A modular fully-lazy lambda lifter in Haskell

Simon L Peyton Jones
Department of Computing Science, University of Glasgow, G12 8QQ

David Lester
Department of Computer Science, University of Manchester, M13 9PL

June 15, 1995

Abstract

An important step in many compilers for functional languages is *lambda lifting*. In his thesis, Hughes showed that by doing lambda lifting in a particular way, a useful property called *full laziness* can be preserved (Hughes [1983]). Full laziness has been seen as intertwined with lambda lifting ever since.

We show that, on the contrary, full laziness can be regarded as a completely separate process to lambda lifting, thus making it easy to use different lambda lifters following a full-laziness transformation, or to use the full-laziness transformation in compilers which do not require lambda lifting.

On the way, we present the complete code for our modular fully-lazy lambda lifter, written in the Haskell functional programming language.

This paper appears in Software Practice and Experience 21(5), May 1991.

1 Introduction

Lambda lifting and full laziness are part of the folk lore of functional programming, yet the few published descriptions of fully-lazy lambda lifting have been either informal or hard to understand, with the notable exception of Bird (Bird [1987]). Our treatment differs from earlier work in the following ways:

- The main technical contribution is to show how full laziness can be separated from lambda lifting, so that the two transformations can be carried out independently. This makes each transformation easier to understand and improves the modularity of the compiler.
- Our treatment deals with a language including let- and letrec-expressions. Not only is this essential for efficient compilation in the later stages of most compilers, but we also show that eliminating let(rec) expressions, by translating them into lambda abstractions, loses full laziness. To our knowledge, this has not previously been realised.
- We show how to decompose each of the transformations further into a composition of simple steps, each of which is very easy to program, thus further improving modularity.

- We have developed an elegant use of parameterised data types, which allows the type system to help express the purpose of each pass.

We present the complete source code for our solution, and we document some of the experience we gained in developing it; indeed the paper can be read as an exercise in functional programming.

The code is presented in the functional programming language Haskell, and the source text for the paper is an executable literate Haskell program. Lines of code are distinguished by a leading `>` sign, the rest of the text being commentary.

We introduce all the notation and background that is required to understand the paper. The first three sections introduce the language to be compiled, the Haskell language, and the main data types to be used. Following these preliminaries we develop a non-fully-lazy lambda lifter and then refine it into a fully lazy one. The paper concludes with some optimisations to the fully-lazy lambda lifter.

2 The language

We begin by defining a small language, $\mathcal{L}$, on which our compiler will operate. The abstract syntax of the language is given by the following productions:

expression	::=	*name*	
	|	*constant*	Literals and built-in functions
	|	$expression_1$ $expression_2$	Application
	|	`let` *defns* `in` *expression*	Non-recursive definitions
	|	`letrec` *defns* `in` *expression*	Recursive definitions
	|	λ *name*`->`*expression*	Lambda abstraction
defns	::=	$def_1 \ldots def_n$ $(n > 0)$	
def	::=	*name* `=` *expression*	

The concrete syntax is conventional: parentheses are used to disambiguate; application associates to the left and binds more tightly than any other operator; the body of a lambda abstraction extends as far to the right as possible; the usual infix operators are permitted; and definitions are delimited using layout. Notice that the bindings in let(rec) expressions are all *simple*; that is, the left hand side of the binding is always just a variable. (Functions are defined by binding variables to lambda abstractions.) Here is an example to illustrate these points:

```
letrec
        fac = \n -> if (n == 0) 1 (n * factorial (n-1))
in
fac 100
```

An important feature of this language is that any pure high-level functional programming

language can be translated into it without loss of expressiveness or efficiency. This process is described in some detail by Peyton Jones (Peyton Jones [1987]).

Notice that let(rec)-expressions are retained, despite the fact that they are responsible for many of the subtleties in the rest of the paper. They can certainly be transformed into applications of lambda abstractions and the `Y` combinator, but doing so straightforwardly may result in a serious loss of efficiency (Peyton Jones [1987, Chapter 14]). For example, eliminating a let-expression introduces a new lambda abstraction, which will (under the compilation scheme described in this paper) be lambda-lifted, and thereby decompose execution into smaller steps. Similarly, a naive treatment of mutual local recursion using `Y` involves much packing and unpacking of tuples, which can easily be avoided if the letrec-expression is handled explicitly. Finally, later on we show that laziness can be lost if let-expressions are not handled specially.

3 Haskell

Haskell is a recently-designed common non-strict pure functional language (Hudak et al. [1992]). For the purposes of this paper it is fairly similar to SML or Miranda[1], except in its treatment of abstract data types and modules. We make little use of Haskell's major technical innovation, namely systematic overloading using type classes.

Haskell modules begin with a declaration naming the module, and importing any modules it requires:

```
> module LambdaLift where
>
> import Utilities
```

Here, the module we will be defining is called `LambdaLift`, and it imports an auxiliary module called `Utilities`, whose interface is given in the Appendix.

4 A data type for compilation

Every compiler has a data type which represents the abstract syntax tree of the program being compiled. The definition of this data type has a substantial impact on the clarity and efficiency of the compiler, so it merits careful thought.

4.1 First failed attempt

As a first attempt, the language described in the previous section can translated directly into the following Haskell *algebraic data type* declaration[2]:

[1] Miranda is a trade mark of Research Software Ltd

[2] Notice that data constructors, such as `EVar` and `ELam`, and type constructors, such as `Expresssion` and `Name`, must both begin with capital letter in Haskell.

```
> data Expression
>      = EConst Constant
>      | EVar Name
>      | EAp Expression Expression
>      | ELam [Name] Expression
>      | ELet IsRec [Definition] Expression
```

We choose to represent names simply by their character string, using a *type synonym* declaration.

```
> type Name = [Char]
```

Constants may be numbers, booleans, the names of global functions, and so on:

```
> data Constant = CNum Integer | CBool Bool | CFun Name
```

The definition of the `Constant` type can be changed without affecting any of the rest of this paper. As an example of the `Expression` type, the $\mathcal{L}$-expression `a+3` would be represented by the Haskell expression

```
EAp (EAp (EConst (CFun "+")) (EVar "a")) (EConst (CNum 3))
```

Let-expressions can usually be treated in the same manner as letrec-expressions, so the two are given a common constructor, and distinguished by a flag of type `IsRec`. It is convenient to use a boolean for this flag:

```
> type IsRec = Bool
> recursive = True
> nonRecursive = False
```

Each definition in the definition list of an `ELet` construction is just a pair:

```
> type Definition = (Name, Expression)
```

4.2 Second failed attempt

It does not take long to discover that this is an insufficiently flexible data type. Many compiler passes add information to the abstract syntax tree, and we need a systematic way to represent this information. Examples of analyses which generate this sort of information are: free-variable analysis, binding level analysis, type inference, strictness analysis, and sharing analysis.

