is described next.

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 [JS98].

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.

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 lifetime. The nice thing is that this dynamic knowledge could elegantly be carried by the in-scope set, which we needed 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 now discuss.

7 Occurrences

We are finally ready to discuss what the simplifier does at at the occurrence of a variable. There are two stages. First, we look up the variable in the substitution (Section 7.1). Then we decide 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 in the substitution at all – for example, it might be a lambda-bound variable that did not need to be renamed. In that case, the next thing to do is to consider inlining it. The subsitution 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 subsitution! Consider this example:

```
\x -> let
```

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

The `DoneEx` contains an out-expression for `f`'s right-hand side, that mentions `x`. But the substitution in force at `f`'s occurrences will map `x` to a fresh variable, `x1` say, since the inner `\x` uses the same name as the outer one. If we do decide to inline `f`, we must not use this substitution when dealing with its body. The right thing to do is to use 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 was a 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 "inlne: OnceSafe"
inline rhs OnceUnsafe cont
 = whnfOrBot rhs && not (veryBoring cont)
inline rhs MultiSafe cont
 = inlineMulti rhs cont
inline rhs MultiUnsafe cont
 = 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 5.3), to ensure that inlining will not happen if there is any work duplication. However, as noted in Section 6, even if the variable occurs just once, it is not always a good idea to inline it. The `veryBoring` predictate therefore examines the context, returning `False` if there is anything at all interesting about it:

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

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; `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. 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 (Figure 1). 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`.

7.4 Inlining functions

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. Just 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
  | ApplyTo InExpr Subst Context
  | Select  InVar [InAlt] Subst Context
  | ArgOf   (OutExpr -> OutExpr)
  | Inline  Context
```

The `Stop` context is used when beginning simplification of a lazy function argument, or the right hand side of a `let` binding. The `ApplyTo` 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. `simplExpr` can deal with applications rather easily:

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

The `Select` context is used when simplifying the scrutinee of a `case` expression. Not only is it easy to ask whether the current expression is the scrutinee of a `case`, but also the case-of-known-constructor transformation:

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

(and similar) is also very easy: `simplExpr` just matches a constructor application with a `Select` continuation.

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

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. The `Inline` context is discussed in next.

7.6 INLINE pragmas

Like some other languages, GHC allows the programmer to specify that a function should be inlined, as a *pragam* in the Haskell source langauge:

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

Uniquely (we believe) 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.

Both these pragmas are translated to constructors in the `Expr` data type:

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

Their effect is quite different, of course:

- 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 add push an `InlinePlease` onto the context stack. The `smallEnough` predicate returns `True` if it finds such a context.

There is an important subtlety, however. Consider

```
g = \a b -> ...big...
f = InlineMe (\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.

8 Related work

There is a modest literature on inlining applied to imperative programming languages, such as C and FORTRAN ([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.

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 25-25% 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! On the other hand, Serrano found only a 15% benefit for the Bigloo Scheme compiler [Ser97]. These compilers are all capable of dealing with, and inlining, *nested* function definitions.

All of these papers pay primary attention to various heuristics for trading off code size against the benefit of inlining functions that are called many times, whereas that (important) issue has played only a small role in this paper (Section 7.4).

A quite separate and unique 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.

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

Julian Seward (so far!).

Bibliography

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.

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

[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, pages 90–106, Amsterdam, June 1997. ACM.

[JS98] SL Peyton Jones and A Santos. A transformation-based optimiser for haskell. *Science of Computer Programming*, 32(1-3):3–47, September 1998.

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

[San95] A Santos. *Compilation by transformation in non-strict functional languages.* Ph.D. thesis, Department of Computing Science, Glasgow University, September 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 *ACM Sigplan International Conference on Functional Programming (ICFP'98)*, pages 313–323, Balitmore, September 1998. ACM.

[WJ99] 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.