Chapter 6

Lambda Lifting

6.1 Introduction

In this chapter[1] we will be looking at ways to extend the set of programs that are acceptable to the machines we have looked at previously in the book. The extension that we introduce is to allow *local function definitions*. We are then faced with alternative approaches:

- add a mechanism to the machines to deal with environments; or
- transform the program so that there are no local function definitions; instead all functions are defined as supercombinators.

In this book we have always assumed that the second approach would be taken.

This chapter is also an appropriate point at which to introduce the concept of *full laziness*. Again, this desirable optimisation of functional languages is achieved using a program transformation.

6.2 Improving the `expr` data type

Before we begin the program proper, we must import the language and utilities modules.

```
> module Lambda where
> import Utils
> import Language
```

Unfortunately, the data types defined there (`coreExpr`, `coreProgram` and so on) are insufficiently flexible for our needs in this chapter, so we will attend to this problem first. 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.

[1]Some of the material in this chapter was first published in [Peyton Jones and Lester 1991].

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

```
> expr * = EVar name
>        | ...
>        | EAnnot annotation (expr *)
```

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

```
> annotation ::= FreeVars (set name)
>              | Level num
```

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 `expr`, 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 `let(rec)` definitions, and the bound variables in lambda abstractions or case expressions. We will call these occurrences *binders*. An example of the need to annotate binders comes in type inference, where 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.

We will address the second problem first, since it has an easy solution. Recall from Section 1.3 that the `expr` type was *parameterised* with respect to the type of its binders; we repeat its definition here as a reminder:

```
> expr *
>   ::=  EVar name                          || Variables
```

```
>       | ENum num                       || Numbers
>       | EConstr num num                || Constructor tag arity
>       | EAp (expr *) (expr *)          || Applications
>       | ELet                           || Let(rec) expressions
>           isRec                        ||   boolean with True = recursive,
>           [(*, expr *)]                ||   Definitions
>           (expr *)                     ||   Body of let(rec)
>       | ECase                          || Case expression
>           (expr *)                     ||   Expression to scrutinise
>           [alter *]                    ||   Alternatives
>       | ELam [*] (expr *)              || Lambda abstractions
```

The type `coreExpr` is a specialised form of `expr` in which the binders are of type `name`. This is expressed using a type synonym (also repeated from Section 1.3):

```
> coreExpr == expr name
```

The advantage of parameterising `expr` is that we can also define other specialised forms. For example, `typedExpr` is a data type in which binders are names annotated with a type:

```
> typedExpr = expr (name, typeExpr)
```

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

Returning to annotations on expressions, we can reuse the same technique by parameterising the data type of expressions with respect 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 Miranda data type definitions:

```
> type AnnExpr a b = (b, AnnExpr' a b)
```

```
> data AnnExpr' a b = AVar Name
>                   | ANum Int
>                   | AConstr Int Int
>                   | AAp (AnnExpr a b) (AnnExpr a b)
>                   | ALet Bool [AnnDefn a b] (AnnExpr a b)
>                   | ACase (AnnExpr a b) [AnnAlt a b]
>                   | ALam [a] (AnnExpr a b)
```

```
> type AnnDefn a b = (a, AnnExpr a b)
```

```
> type AnnAlt a b  = (Int, [a], (AnnExpr a b))
```

```
> type AnnProgram a b = [(Name, [a], AnnExpr a b)]
```

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 *` is nearly isomorphic to `annExpr * **`, but there are two reasons why we choose 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 * **`.

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 modern functional programming languages.

Exercise 6.1. Our present pretty-printing function, `pprint`, defined in Section 1.5, is only able to print `corePrograms`. In order to print out intermediate stages in the lambda lifter we will need a function `pprintGen` which can display values of type `program *`. (The '`Gen`' is short for 'generic'.) `pprintGen` needs an extra argument to tell it how to print the binders:

```
> pprintGen :: (* -> iseq)      || function from binders to iseq
>              -> program *     || the program to be formatted
>              -> [char]        || result string
```

For example, once we have written `pprintGen` we can define `pprint` in terms of it:

```
> pprint prog = pprintGen iStr prog
```

Write a definition for `pprintGen`, and its associated functions `pprExprGen`, and so on.