The most obvious way to add such information is to add a new constructor for annotations to the `Expr` data type, thus:

```
> data Expression
>      = EVar Name
>      | ...
>      | EAnnot Annotation Expression
```

together with an auxiliary data type for annotations, which can be extended as required[3]:

[3] The type `Set a` is a standard abstract data type for sets, whose interface is given in the Appendix, but

```
> data Annotation = FreeVars (Set Name)
>                 | Level Integer
```

This allows annotations to appear freely throughout the syntax tree, which appears admirably flexible. In practice, it suffers from two major disadvantages

- It is easy enough to *add* annotation information in the form just described, but writing a compiler pass which *uses* information placed there by a previous pass is downright awkward. Suppose, for example, that a pass wishes to use the free-variable information left at every node of the tree by a previous pass. Presumably this information is attached immediately above every node, but the data type would permit several annotation nodes to appear above each node, and worse still none (or more than one) might have a free-variable annotation.

  Even if the programmer is prepared to certify that there is exactly one annotation node above every tree node, and that it is a free-variable annotation, the implementation will still perform pattern-matching to check these assertions when extracting the annotation.

  Both of these problems, namely the requirement for uncheckable programmer assertions, and some implementation overhead, are directly attributable to the fact that every annotated tree has the rather uninformative type `Expression`, which says nothing about which annotations are present.

- The second major problem is that further experimentation reveals that *two* distinct forms of annotation are required. The first annotates expressions as above, but the second annotates the binding occurrences of variables; that is, the occurrences on the left-hand sides of definitions, and the bound variable in a lambda abstraction. We will call these occurrences *binders*. An example of the need for the latter comes in type inference, where it the compiler infers a type for each binder, as well as for each sub-expression.

  It is possible to use the expression-annotation to annotate binders, but it is clumsy and inconvenient to do so.

4.3 A happy ending

We will address the second problem first, since it has an easy solution. All we need do is parameterise the `Expression` type with respect to the type of binders, thus:

```
> data Expr binder
>       = EConst Constant
>       | EVar Name
>       | EAp (Expr binder) (Expr binder)
>       | ELam [binder] (Expr binder)
>       | ELet IsRec [Defn binder] (Expr binder)
>
> type Defn binder = (binder, Expr binder)
```

whose implementation is not further defined.

`binder` is a type variable, which in Haskell must begin with a lower-case letter. We can easily recover a definition of our original `Expression` type, in which a binder is represented by a `Name`, using a type synonym declaration, thus:

```
> type Expression = Expr Name
```

Alternatively, a data type in which binders are names annotated with a type can be defined thus:

```
> type TypedExpression = Expr (Name, TypeExpr)
```

where `TypeExpr` is a data type representing type expressions.

Returning to annotations on expressions, we can re-use the same technique by parameterising the data type of expressions with repect to the annotation type. We want to have an annotation on every node of the tree, so one possibility would be to add an extra field to every constructor with the annotation information. This is inconvenient if, for example, you simply want to extract the free-variable information at the top of a given expression without performing case analysis on the root node. This leads to the following idea: each level of the tree is a pair, whose first component is the annotation, and whose second component is the abstract syntax tree node. Here are the corresponding Haskell data type definitions:

```
> type AnnExpr binder annot = (annot, AnnExpr' binder annot)
>
> data AnnExpr' binder annot
>        = AConst Constant
>        | AVar Name
>        | AAp (AnnExpr binder annot) (AnnExpr binder annot)
>        | ALam [binder] (AnnExpr binder annot)
>        | ALet IsRec [AnnDefn binder annot] (AnnExpr binder annot)
>
> type AnnDefn binder annot = (binder, AnnExpr binder annot)
```

Notice the way that the mutual recursion between `AnnExpr` and `AnnExpr'` ensures that every node in the tree carries an annotation. The sort of annotations carried by an expression are now manifested in the type of the expression. For example, an expression annotated with free variables has type `AnnExpr Name (Set Name)`.

It is a real annoyance that `AnnExpr'` and `Expr` have to define two essentially identical sets of constructors. There appears to be no way around this within the Hindley-Milner type system. It would be possible to abandon the `Expr` type altogether, because the `Expr a` is nearly isomorphic to `AnnExpr a ()`, but there are two reasons why we chose not to do this. Firstly, the two types are not quite isomorphic, because the latter distinguishes `((), ⊥ )` from ⊥ while the former does not. Secondly (and more seriously), it is very tiresome to write all the `()`'s when building and pattern-matching on trees of type `AnnExpr a ()`.

Finally, we define two useful functions, `bindersOf` and `rhssOf` (right-hand-sides of), which each take the list of definitions in an `ELet`, and pick out the list of variables bound by the let(rec), and list of right-hand sides to which they are bound, respectively:

```
> bindersOf        :: [(binder,rhs)] -> [binder]
```

```
> bindersOf defns =  [name | (name, rhs) <- defns]
>
> rhssOf         :: [(binder,rhs)] -> [rhs]
> rhssOf defns   =  [rhs | (name,rhs) <- defns]
```

This definition illustrates the use of a *type signature* to express the type of the function to be defined; and of a *list comprehension* in the right hand side of `bindersOf`, which may be read "the list of all `name`s, where the pair `(name,rhs)` is drawn from the list `defns`". Both of these are now conventional features of functional programming languages.

This completes our development of the central data type. The discussion has revealed some of the strengths, and a weakness, of the algebraic data types provided by all modern functional programming languages.

5 Lambda lifting

Any implementation of a lexically-scoped programming language has to cope with the fact that a function may have free variables. Unless these are removed in some way, an environment-based implementation has to manipulate linked environment frames, and a reduction-based system is made significantly more complex by the need to perform α-renaming during substitution. A popular way of avoiding these problems, especially in graph reduction implementations, is to eliminate all free variables from function definitions by means of a transformation known as *lambda lifting*. Lambda lifting is a term coined by Johnsson (Johnsson [1985]), but the transformation was independently developed by Hughes (Hughes [1983]). A tutorial treatment is given by Peyton Jones (Peyton Jones [1987]).

The lambda-lifting issue is not restricted to functional languages. For example, Pascal allows a function to be declared locally within another function, and the inner function may have free variables bound by the outer scope. On the other hand, the C language does not permit such local definitions. In the absence of side effects, it is simple to make a local function definition into a global one: all we need do is add the free variables as extra parameters, and add these parameters to every call. This is exactly what lambda lifting does.

In a functional-language context, lambda lifting transforms an *expression* into a *set of supercombinator definitions*, each of which defines a function of zero or more arguments, and whose body contains no embedded lambda abstractions. In this paper we will use the convention that the value of the set of definitions is the value of the supercombinator `$main`; this artifice avoids the need to speak of "a set of supercombinator definitions together with an expression to be evaluated".

To take a simple example, consider the program

```
let
    f = \x -> let g = \y -> x*x + y in (g 3 + g 4)
in
f 6
```

Here, `x` is free in the abstraction `\y -> x*x + y`. The free variable can be removed by defining a new function `$g` which takes `x` as an extra parameter, but whose body is the offending

abstraction, and redefining g in terms of $g, giving the following set of supercombinator definitions:

```
$g x y = x*x + y
f x = let g = $g x in (g 3 + g 4)
$main = f 6
```

(To stress the fact that this program is a set of supercombinator definitions, we permit ourselves to write the arguments of the supercombinator on the left of the = sign.)

Matters are no more complicated when recursion is involved. Suppose that g was recursive, thus:

```
let
    f = \x -> letrec g = \y -> cons (x*y) (g y) in g 3
in
f 6
```

Now x and g are both free in the \y-abstraction, so the lambda lifter will produce the following set of supercombinators:

```
$g g x y = cons (x*y) (g y)
f x = letrec g = $g g x in g 3
$main = f 6
```

Notice that the definition of g is still recursive, but the lambda lifter has eliminated the local lambda abstraction. The program is now directly implementable by most compiler back ends.

This is not the only way to handle local recursive function definitions. The main alternative is described by Johnsson (Johnsson [1987]), who generates directly-recursive supercombinators from locally-recursive function definitions; in our example, $g would be directly recursive rather than calling its parameter g, thus:

```
$g x y = cons (x*y) ($g x y)
f x = $g x 3
$main = f 6
```

The lambda lifter he describes is much more complex than the one we develop here. For Johnsson it is worth the extra work, because the back end of his compiler can produce better code from directly-recursive supercombinators, but it is not clear that this applies universally to all implementations. At all events, we will stick to the simple lambda lifter here, since our main concern is the interaction with full laziness.

5.1 Implementing a simple lambda lifter

We are now ready to develop a simple lambda lifter. It will take an expression and deliver a list of supercombinator definitions, so this is its type:

```
> lambdaLift :: Expression -> [SCDefn]
```

Each supercombinator definition consists of the name of the supercombinator, the list of its arguments, and its body:

```
> type SCDefn = (Name, [Name], Expression)
```

It should be the case that there are no `ELam` constructors anywhere in the third component of the triple. Unfortunately, there is no way to express (and hence enforce) this constraint in the type, except by declaring yet another new variant of `Expr` lacking such a constructor. This is really another shortcoming of the type system: there is no means of expressing this sort of subtyping relationship.

The lambda lifter works in three passes:

- First, we annotate every node in the expression with its free variables. This is used by the following pass to decide which extra parameters to add to a lambda abstraction. The `freeVars` function has type

  ```
  > freeVars :: Expression -> AnnExpr Name (Set Name)
  ```

- Second, the function `abstract` abstracts the free variables from each lambda abstraction, replacing the lambda abstraction with the application of the new abstraction (now a supercombinator) to the free variables. For example, the lambda abstraction

  ```
  (\x -> y*x + y*z)
  ```

  would be transformed to

  ```
  (\y -> \z -> \x ->  y*x + y*z) y z
  ```

  `abstract` has the type signature:

  ```
  > abstract :: AnnExpr Name (Set Name) -> Expression
  ```

  Notice, from the type signature, that `abstract` removes the free variable information, which is no longer required.

- Finally, `collectSCs` gives a unique name to each supercombinator, collects all the supercombinator definitions into a single list, and introduces the `$main` supercombinator definition.

  ```
  > collectSCs :: Expression -> [SCDefn]
  ```

The compiler itself is the composition of these three functions[4]:

```
> lambdaLift = collectSCs . abstract . freeVars
```

It would of course be possible to do all the work in a single pass, but the modularity provided by separating them has a number of advantages: each individual pass is easier to understand, the passes may be reusable (for example, we reuse `freeVars` below), and modularity makes it easier to change the algorithm somewhat.

[4] Infix "." denotes function composition.

As an example of the final point, the Haskell compiler under development at Glasgow will be able to generate better code by omitting the `collectSCs` pass, because more is then known about the context in which the supercombinator is applied. For example, consider the expression, which might be produced by the `abstract` pass:

```
let f = (\v.\x. v-x) v
in ...f...f...
```

Here `abstract` has removed `v` as a free variable from the `\x`-abstraction. Rather than compiling the supercombinator independently of its context, our compiler constructs a closure for `f`, whose code accesses `v` directly from the closure and `x` from the stack. The calls to `f` thus do not have to move `v` onto the stack. The more free variables there are the more beneficial this becomes. Nor do the calls to `f` become less efficient because the definition is a local one; the compiler can see the binding for `f` and can jump directly to its code.

5.2 Free variables

We begin by giving the code for the free-variable pass, not because it is particularly subtle, but because it serves to illustrate HASKELL language notation.

```
> freeVars (EConst k) = (setEmpty, AConst k)
> freeVars (EVar v)   = (setSingleton v, AVar v)

> freeVars (EAp e1 e2) =
>   (setUnion (freeVarsOf e1') (freeVarsOf e2'), AAp e1' e2')
>   where
>   e1' = freeVars e1
>   e2' = freeVars e2

> freeVars (ELam args body) =
>   (setDifference (freeVarsOf body') (setFromList args), ALam args body')
>   where
>   body' = freeVars body

> freeVars (ELet isRec defns body) =
>   (setUnion defnsFree bodyFree, ALet isRec (zip binders rhss') body')
>   where
>   binders = bindersOf defns
>   binderSet = setFromList binders
>   rhss' = map freeVars (rhssOf defns)
>   freeInRhss = setUnionList (map freeVarsOf rhss')
>   defnsFree | isRec     = setDifference freeInRhss binderSet
>             | not isRec = freeInRhss
>   body' = freeVars body
>   bodyFree = setDifference (freeVarsOf body') binderSet

> freeVarsOf :: AnnExpr Name (Set Name) -> Set Name
```

```
> freeVarsOf (free_vars, expr) = free_vars
```

In the definition of `defnsFree`, the boolean condition between the vertical bar and the equals sign is a *guard*, which serves to select the appropriate right-hand side. The function `zip` in the definition of `defns'` is a standard function which takes two lists and returns a list consisting of pairs of corresponding elements of the argument lists. The set operations `setUnion`, `setDifference`, and so on, are defined in the utilities module, whose interface is given in the Appendix. Otherwise the code should be self-explanatory.

5.3 Generating supercombinators

The next pass merely replaces each lambda abstraction, which is now annotated with its free variables, with a new abstraction (the supercombinator) applied to its free variables. The full definition is given in the Appendix; and the only interesting equation is that dealing with lambda abstractions:

```
> abstract (free, ALam args body) =
>   foldl EAp sc (map EVar fvList)
>   where
>   fvList = setToList free
>   sc = ELam (fvList ++ args) (abstract body)
```

The function `foldl` is a standard function; given a dyadic function $\oplus$, a value b, and a list $xs = [x_1, ..., x_n]$, `foldl` $\oplus$ b xs computes $(\ldots((b \oplus x_1) \oplus x_2) \oplus \ldots x_n)$. Notice the way that the free-variable information is discarded by the pass, since it is no longer required.

We also observe that `abstract` treats the two expressions `ELam args1 (ELam args2 body))` and `(ELam (args1++args2) body)` differently. In the former case, the two abstractions will be treated separately, generating two supercombinators, while in the latter only one supercombinator is produced. It is clearly advantageous to merge directly-nested `ELam`s before performing lambda lifting. This is equivalent to the η-abstraction optimisation noted by Hughes (Hughes [1983]).