Exercise 6.2. Do a similar job for printing values of type `annProgram * **`. Here you will need two extra arguments, one for formatting the binders and one for formatting the annotations:

```
> pprintAnn :: (* -> iseq)              || function from binders to iseq
>              -> (** -> iseq)          || function from annotations to iseq
>              -> annProgram * **       || program to be displayed
>              -> [char]                || result string
```

Write a definition for `pprintAnn` and its associated auxiliary functions.

6.3 Mark 1: A simple lambda lifter

Any implementation of a lexically scoped programming language has to cope with the fact that a function or procedure 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 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 1985], but the transformation was independently developed by [Hughes 1983].

In our context, lambda lifting transforms a Core-language program into an equivalent one in which there are no embedded lambda abstractions. To take a simple example, consider the program

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

The `\y` abstraction can be removed by defining a new supercombinator `$g` which takes `x` as an extra parameter, but whose body is the offending abstraction, and replacing the `\y` abstraction with an application of `$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
```

How did we decide to make just `x` into the extra parameter to `$g`? We did it because `x` is a *free variable* of the abstraction `\y. x*x + y`:

> **Definition.** *An occurrence of a variable v in an expression e is said to be* free *in e if the occurrence is not bound by an enclosing lambda or* `let(rec)` *expression in e.*

On the other hand, `y` is not free in `(\y. x*x + y)`, because its occurrence is in the scope of an enclosing lambda abstraction which binds it.

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

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

Now `x` and `g` are both free in the `\y` abstraction, so the lambda lifter will make them both into extra parameters of `$g`, producing 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; and in particular by all of the abstract machines in this book.

There one final gloss to add: there is no need to treat other top-level functions as extra parameters. For example, consider the program

```
h p q = p*q
f x = let g = \y. (h x x) + y in (g 3 + g 4)
main = f 6
```

Here we do not want to treat `h` as a free variable of the `\y` abstraction, because it is a constant which can be referred to directly from the body of the new `$g` supercombinator. The same applies, of course, to the `+` and `*` functions! In short, only supercombinator arguments, and variables bound by lambda abstractions or `let(rec)` expressions, are candidates for being free variables.

It is worth noting in passing that the lexical-scoping 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.

6.3.1 Implementing a simple lambda lifter

We are now ready to develop a simple lambda lifter. It will take a `coreProgram` and return an equivalent `coreProgram` in which there are no occurrences of the `ELam` constructor.

```
> lambdaLift :: CoreProgram -> CoreProgram
```

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 :: CoreProgram -> AnnProgram Name (Set Name)
  ```

  The type `set *` is a standard abstract data type for sets, whose definition is given in Appendix A.4.

- Second, the function `abstract` abstracts from each lambda abstraction $\backslash x_1 \ldots x_n \,.\, e$ its free variables $v_1 \ldots v_m$, replacing the lambda abstraction with an expression of the form

  $$\texttt{(let sc = }\backslash v_1 \ldots v_m\ x_1 \ldots x_n\,.\ e\ \texttt{in sc)}\ v_1 \ldots v_m$$

  We could use a direct application of the lambda abstraction to the free variables, but we need to give the new supercombinator a name, so we take the first step here by always giving it the name `sc`. For example, the lambda abstraction

```
(\x. y*x + y*z)
```

would be transformed to

```
(let sc = (\y z x.  y*x + y*z) in sc) y z
```

`abstract` has the type signature:

```
> abstract :: AnnProgram Name (Set Name) -> CoreProgram
```

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

- Now we traverse the program giving a unique name to each variable. This will have the effect of making unique all the `sc` variables introduced by the previous pass. Indeed, the sole purpose of introducing the extra `let` expressions in the first place was to give each supercombinator a name which could then be made unique. As a side effect, all other names in the program will be make unique, but this does not matter, and it will turn out to be useful later.

```
> rename :: CoreProgram -> CoreProgram
```

- Finally, `collectSCs` collects all the supercombinator definitions into a single list, and places them at the top level of the program.

```
> collectSCs :: CoreProgram -> CoreProgram
```

The lambda lifter itself is the composition of these three functions:

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

To make it easier to see what is happening we define the a function `runS` (the '`S`' stands for 'simple') to integrate the parser and printer:

```
> runS = pprint . lambdaLift . parse
```

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.