5.4 Collecting supercombinators

Finally, we have to name the supercombinators and collect them together. To generate new names, the main function has to carry around a *name supply*; in particular, it must take the name supply as an argument and return a depleted supply as a result. In addition, it must return the collection of supercombinators it has found, and the transformed expression. Because of these auxiliary arguments and results, we define a function `collectSCs_e` which does all the work, with `collectSCs` being defined in terms of it:

```
> collectSCs_e :: NameSupply -> Expression
>              -> (NameSupply, Bag SCDefn, Expression)

> collectSCs e = [("$main", [], e')] ++ bagToList scs
>                where
>                (_, scs, e') = collectSCs_e initialNameSupply e
```

The name supply is represented by an abstract data type `NameSupply`, whose interface is given in the Appendix, but in which we take no further interest. The collection of supercombinators is a *bag*, represented by the abstract data type of `Bag`, which has similar operations defined on it as those for `Set`.

The "_" in the last line of `collectSCs` is a *wildcard* which signals the fact that we are not interested in the depleted name supply resulting from transforming the whole program. The code is now easy to write.

```
> collectSCs_e ns (EConst k) = (ns, bagEmpty, EConst k)
> collectSCs_e ns (EVar v)   = (ns, bagEmpty, EVar v)
> collectSCs_e ns (EAp e1 e2) =
>   (ns2, bagUnion scs1 scs2, EAp e1' e2')
>   where
>   (ns1, scs1, e1') = collectSCs_e ns  e1
>   (ns2, scs2, e2') = collectSCs_e ns1 e2
```

In the case of lambda abstractions we replace the abstraction by a name, and add a supercombinator to the result.

```
> collectSCs_e ns (ELam args body) =
>   (ns2, bagInsert (name, args, body') bodySCs, EConst (CFun name))
>   where
>   (ns1, bodySCs, body') = collectSCs_e ns body
>   (ns2, name) = newName ns1 "SC"
```

A common paradigm occurs in the case for let(rec):

```
> collectSCs_e ns (ELet isRec defns body) =
>   (ns2, scs, ELet isRec defns' body')
>   where
>   (ns1, bodySCs, body') = collectSCs_e ns body
>   ((ns2, scs), defns') = mapAccuml collectSCs_d (ns1, bodySCs) defns
>
>   collectSCs_d (ns, scs) (name, rhs) =
>     ((ns1, bagUnion scs scs'), (name, rhs'))
>     where
>     (ns1, scs', rhs') = collectSCs_e ns rhs
```

When processing a list of definitions, we need to generate a new list of definitions, threading the name supply through each. This is done by a new higher-order function `mapAccuml`, which behaves like a combination of `map` and `foldl`; it applies a function to each element of a list, passing an accumulating parameter from left to right, and returning a final value of this accumulator together with the new list. `mapAccuml` is defined in the Appendix.

This completes the definition of the simple lambda lifter. We now turn our attention to full laziness.

6 Separating full laziness from lambda lifting

Previous accounts of full laziness have invariably linked it to lambda lifting, by describing "fully-lazy lambda lifting", which turns out to be rather a complex process. Hughes gives an algorithm but it is extremely subtle and does not handle let(rec) expressions (Hughes [1983]). On the other hand, Peyton Jones does cover let(rec) expressions, but the description is only informal and no algorithm is given (Peyton Jones [1987]).

In this section we show how full laziness and lambda lifting can be cleanly separated. This is done by means of a transformation involving let-expressions. Lest it be supposed that we have simplified things in one way only by complicating them in another, we also show that performing fully-lazy lambda lifting without let(rec) expressions risks an unexpected loss of laziness. Furthermore, much more efficient code can be generated for let(rec) expressions in later phases of most compilers than for their equivalent lambda expressions.

6.1 A review of full laziness

We begin by briefly reviewing the concept of full laziness. Consider again the example given earlier.

```
let
    f = \x -> let g = \y -> x*x + y in (g 3 + g 4)
in
f 6
```

The simple lambda lifter generates the program:

```
$g x y = x*x + y
f x = let g = $g x in (g 3 + g 4)
$main = f 6
```

In the body of `f` there are two calls to `g` and hence to `$g`. But `($g x)` is not a reducible expression, so `x*x` will be computed twice. But `x` is fixed in the body of `f`, so some work is being duplicated. It would be better to share the calculation of `x*x` between the two calls to `$g`. This can be achieved as follows: instead of making `x` a parameter to `$g`, we make `x*x` into a parameter, like this:

```
$g p y = p + y
f x = let g = $g (x*x) in (g 3 + g 4)
```

(we omit the definition of `$main` from now on, since it does not change). So a fully-lazy lambda lifter will make each *maximal free sub-expresssion* (rather than each *free variable*) of a lambda abstraction into an argument of the corresponding supercombinator. A maximal free expression (or MFE) of a lambda abstraction is an expression which contains no occurrences of the variable bound by the abstraction, and is not a sub-expression of a larger expression with this property.

Full laziness corresponds precisely to moving a loop-invariant expression outside the loop, so that it is computed just once at the beginning rather than once for each loop iteration.

How important is full laziness for "real" programs? No serious studies have yet been made of this question, though we plan to do so. However, recent work by Holst suggests that the importance of full laziness may be greater than might at first be supposed (Holst [1991]). He shows how to perform a transformation which automatically enhances the effect of full laziness, to the point where the optimisations obtained compare favourably with those gained by partial evaluation (Jones, Sestoft & Søndergaard [1989]), though with much less effort.

6.2 Full-lazy lambda lifting in the presence of let(rec)s

Writing a fully-lazy lambda lifter, as outlined in the previous section, is somewhat subtle. Our language, which includes let(rec) expressions, appears to make this worse by introducing a new language construct. For example, suppose the definition of `g` in our running example was slightly more complex, thus:

```
g = \y -> let z = x*x
          in let p = z*z
          in p + y
```

Now, the sub-expression `x*x` is a MFE of the `\y`-abstraction, but sub-expression `z*z` is not since `z` is bound inside the `\y`-abstraction. Yet it is clear that `p` depends only on `x` (albeit indirectly), and so we should ensure that `z*z` should only be computed once.

Does a fully-lazy lambda lifter spot this if let-expressions are coded as lambda applications? No, it does not. The definition of `g` would become

```
g = \y -> (\z -> (\p -> p+y) (z*z)) (x*x)
```

Now, `x*x` is free as before, but `z*z` is not. In other words, *if the compiler does not treat let(rec)-expressions specially, it may lose full laziness which the programmer might reasonably expect to be preserved.*

Fortunately, there is a straightforward way to handle let(rec)-expressions, as described by Peyton Jones, namely to "float" each let(rec) definition outward until it is outside any lambda abstraction in which it is free (Peyton Jones [1987, Chapter 15]). For example, all we need do is transform the definition of `g` to the following:

```
g = let z = x*x
    in let p = z*z
    in \y -> p + y
```

Now `x*x` and `z*z` will each be computed only once. Notice that this property should hold *for any implementation of the language*, not merely for one based on lambda lifting and graph reduction. This is a clue that full laziness and lambda lifting are not as closely related as at first appears, a topic to which we will return in the next section.

Meanwhile, how can we decide how far out to float a definition? It is most easily done by using *lexical level numbers* (or *de Bruijn numbers*). There are three steps:

- First, assign to each lambda-bound variable a level number, which says how many lambdas enclose it. Thus in our example, `x` would be assigned level number 1, and `y`

level number 2.

- Now, assign a level number to each let(rec)-bound variable (outermost first), which is the maximum of the level numbers of its free variables, or zero if there are none. In our example, both `p` and `z` would be assigned level number 1. Some care needs to be taken to handle letrecs correctly.

- Finally, float each definition (whose binder has level n, say) outward, until it is outside the lambda abstraction whose binder has level $n + 1$, but still inside the level-n abstraction. There is some freedom in this step about exactly where between the two the definition should be placed.

Each mutually-recursive set of definitions defined in a letrec should be floated out together, because they depend on each other and must remain in a single letrec. If, in fact, the definitions are *not* mutually recursive despite appearing in the same letrec, this policy might lose laziness by retaining in an inner scope a definition which could otherwise be floated further outwards. The standard solution is to perform *dependency analysis* on the definitions in each letrec expression, to break each group of definitions into its minimal subgroups. We take no further interest in this optimisation, which is discussed in Chapter 6 of Peyton Jones (Peyton Jones [1987]).

Finally, a renaming pass should be carried out before the floating operation, so that there is no risk that the bindings will be altered by the movement of the let(rec) definitions. For example, the expression

```
\y -> let y = x*x in y
```

is obviously not equivalent to

```
let y = x*x in \y->y
```

All that is required is to give every binder a unique name to eliminate the name clash.

6.3 Full laziness without lambda lifting

At first it appears that the requirement to float let(rec)s outward in order to preserve full laziness merely further complicates the already subtle fully lazy lambda lifting algorithm suggested by Hughes. However, a simple transformation allows *all* the full laziness to be achieved by let(rec) floating, while lambda lifting is performed by the original simple lambda lifter.

The transformation is this: *before floating let(rec) definitions, replace each MFE e with the expression* `let v = `e` in v`. This transformation both gives a name to the MFE and makes it accessible to the let(rec) floating transformation, which can now float out the new definitions. Ordinary lambda lifting can then be performed. For example, consider the original definition of `g`:

```
let
    f = \x -> let g = \y -> x*x + y
              in (g 3 + g 4)
```

```
in
f 6
```

The subexpression `x*x` is an MFE, so it is replaced by a trivial let-expression:

```
let
    f = \x -> let g = \y -> (let v = x*x in v) + y
              in (g 3 + g 4)
in
f 6
```

Now the let-expression is floated outward:

```
let
    f = \x -> let g = let v = x*x in \y -> v + y
              in (g 3 + g 4)
in
f 6
```

Finally, ordinary lambda lifting will discover that `v` is free in the `\y`-expression, and the resulting program becomes:

```
$g v y = v + y
f x = let g = let v = x*x in $g v
      in (g 3 + g 4)
$main = f 6
```

A few points should be noted here. Firstly, the original definition of a maximal free expression was relative to a *particular* lambda abstraction. The new algorithm we have just developed transforms certain expressions into trivial let-expresssions. Which expressions are so transformed? Just the ones which are MFEs of *any* enclosing lambda abstraction. For example, in the expression:

```
\y -> \z -> (y + (x*x)) / z
```

two MFEs are identified: `(x*x)`, since it is an MFE of the `\y`-abstraction, and `(y + (x*x))`, since it is an MFE of the `\z`-abstraction. After introducing the trivial let-bindings, the expression becomes

```
\y -> \z -> (let v1 = y + (let v2 = x*x in v2) in v1) / z
```

Secondly, the newly-introduced variable `v` must either be unique, or the expression must be uniquely renamed after the MFE-identification pass.

Thirdly, in the final form of the program `v` is only referenced once, so it would be sensible to replace the reference by the right-hand-side of the definition and eliminate the definition, yielding exactly the program we obtained using Hughes's algorithm. This is a straightforward transformation, and we will not discuss it further here, except to note that this property will hold for all let-definitions which are floated out past a lambda. In any case, many compiler back ends will generate the same code regardless of whether or not the transformation is performed.

6.4 A fully lazy lambda lifter

Now we are ready to define the fully lazy lambda lifter. It can be decomposed into the following stages:

- First we must make sure that all `ELam` constructors bind only a single variable, because the fully-lazy lambda lifter must treat each lambda individually. It would be possible to encode this in later phases of the algorithm, by dealing with a list of arguments, but it turns out that we can express an important optimisation by altering this pass alone.

  ```
  > separateLams :: Expression -> Expression
  ```

- First we annotate all binders and expressions with level numbers, which we represent by natural numbers starting with zero:

  ```
  > type Level = Int
  > addLevels :: Expression -> AnnExpr (Name, Level) Level
  ```

- Next we identify all MFEs, by replacing them with trivial let-expressions. Level numbers are no longer required on every sub-expression, only on binders.

  ```
  > identifyMFEs :: AnnExpr (Name, Level) Level -> Expr (Name, Level)
  ```

- A renaming pass makes all binders unique, so that floating does not cause name-capture errors. This must be done after `identifyMFEs`, which introduces new bindings.

  ```
  > rename :: Expr (Name, a) -> Expr (Name, a)
  ```

- Now the let(rec) definitions can be floated outwards. The level numbers are not required any further.

  ```
  > float :: Expr (Name, Level) -> Expression
  ```

- Finally, ordinary lambda lifting can be carried out, using `lambdaLift` from the section dealing with simple lambda lifting.

The fully lazy lambda lifter is just the composition of these passes:

```
> fullyLazyLift = lambdaLift . float . rename .
>                 identifyMFEs . addLevels . separateLams
```

6.5 Separating the lambdas

The first pass, which separates lambdas so that each `ELam` only binds a single argument, is completely straightforward. The only interesting equation is that which handles abstractions, which we give here:

```
> separateLams (ELam args body) = foldr mkLam (separateLams body) args
>                                 where
>                                 mkLam arg body = ELam [arg] body
```

The other equations are given in the Appendix.

6.6 Adding level numbers

There are a couple of complications concerning annotating an expression with level-numbers.

At first it looks as though it is sufficient to write a function which returns an expresssion annotated with level numbers; then for an application, for example, one simply takes the maximum of the levels of the two sub-expressions. Unfortunately, this approach loses too much information, because there is no way of mapping the level-number of the *body* of a lambda abstraction to the level number of the abstraction *itself*. The easiest solution is first to annotate the expression with its free variables, and then use a mapping `freeSetToLevel` from variables to level-numbers, to convert the free-variable annotations to level numbers.

```
> freeSetToLevel :: Set Name -> Assn Name Level -> Level
> freeSetToLevel free_vars env =
>       maximum (0 : map (assLookup env) (setToList free_vars))
>       -- If there are no free variables, return level zero
```

The second complication concerns letrec expressions. What is the correct level number to attribute to the newly-introduced variables? The right thing to do is to take the maximum of the levels of the free variables of all the right-hand sides *without* the recursive variables, or equivalently map the recursive variables to level zero when taking this maximum. This level should be attributed to each of the new variables. Let-expressions are much simpler: just attribute to each new variable the level number of its right-hand side.