As an example of the final point, some compilers are able to generate better code by omitting the `collectSCs` pass, because more is then known about the context in which the supercombinator is applied [Peyton Jones 1991]. For example, consider the following 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[2]. Rather than compiling the supercombinator independently of its context, a compiler could construct 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.

In the following sections we give definitions for each of these passes. We omit the equations for case expressions, which appear as Exercise 6.4.

6.3.2 Free variables

The core of the free-variable pass is function freeVars_e which has type

```
> freeVars_e :: (Set Name)                    -- Candidates for free variables
>               -> CoreExpr                   -- Expression to annotate
>               -> AnnExpr Name (Set Name)    -- Annotated result
```

Its first argument is the set of local variables which are in scope; these are the possible free variables. The second argument is the expression to be annotated, and the result is the annotated expression. The main function freeVars just runs down the list of supercombinator definitions, applying freeVars_e to each:

```
> freeVars prog = [ (name, args, freeVars_e (setFromList args) body)
>                 | (name, args, body) <- prog
>                 ]
```

The freeVars_e function runs over the expression recursively; in the case of numbers there are no free variables, so this is what is returned in the annotated expression.

```
> freeVars_e lv (ENum k)      = (setEmpty, ANum k)
```

In the case of a variable, we check to see whether it is in the set of candidates to decide whether to return the empty set or a singleton set:

```
> freeVars_e lv (EVar v) | setElementOf v lv = (setSingleton v, AVar v)
>                        | otherwise         = (setEmpty, AVar v)
```

The case for applications is straightforward: we first annotate the expression e1 with its free variables, then annotate e2, returning the union of the two sets of free variables as the free variables of EAp e1 e2.

```
> freeVars_e lv (EAp e1 e2)
>  = (setUnion (freeVarsOf e1') (freeVarsOf e2'), AAp e1' e2')
>    where e1'           = freeVars_e lv e1
>          e2'           = freeVars_e lv e2
```

[2]We are ignoring the let expression which abstract introduces to name the supercombinator.

In the case of a lambda abstractions we need to add the args to the local variables passed in, and subtract them from the free variables passed out:

```
> freeVars_e lv (ELam args body)
>  = (setSubtraction (freeVarsOf body') (setFromList args), ALam args body')
>    where body'          = freeVars_e new_lv body
>          new_lv         = setUnion lv (setFromList args)
```

The equation for let(rec) expressions has rather a lot of plumbing, but is quite straightforward. The local variables in scope that are passed to the body is body_lv; the set of local variables passed to each right-hand side is rhs_lv. Next we annotate each right-hand side with its free variable set, giving rhss', from this we can construct the annotated definitions: defns'. The annotated body of the let(rec) is body'. The free variables of the definitions is calculated to be defnsFree, and those of the body are bodyFree.

```
> freeVars_e lv (ELet is_rec defns body)
>  = (setUnion defnsFree bodyFree, ALet is_rec defns' body')
>    where binders        = bindersOf defns
>          binderSet      = setFromList binders
>          body_lv        = setUnion lv binderSet
>          rhs_lv | is_rec    = body_lv
>                 | otherwise = lv
>
>          rhss'          = map (freeVars_e rhs_lv) (rhssOf defns)
>          defns'         = zip2 binders rhss'
>          freeInValues   = setUnionList (map freeVarsOf rhss')
>          defnsFree | is_rec    = setSubtraction freeInValues binderSet
>                    | otherwise = freeInValues
>          body'          = freeVars_e body_lv body
>          bodyFree       = setSubtraction (freeVarsOf body') binderSet
```

The function zip2 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, setSubtraction and so on are defined in the utilities module, whose interface is given in Appendix A.4.

We postpone dealing with case and constructor expressions:

```
> freeVars_e lv (ECase e alts) = freeVars_case lv e alts
> freeVars_e lv (EConstr t a) = error "freeVars_e: no case for constructors"
> freeVars_case lv e alts = error "freeVars_case: not yet written"
```

freeVarsOf and freeVarsOf_alt are simple auxiliary functions:

```
> freeVarsOf :: AnnExpr Name (Set Name) -> Set Name
> freeVarsOf (free_vars, expr) = free_vars
```

```
> freeVarsOf_alt :: AnnAlt Name (Set Name) -> Set Name
> freeVarsOf_alt (tag, args, rhs)
>  = setSubtraction (freeVarsOf rhs) (setFromList args)
```

6.3.3 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.

```
> abstract prog = [ (sc_name, args, abstract_e rhs)
>                 | (sc_name, args, rhs) <- prog
>                 ]
```

As usual, we define an auxiliary function `abstract_e` to do most of the work:

```
> abstract_e :: AnnExpr Name (Set Name) -> CoreExpr
```

It takes an expression annotated with the free variable information and returns an expression with each lambda abstraction replaced by a new abstraction applied to the free variables. There is little to say about the first four cases, they just recursively abstract each expression.

```
> abstract_e (free, AVar v)    = EVar v
> abstract_e (free, ANum k)    = ENum k
> abstract_e (free, AAp e1 e2) = EAp (abstract_e e1) (abstract_e e2)
> abstract_e (free, ALet is_rec defns body)
>  =  ELet is_rec [ (name, abstract_e body) | (name, body) <- defns]
>                 (abstract_e body)
```

The function `foldll` is a standard function, defined in Appendix A.5; given a dyadic function $\oplus$, a value b, and a list $xs = [x_1, ..., x_n]$, `foldll` $\oplus$ 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.

The final case we show is the heart of the `abstract_e` function. First we create a list of free variables: `fvList`. We recall that there is no ordering implicit in a set; the function `setToList` has induced an ordering on the elements, but we do not much care what order this is. Next we make a new supercombinator. This involves

1. applying `abstract_e` to the body of the lambda expression; and
2. augmenting the argument list, by prefixing the original one with the free-variable list.

Next, to allow the `collectSCs` pass to detect this new supercombinator, we wrap it into a `let` expression. Finally, we apply the new supercombinator to each free variable in turn.

```
> abstract_e (free, ALam args body)
>  = foldll EAp sc (map EVar fvList)
```

```
>    where
>    fvList = setToList free
>    sc = ELet nonRecursive [("sc",sc_rhs)] (EVar "sc")
>    sc_rhs = ELam (fvList ++ args) (abstract_e body)
```

`case` expressions and constructors are deferred:

```
> abstract_e (free, AConstr t a) = error "abstract_e: no case for Constr"
> abstract_e (free, ACase e alts) = abstract_case free e alts
> abstract_case free e alts = error "abstract_case: not yet written"
```

It is worth observing that `abstract_e` 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 1983].

6.3.4 Making all the variables unique

Next, we need to make each variable so that all the `sc` variables introduced by `abstract` are unique. The auxiliary function, `rename_e`, takes an environment mapping old names to new names, a name supply and an expression. It returns a depleted name supply and a new expression.

```
> rename_e :: ASSOC Name Name                 -- Binds old names to new
>             -> NameSupply                   -- Name supply
>             -> CoreExpr                     -- Input expression
>             -> (NameSupply, CoreExpr)       -- Depleted supply and result
```

Now we can define `rename` in terms of `rename_e`, by applying the latter to each supercombinator definition, plumbing the name supply along with `mapAccuml`.

```
> rename prog
>  = second (mapAccuml rename_sc initialNameSupply prog)
>    where
>    rename_sc ns (sc_name, args, rhs)
>     = (ns2, (sc_name, args', rhs'))
>       where
>       (ns1, args', env) = newNames ns args
>       (ns2, rhs') = rename_e env ns1 rhs
```

The function `newNames` takes a name supply and a list of names as its arguments. It allocates a new name for each old one from the name supply, returning the depleted name supply, a list of new names and an association list mapping old names to new ones.

```
> newNames :: NameSupply -> [Name] -> (NameSupply, [Name], ASSOC Name Name)
> newNames ns old_names
>  = (ns', new_names, env)
>    where
>    (ns', new_names) = getNames ns old_names
>    env = zip2 old_names new_names
```