Now we are ready to define `addLevels`. It is the composition of two passes, the first of which annotates the expression with its free variables, while the second uses this information to generate level-number annotations.

```
> addLevels = freeToLevel . freeVars
```

We have defined the `freeVars` function already, so it remains to define `freeToLevel`. The main function will need to carry around the current level, and a mapping from variables to level numbers, so as usual we define `freeToLevel` in terms of `freeToLevel_e` which does all the work.

```
> freeToLevel_e :: Level                          -- Level of context
>               -> Assn Name Level                -- Level of in-scope names
>               -> AnnExpr Name (Set Name)        -- Input expression
>               -> AnnExpr (Name, Level) Level    -- Result expression

> freeToLevel e = freeToLevel_e 0 [] e
```

We represent the name-to-level mapping as an association list, with type `Assn Name Level`. The interface of association lists is given in the Appendix. but notice that it is *not* abstract. It is so convenient to use all the standard functions on lists, and notation for lists, rather than to invent their analogues for associations, that we have compromised the abstraction.

For constants, variables and applications, it is simpler and more efficient to ignore the free-variable information and calculate the level number directly.

```
> freeToLevel_e level env (_, AConst k) = (0, AConst k)
```

```
> freeToLevel_e level env (_, AVar v) = (assLookup env v, AVar v)
> freeToLevel_e level env (_, AAp e1 e2) =
>   (max (levelOf e1') (levelOf e2'), AAp e1' e2')
>    where
>    e1' = freeToLevel_e level env e1
>    e2' = freeToLevel_e level env e2
```

This cannot be done for lambda abstractions, so we compute the level number of the abstraction using `freeSetToLevel`. We also assign a level number to each variable in the argument list. At present we expect there to be only one such variable, but we will allow there to be several and assign them all the same level number. This works correctly now, and turns out to be just what is needed to support a useful optimisation later.

```
> freeToLevel_e level env (free, ALam args body) =
>   (freeSetToLevel free env, ALam args' body')
>   where
>   body' = freeToLevel_e (level + 1) (args' ++ env) body
>   args' = zip args (repeat (level+1))
```

Let(rec)-expressions follow the scheme outlined at the beginning of this section.

```
> freeToLevel_e level env (free, ALet isRec defns body) =
>   (levelOf body', ALet isRec defns' body')
>   where
>   binders = bindersOf defns
>   freeRhsVars = setUnionList [free | (free, _) <- rhssOf defns]
>   maxRhsLevel = freeSetToLevel freeRhsVars
>                                ([(name,0) | name <- binders] ++ env)
>   defns' = map freeToLevel_d defns
>   body' = freeToLevel_e level (bindersOf defns' ++ env) body
>
>   freeToLevel_d (name, rhs) = ((name, levelOf rhs'), rhs')
>                               where rhs' = freeToLevel_e level envRhs rhs
>   envRhs | isRec     = [(name,maxRhsLevel) | name <- binders] ++ env
>          | not isRec = env
```

Notice that the level of the whole let(rec) expression is that of the body. This is valid provided the body refers to all the binders directly or indirectly. If any definition is unused, we might assign a level number to the letrec which would cause it to be floated outside the scope of some variable mentioned in the unused definition. This is easily fixed, but it is simpler to assume that the expression contains no redundant definitions[5].

Finally the auxillary function `levelOf` extracts the level from an expression:

```
> levelOf :: AnnExpr a Level -> Level
> levelOf (level, e) = level
```

[5] The dependency analysis phase referred to earlier could eliminate such definitions.

6.7 Identifying MFEs

It is simple to identify MFEs, by comparing the level number of an expression with the level of its context. This requires an auxiliary parameter to give the level of the context.

```
> identifyMFEs_e :: Level -> AnnExpr (Name, Level) Level -> Expr (Name, Level)

> identifyMFEs e = identifyMFEs_e 0 e
```

Once an MFE e has been identified, our strategy is to wrap it in a trivial let-expression of the form `let v = e in v`; but not all MFEs deserve special treatment in this way. For example, it would be a waste of time to wrap such a let-expression around an MFE consisting of a single variable or constant. Other examples are given below, in the section on redundant full laziness. We encode this knowledge of which MFEs deserve special treatment in a function `notMFECandidate`.

```
> notMFECandidate (AConst k) = True
> notMFECandidate (AVar v) = True
> notMFECandidate _ = False     -- For now, everything else is a candidate
```

`identifyMFEs_e` works by comparing the level number of the expression with that of its context. If they are the same, or for some other reason the expression is not a candidate for special treatment, the expression is left unchanged, except that `identifyMFEs_e1` is used to apply `identifyMFEs_e` to its subexpressions; otherwise we use `transformMFE` to perform the appropriate transformation.

```
> identifyMFEs_e cxt (level, e) =
>   if (level == cxt || notMFECandidate e)
>   then e'
>   else transformMFE level e'
>   where
>   e' = identifyMFEs_e1 level e

> transformMFE level e = ELet nonRecursive [(("v",level), e)] (EVar "v")
```

`identifyMFEs_e1` applies `identifyMFEs_e` to the components of the expression. Its definition is straightforward, and is given in the Appendix.

6.8 Renaming and floating

The renaming pass is entirely straightforward, involving plumbing a name supply in a similar manner to `collectSCs`. The final pass, which floats let(rec) expressions out to the appropriate level, is also fairly easy.

Complete definitions for `rename` and `float` are given in the Appendix.

7 Avoiding redundant full laziness

Full laziness does not come for free. It has two main negative effects:

- Multiple lambda abstractions, such as `\x->\y->E` turn into one supercombinator under the simple scheme, but two under the fully lazy scheme. Two reductions instead of one are therefore required to apply it to two arguments, which may well be more expensive.
- Lifting out MFEs removes subexpressions from their context, and thereby reduces opportunities for a compiler to perform optimisations. Such optimisations might be partially restored by an interprocedural analysis which figured out the contexts again, but it is better still to avoid creating the problem.

These points are elaborated by Fairbairn (Fairbairn [1985]) and Goldberg (Goldberg [1988]). Furthermore, they point out that often no benefit arises from lifting out *every* MFE from *every* lambda abstraction. In particular,

- If no partial applications of a multiple abstraction can be shared, then nothing is gained by floating MFEs out to *between* the nested abstractions.
- Very little is gained by lifting out an MFE that is not a reducible expression. No work is shared thereby, though there may be some saving in storage because the closure need only be constructed once. This is more than outweighed by the loss of compiler optimisations caused by removing the expression from its context.
- Lifting out an MFE which is a constant expression (ie level 0), thereby adding an extra parameter to pass in, is inefficient. It would be better to make the constant expression into a supercombinator.