The definition of `rename_e` is now straightforward, albeit dull. When we meet a variable, we look it up in the environment. For top-level functions and built-in functions (such as `+`) we will find no substitution for it in the environment, so we just use the existing name:

```
> rename_e env ns (EVar v)      = (ns, EVar (aLookup env v v))
```

Numbers and applications are easy.

```
> rename_e env ns (ENum n)      = (ns, ENum n)
> 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
```

When we meet an `ELam` we need to invent new names for the arguments, using `newNames`, and augment the environment with the mapping returned by `newNames`.

```
> rename_e env ns (ELam args body)
>  = (ns1, ELam args' body')
>    where
>    (ns1, args', env') = newNames ns args
>    (ns2, body') = rename_e (env' ++ env) ns1 body
```

`let(rec)` expressions work similarly:

```
> rename_e env ns (ELet is_rec defns body)
>  = (ns3, ELet is_rec (zip2 binders' rhss') body')
>    where
>    (ns1, body') = rename_e body_env ns body
>    binders = bindersOf defns
>    (ns2, binders', env') = newNames ns1 binders
>    body_env = env' ++ env
>    (ns3, rhss') = mapAccuml (rename_e rhsEnv) ns2 (rhssOf defns)
>    rhsEnv | is_rec    = body_env
>           | otherwise = env
```

We leave `case` expressions as an exercise:

```
> rename_e env ns (EConstr t a) = error "rename_e: no case for constructors"
> rename_e env ns (ECase e alts) = rename_case env ns e alts
> rename_case env ns e alts = error "rename_case: not yet written"
```

6.3.5 Collecting supercombinators

Finally, we have to name the supercombinators and collect them together. The main function, `collectSCs_e`, therefore has to return the collection of supercombinators it has found, as well as the transformed expression.

```
> collectSCs_e :: CoreExpr -> ([CoreScDefn], CoreExpr)
```

`collectSCs` is defined using `mapAccuml` to do all the plumbing:

```
> collectSCs prog
>  = concat (map collect_one_sc prog)
>    where
>    collect_one_sc (sc_name, args, rhs)
>     = (sc_name, args, rhs') : scs
>      where
>      (scs, rhs') = collectSCs_e rhs
```

The code for `collectSCs_e` is now easy to write. We just apply `collectSCs_e` recursively to the sub-expressions, collecting up the supercombinators thus produced.

```
> collectSCs_e (ENum k)      = ([], ENum k)
> collectSCs_e (EVar v)      = ([], EVar v)
> collectSCs_e (EAp e1 e2)   = (scs1 ++ scs2, EAp e1' e2')
>                              where
>                              (scs1, e1') = collectSCs_e e1
>                              (scs2, e2') = collectSCs_e e2

> collectSCs_e (ELam args body) = (scs, ELam args body')
>                                 where
>                                 (scs, body') = collectSCs_e body
> collectSCs_e (EConstr t a) = ([], EConstr t a)
> collectSCs_e (ECase e alts)
>  = (scs_e ++ scs_alts, ECase e' alts')
>    where
>    (scs_e, e') = collectSCs_e e
>    (scs_alts, alts') = mapAccuml collectSCs_alt [] alts
>    collectSCs_alt scs (tag, args, rhs) = (scs++scs_rhs, (tag, args, rhs'))
>                                          where
>                                          (scs_rhs, rhs') = collectSCs_e rhs
```

The case for `let(rec)` is the interesting one. We need to process the definitions recursively and then split them into two groups: those of the form v = *args*. e (the supercombinators), and the others (the non-supercombinators). The supercombinators are returned as part of the supercombinator list, and a new `let(rec)` is formed from the remaining non-supercombinators:

```
> collectSCs_e (ELet is_rec defns body)
```

```
>  = (rhss_scs ++ body_scs ++ local_scs, mkELet is_rec non_scs' body')
>    where
>    (rhss_scs,defns') = mapAccuml collectSCs_d [] defns
>
>    scs'     = [(name,rhs) | (name,rhs) <- defns',  isELam rhs ]
>    non_scs' = [(name,rhs) | (name,rhs) <- defns',  not (isELam rhs)]
>    local_scs = [(name,args,body) | (name,ELam args body) <- scs']
>
>    (body_scs, body') = collectSCs_e body
>
>    collectSCs_d scs (name,rhs) = (scs ++ rhs_scs, (name, rhs'))
>                                  where
>                                  (rhs_scs, rhs') = collectSCs_e rhs
```

The auxiliary function `isELam` tests for an `ELam` constructor; it is used to identify supercombinators.

```
> isELam :: Expr a -> Bool
> isELam (ELam args body) = True
> isELam other            = False
```

The `mkELet` function just builds an `ELet` expression:

```
> mkELet is_rec defns body = ELet is_rec defns body
```

6.4 Mark 2: Improving the simple lambda lifter

This completes the definition of the simple lambda lifter. We now consider some simple improvements.

6.4.1 Simple extensions

Exercise 6.3. The simple lambda lifter generates lots of `let` expressions with an empty list of bindings, because `collectSCs` removes the single binding from each of the supercombinator `let` expressions introduced by `abstract`. Modify `mkELet` to elide these redundant `let` expressions.

Exercise 6.4. Give definitions for `freeVars_case`, `abstract_case` and `collectSCs_case`, and test them.

6.4.2 Eliminating redundant supercombinators

Consider the Core-language program

```
        f = \x. x+1
```

This will be transformed by `lambdaLift` to

```
f = $f
$f x = x+1
```

It would be nicer to avoid introducing the redundant definition. This improvement will become rather more significant when we come to consider full laziness, because many supercombinators of this form will be introduced.

Exercise 6.5. Add a special case to the function `collect_one_sc` (in `collectSCs`), to behave differently when `rhs` is a lambda abstraction. You should be able to avoid introducing a new supercombinator in this situation.

6.4.3 Eliminating redundant local definitions

A similar situation can arise with local definitions. Consider the Core-language program

```
f x = let g = (\y. y+1) in g (g x)
```

The lambda lifter will produce the program

```
f x = let g = $g in g (g x)
$g y = y+1
```

Exercise 6.6. Improve the definition of `collectSCs_d` (in the `ELet` case of `collectSCs_e`), so that it gives special treatment to definitions whose right-hand side is a lambda abstraction. For the above example you should generate

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

6.5 Mark 3: Johnsson-style lambda lifting

There is an interesting variant of the lambda lifting technique, which was discovered by [Johnsson 1985]. One slight problem with our current technique is that it produces programs in which many of the calls are to functions which are passed in as arguments. For example, consider the recursive example in Section 6.3:

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

Our current lambda lifter produces 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 `$g` makes a call to its argument `g`. In some implementations it would be more efficient if `$g` was directly recursive, like this:

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

The inner `letrec` has vanished altogether, and the supercombinator g has become directly recursive.

To get a more detailed idea of how to do Johnsson-style lambda lifting, we will look at a slightly more complicated example:

```
f x y = letrec
            g = \p. ...h...x...
            h = \q. ...g...y...
        in
        ...g...h...
```

Here, `g` is meant to be a function which calls `h`, and mentions the variable `x`; similarly `h` calls `g` and mentions `y`. The first step is to transform the definition like this:

```
f x y = letrec
            g = \x y p. ...(h x y)...x...
            h = \x y q. ...(g x y)...y...
        in
        ...(g x y)...(h x y)...
```

This transformation, which we call the *abstraction step*, is a little tricky. It does the following:

- take the free variables of the right-hand sides of the `letrec`, namely `g`, `h`, `x` and `y`;
- exclude the variables being bound (`g` and `h`) to leave just `x` and `y`;
- make these variables into extra arguments of each right-hand side;
- and replaced all occurrences of `g` with `(g x y)`, similarly for `h`.

It is important that we make `y` into an extra parameter of `g` even though `y` does not occur directly in its right-hand side, because `g` will need it to pass to `h`. In general, each member of the mutually recursive group must take as extra arguments the free variables of *all* the members together.