These observations suggest some improvements to the fully lazy lambda lifter, and they turn out to be quite easy to incorporate:

- If a multiple abstraction is *not* separated into separate `ELam` constructors by the `separateLam` pass, then all the variables bound by it will be given the *same* level number. It follows that no MFE will be identified which is free in the inner abstraction but not the outer one. This ensures that no MFEs will be floated out to between two abstractions represented by a single `ELam` constructor.

  All that is required is to modify the `separateLams` pass to keep in a single `ELam` constructor each multiple abstraction of which partial applications cannot be shared. This sharing information is not trivial to deduce, but at least we have an elegant way to use its results by modifying only a small part of our algorithm.

  This is one reason why we chose to allow `ELam` constructors to take a list of binders.

- `identifyMFEs` uses a predicate `notMFECandidate` to decide whether to identify a particular subexpression as an MFE. This provides a convenient place to add extra conditions to exclude from consideration expressions which are not redexes. This condition, too, is undecidable in general, but a good approximation can be made in many cases; for example `(+ 3)` is obviously not a redex.

- When identifying MFEs it is easy to spot those that are constant expressions, because their level numbers are zero. We can rather neatly ensure that it is turned into a supercombinator by the subsequent lambda-lifting pass, by making it into a lambda abstraction *with an empty argument list*. This was the other reason why we decided to make the `ELam` constructor take a list of arguments. The modification affects only `identifyMFEs_e`, thus:

```
> identifyMFEs_e cxt (level, e) =
>   if (level == cxt || notMFECandidate e) then
>              e'
>   else if (level > 0) then
>              ELet nonRecursive [(("v",level), e')] (EVar "v")
>   else
>              ELam [] e'
>   where
>   e' = identifyMFEs_e1 level e
```

8 Retrospective and comparison with other work

It is interesting to compare our approach with Bird's very nice paper (Bird [1987]) which addresses a similar problem. Bird's objective is to give a formal development of an efficient fully lazy lambda lifter, by successive transformation of an initial specification. The resulting algorithm is rather complex, and would be hard to write down directly, thus fully justifying the effort of a formal development.

In contrast, we have expressed our algorithm as a composition of a number of very simple phases, each of which can readily be specified and written down directly. The resulting program has a constant-factor inefficiency, because it makes many traversals of the expression. This is easily removed by folding together successive passes into a single function, eliminating the intermediate data structure. Unlike Bird's transformations, this is a straightforward process.

Our approach has the major advantage that is is *modular*. This allows changes to be made more easily. For example, it would be a simple matter to replace the lambda lifter with one which performed lambda-lifting in the way suggested by Johnsson (Johnsson [1985]), whereas doing the same for Bird's algorithm would be a major exercise. Similarly, it proved rather easy to modify the algorithm to be more selective about where full laziness is introduced.

The main disadvantage of our approach is that we are unable to take advantage of one optimisation suggested by Hughes, namely ordering the parameters to a supercombinator to reduce the number of MFEs. The reason for this is that the optimisation absolutely requires that lambda lifting be entwined with the process of MFE identification, while we have carefully separated these activities! Happily for us, the larger MFEs created by this optimisation are always partial applications, which should probably *not* be identified as MFEs because no work is shared thereby. Even so, matters might not have fallen out so fortuitously, and our separation of concerns has certainly made some sorts of transformation rather difficult.

Acknowledgements

We owe our thanks to John Robson and Kevin Hammond, and two anonymous referees for their useful comments on a draft of this paper.

References

RS Bird [1987], "A formal development of an efficient supercombinator compiler," *Science of Computer Programming* 8, 113–137.

Jon Fairbairn [Feb 1985], "Removing redundant laziness from supercombinators," in *Proc Aspenas workshop on implementation of functional languages.*

Benjamin F Goldberg [April 1988], "Multiprocessor execution of functional programs," YALEU/DCS/RR-618, Dept of Computer Science, Yale University.

CK Holst [1991], "Improving full laziness," in *Proceedings of the 1990 Glasgow Workshop on Functional Programming, Ullapool*, SL Peyton Jones, G Hutton & CK Holst, eds., Workshops in Computing, Springer Verlag.

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

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

Thomas Johnsson [1985], "Lambda lifting: transforming programs to recursive equations," in *Proc IFIP Conference on Functional Programming and Computer Architecture*, Jouannaud, ed., LNCS 201, Springer Verlag, 190–205.

Thomas Johnsson [1987], "Compiling lazy functional languages," PhD thesis, PMG, Chalmers University, Goteborg, Sweden.

ND Jones, P Sestoft & H Søndergaard [1989], "Mix: a self-applicable partial evaluator for experiments in compiler generation," *Lisp and Symbolic Computation* 2, 9–50.

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

A Appendix: Omitted code

This appendix contains the definitions of functions omitted from the main paper.

A.1 abstract

We begin with the function abstract.

```
> abstract (_, AConst k) = EConst k
> abstract (_, AVar v) = EVar v
> abstract (_, AAp e1 e2) = EAp (abstract e1) (abstract e2)

> abstract (free, ALam args body) =
>   foldl EAp sc (map EVar fvList)
>   where
>   fvList = setToList free
>   sc = ELam (fvList ++ args) (abstract body)

> abstract (_, ALet isRec defns body) =
>   ELet isRec [(name, abstract body) | (name, body) <- defns] (abstract body)
```

A.2 separateLams

Next comes the code for separateLams.

```
> separateLams (EConst k) = EConst k
> separateLams (EVar v) = EVar v
> separateLams (EAp e1 e2) = EAp (separateLams e1) (separateLams e2)
> separateLams (ELam args body) = foldr mkLam (separateLams body) args
>                                 where
>                                 mkLam arg body = ELam [arg] body
> separateLams (ELet isRec defns body) =
>   ELet isRec [(name, separateLams rhs) | (name,rhs) <- defns]
>              (separateLams body)
```

A.3 identifyMFEs_e1

identifyMFEs_e1 applies identifyMFEs_e to the components of the expresion.

```
> identifyMFEs_e1 :: Level -> AnnExpr' (Name, Level) Level -> Expr (Name, Level)
> identifyMFEs_e1 level (AConst k) = EConst k
> identifyMFEs_e1 level (AVar v)   = EVar v
> identifyMFEs_e1 level (AAp e1 e2) =
>   EAp (identifyMFEs_e level e1) (identifyMFEs_e level e2)
```

When it encounters a binder it changes the "context" level number carried down as its first argument.

```
> identifyMFEs_e1 level (ALam args body) =
>   ELam args (identifyMFEs_e argLevel body)
>   where
>   (_, argLevel) = head args
>
> identifyMFEs_e1 level (ALet isRec defns body) =
>   ELet isRec defns' body'
>   where
>   body' = identifyMFEs_e level body
>   defns' = [ ((name,rhsLevel),identifyMFEs_e rhsLevel rhs)
>            | ((name,rhsLevel),rhs) <- defns]
```