Now all we need do is float the definitions of `g` and `h` out to the top level, leaving:

```
f x y = ...(g x y)...(h x y)...
g x y p = ...(h x y)...x...
h x y q = ...(g x y)...y...
```

One last point. Before doing this process it is important that all binders are unique. Otherwise name clashes could arise, in two ways. The obvious way is that two supercombinators could have the same name. The less obvious way is illustrated by the following variant of the same example:

```
f x y = letrec
              g = \p. ...h...x...
              h = \x. ...g...y...
           in
           ...g...h...
```

Now h uses the same name for its argument as f, which will cause trouble when we try to make the free variable of g, namely x, into an extra argument to h! All in all, it is much easier simply to rename the program before starting work.

6.5.1 Implementation

The Johnsson-style lambda lifter can be implemented in several passes:

- The first pass renames the program so that all binders are unique. We can reuse the `rename` function for this purpose.
- Next, we annotate the program with its free-variable information, using the existing function `freeVars`.
- Now comes the main abstraction step discussed above:

  ```
  > abstractJ :: AnnProgram Name (Set Name) -> CoreProgram
  ```

- Finally, we can collect supercombinators with `collectSCs`.

The full Johnsson-style lambda lifter is just the composition of these stages:

```
> lambdaLiftJ = collectSCs . abstractJ . freeVars . rename
> runJ = pprint . lambdaLiftJ . parse
```

6.5.2 Abstracting free variables in functions

The only new function we need is `abstractJ`. The abstraction process makes substitutions as it goes along, replacing g with $g\ v_1 \ldots v_n$, where g is one of the new supercombinators and $v_1, \ldots, v_n$ are the free variables of its declaration group. It follows that the auxiliary function `abstractJ_e` needs to take an environment mapping each supercombinator g to the free variables of its group $v_1, \ldots, v_n$:

```
> abstractJ_e :: ASSOC Name [Name]              -- Maps each new SC to
>                                               --   the free vars of its group
>                -> AnnExpr Name (Set Name)     -- Input expression
>                -> CoreExpr                    -- Result expression
```

To be fair, it looks as though the first argument could be of type `assoc name coreExpr` but, as we shall see, we need to make use of the environment in another way as well, which leads to the type we suggest here.

It is now easy to define `abstractJ` in terms of `abstractJ_e`, by applying the latter to each top-level definition:

```
> abstractJ prog = [ (name,args,abstractJ_e [] rhs)
>                  | (name, args, rhs) <- prog]
```

Now we come to `abstractJ_e`. The cases for constants and applications are easy.

```
> abstractJ_e env (free, ANum n)      = ENum n
> abstractJ_e env (free, AConstr t a) = EConstr t a
> abstractJ_e env (free, AAp e1 e2)   = EAp (abstractJ_e env e1)
>                                           (abstractJ_e env e2)
```

When we come to a variable g, we look it up in the environment, getting back a list of variables $v_1, \ldots, v_n$. We then return the application $g\ v_1 \ldots v_n$. If g does not appear in the environment we return the empty list from the environment lookup, and hence the 'application' we construct will simply be the variable g!

```
> abstractJ_e env (free, AVar g)
>  = foldll EAp (EVar g) (map EVar (aLookup env g []))
```

Sometimes we may find a lambda abstraction on its own; for example:

```
        f xs = map (\x. x+1) xs
```

The `\x`-abstraction is not the right-hand side of a `let(rec)` definition, so we treat it in just same way as we did in the simple lambda lifter (see the `ELam` case of `abstract`).

There is just one important difference. Since `abstractJ_e` is simultaneously performing a substitution on the expression, the free-variables information does not reflect the post-substitution state of affairs. Rather, we need to perform the substitution on the free-variable set too, to find what variables are free in the result. This is done by the function `actualFreeList`, which is defined at the end of this section. It was the need to perform this operation on the free-variable information which guided our choice of environment representation.

```
> abstractJ_e env (free, ALam args body)
>  = foldll EAp sc (map EVar fv_list)
>    where
>    fv_list = actualFreeList env free
>    sc = ELet nonRecursive [("sc",sc_rhs)] (EVar "sc")
>    sc_rhs = ELam (fv_list ++ args) (abstractJ_e env body)
```