A.4 rename

The function `rename` gives unique names to the variables in an expression. We need an auxiliary function `rename_e` to do all the work:

```
> rename e = e' where (_, e') = rename_e [] initialNameSupply e

> rename_e :: Assn Name Name -> NameSupply -> Expr (Name,a)
>          -> (NameSupply, Expr (Name,a))
> rename_e env ns (EConst k) = (ns, EConst k)
> rename_e env ns (EVar v) = (ns, EVar (assLookup env v))
> rename_e env ns (EAp e1 e2) =
>   (ns2, EAp e1' e2')
>   where
>   (ns1, e1') = rename_e env ns e1
>   (ns2, e2') = rename_e env ns1 e2
> rename_e env ns (ELam args body) =
>   (ns1, ELam args' body')
>   where
>   (ns1, args') = mapAccuml newBinder ns args
>   (ns2, body') = rename_e (assocBinders args args' ++ env) ns1 body

> rename_e env ns (ELet isRec defns body) =
>   (ns3, ELet isRec (zip binders' rhss') body')
>   where
>   (ns1, body') = rename_e env' ns body
>   binders = bindersOf defns
>   (ns2, binders') = mapAccuml newBinder ns1 binders
>   env' = assocBinders binders binders' ++ env
>   (ns3, rhss') = mapAccuml (rename_e rhsEnv) ns2 (rhssOf defns)
>   rhsEnv |     isRec = env'
>          | not isRec = env
```

`newBinder` is just like `newName`, but works over `(Name,a)` binders.

```
> newBinder ns (name, info) =
>   (ns1, (name', info)) where (ns1, name') = newName ns name
```

`assocBinders` builds an assocation list between the names inside two lists of `(Name,a)` binders.

```
> assocBinders :: [(Name,a)] -> [(Name,a)] -> Assn Name Name
> assocBinders binders binders' = zip (map fst binders) (map fst binders')
```

A.5 `float`

The final pass floats let(rec) expressions out to the appropriate level. The main function has to return an expression together with a list of definitions which should be floated outside the expression.

```
> float_e :: Expr (Name, Level) -> (FloatedDefns, Expression)
```

The top-level function `float` uses `float_e` to do the main work, but if any definitions are floated out to the top level, `float` had better install them at this level, thus:

```
> float e = install floatedDefns e'  where  (floatedDefns, e') = float_e e
```

There are many possible representations for the `FloatedDefns` type, and we will choose a simple one, by representing the definitions being floated as a list, each element of which represents a group of definitions, identified by its level, and together with its `IsRec` flag.

```
> type FloatedDefns = [(Level, IsRec, [Defn Name])]
```

Since the definitions in the list may depend on one another, we add the following constraint: a definition group may depend only on definition groups appearing *earlier* in the `FloatedDefns` list.

It is now possible to define `install`, which wraps an expression in a nested set of let(rec)s containing the specified definitions.

```
> install :: FloatedDefns -> Expression -> Expression
> install defnGroups e =
>   foldr installGroup e defnGroups
>   where
>   installGroup (level, isRec, defns) e = ELet isRec defns e
```

We can now proceed to a definition of `float_e`. The cases for variables, constants and applications are straightforward.

```
> float_e (EConst k) = ([], EConst k)
> float_e (EVar v) = ([], EVar v)
> float_e (EAp e1 e2) = (fd1 ++ fd2, EAp e1' e2')
>                       where
>                       (fd1, e1') = float_e e1
>                       (fd2, e2') = float_e e2
```

How far out should a definition be floated? There is more than one possible choice, but here we choose to install a definition just inside the innermost lambda which binds one its free variables[6].

```
> float_e (ELam args body) =
>   (outerLevelDefns, ELam args' (install thisLevelDefns body'))
>   where
>   args' = [arg | (arg,level) <- args]
>   (_,thisLevel) = head args           -- Extract level of abstraction
>   (floatedDefns, body') = float_e body
>   thisLevelDefns  = filter groupIsThisLevel       floatedDefns
>   outerLevelDefns = filter (not.groupIsThisLevel) floatedDefns
>   groupIsThisLevel (level,isRec,defns) = level >= thisLevel
```

The case for a let(rec) expression adds its definition group to those floated out from its body, and from its right-hand sides. The latter must come first, since the new definition group may depend on them.

```
> float_e (ELet isRec defns body) =
>   (rhsFloatDefns ++ [thisGroup] ++ bodyFloatDefns, body')
>   where
>   (bodyFloatDefns, body') = float_e body
>   (rhsFloatDefns, defns') = mapAccuml float_defn [] defns
>   thisGroup = (thisLevel, isRec, defns')
>   (_,thisLevel) = head (bindersOf defns)
>
> float_defn floatedDefns ((name,level), rhs) =
>   (rhsFloatDefns ++ floatedDefns, (name, rhs'))
>   where
>   (rhsFloatDefns, rhs') = float_e rhs
```

A.6 mapAccuml

Finally, here is the full definition of utility function `mapAccuml`.

```
> mapAccuml :: (b -> a -> (b, c))   -- Function of element of input list
>                                   --   and accumulator, returning new
>                                   --   accumulator and element of result list
>           -> b                    -- Initial accumulator
>           -> [a]                  -- Input list
>           -> (b, [c])             -- Final accumulator and result list
>
> mapAccuml f b []     = (b, [])
> mapAccuml f b (x:xs) = (b2, x':xs') where (b1, x') = f b x
>                                           (b2, xs') = mapAccuml f b1 xs
```

[6] Recall that all variables bound by a single `ELam` construct are given the same level.

B Appendix: Interface to `Utilities` module

This is the text of the `interface` for the `Utilities` module. This interface is imported by the line `import Utilities` in the `LambdaLift` module. The interface can be deduced by the compiler from the text of the `Utilities` module; indeed the interface below is exactly that produced by the compiler, modulo some reordering and comments.

The first line introduces and names the interface.

```
> interface Utilities where
```

Next we have declarations for the `Set` type. Notice that the data type `Set` is given with an abbreviated `data` declaration, which omits the constructor(s). This makes the `Set` type *abstract* since the importing module can only build sets and take them apart using the functions provided in the interface.

```
> data Set a
> setDifference :: (Ord a) => (Set a) -> (Set a) -> Set a
> setIntersect  :: (Ord a) => (Set a) -> (Set a) -> Set a
> setUnion      :: (Ord a) => (Set a) -> (Set a) -> Set a
> setUnionList  :: (Ord a) => [Set a] -> Set a
> setToList     :: (Set a) -> [a]
> setSingleton  :: a -> Set a
> setEmpty      :: Set a
> setFromList   :: (Ord a) => [a] -> Set a
```

Bags, association lists and the name supply follow similarly. Notice that association lists are declared with a type synonym, so they are not abstract.

```
> data Bag a
> bagUnion     :: (Bag a) -> (Bag a) -> Bag a
> bagInsert    :: a -> (Bag a) -> Bag a
> bagToList    :: (Bag a) -> [a]
> bagFromList  :: [a] -> Bag a
> bagSingleton :: a -> Bag a
> bagEmpty     :: Bag a

> type Assn a b = [(a, b)]
> assLookup :: (Eq a) => [(a, b)] -> a -> b

> data NameSupply
> initialNameSupply :: NameSupply
> newName           :: NameSupply -> [Char] -> (NameSupply, [Char])
```