Lastly, we treat `let(rec)` expressions. Each variable bound to a lambda abstraction will be turned into a supercombinator, while the others (not bound to a lambda abstraction) will not. It follows that we need to treat separately these two kinds of definitions, which we call 'function definitions' and 'variable definitions' respectively.

```
> abstractJ_e env (free, ALet isrec defns body)
>  = ELet isrec (fun_defns' ++ var_defns') body'
>    where
>    fun_defns = [(name,rhs) | (name,rhs) <- defns,  isALam rhs ]
>    var_defns = [(name,rhs) | (name,rhs) <- defns,  not (isALam rhs)]
```

Now that we have separated the function definitions from the variable definitions we can compute the set of variables to abstract from the functions. We take the union of the free variables of the function definitions, remove from this set the function names being bound, and then use `actualFreeList` (for the same reason as in the `ELam` equation) to get the result:

```
>    fun_names = bindersOf fun_defns
>    free_in_funs = setSubtraction
>                        (setUnionList [freeVarsOf rhs | (name,rhs)<-fun_defns])
>                        (setFromList fun_names)
>    vars_to_abstract = actualFreeList env free_in_funs
```

Next, we compute the new environment, to be used in the right-hand sides and in the body of the `let(rec)`:

```
>    body_env = [(fun_name, vars_to_abstract) | fun_name <- fun_names] ++ env
>    rhs_env | isrec     = body_env
>            | otherwise = env
```

Lastly, we compute the new function definitions, variable definitions and body, by recursively using `abstractJ_E` with the appropriate environment:

```
>    fun_defns' = [ (name, ELam (vars_to_abstract ++ args)
>                               (abstractJ_e rhs_env body))
>                 | (name, (free, ALam args body)) <- fun_defns
>                 ]
>    var_defns' = [(name, abstractJ_e rhs_env rhs) | (name, rhs) <- var_defns]
>    body' = abstractJ_e body_env body
```

The function `actualFreeList` takes the environment and a set of free variables, applies the environment to the set, and returns a list (without duplicates) of the post-substitution free variables.

```
> actualFreeList :: ASSOC Name [Name] -> Set Name -> [Name]
> actualFreeList env free
>  = setToList (setUnionList [ setFromList (aLookup env name [name])
>                            | name <- setToList free
>                            ])
```

The function `isALam` identifies `ALam` constructors.

```
> isALam :: AnnExpr a b -> Bool
> isALam (free, ALam args body) = True
> isALam other                  = False
```

This concludes the Johnsson-style lambda lifter.

Exercise 6.7. Add a case for `case` expressions to the function `abstractJ_e`.

6.5.3 A tricky point†

When a `letrec` contains a mixture of function and variable definitions, the lambda lifter we have designed may introduce some redundant parameters. For example, consider the definition

```
f x y = letrec
          g = \p. h p + x ;
          h = \q. k + y + q;
          k = g y
        in
        g 4 ;
```

The free variables of the `g`/`h` group are `x`, `y` and `k`, so we will transform to:

```
f x y = letrec
           k = g x y k y
        in
        g 4

g x y k p = h x y k p + x ;
h x y k q = k + y + q;
```

Here the extra parameter `x` is not actually used in `h`, so better definitions for `g` and `h` would be

```
g x y k p = h y k p + x ;
h y k q = k + y + q;
```

Exercise 6.8. †Modify `abstractJ` to solve perform this more sophisticated transformation. Warning: this is quite a difficult job!

6.6 Mark 4: A separate full laziness pass

We now turn our attention to an important property of functional programs called *full laziness*. 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 1983] gives an algorithm, but it is extremely subtle and does not handle `let(rec)` expressions. On the other hand, [Peyton Jones 1987] does cover `let(rec)` expressions, but the description is only informal and no algorithm is given.

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.

6.6.1 A review of full laziness

We begin by briefly reviewing the concept of full laziness. Consider again the example given in Section 6.3.

```
f x = let g = \y. x*x + y in (g 3 + g 4)
main = 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 1990]. 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 *et al.* 1989], though with much less effort.

6.6.2 Fully-lazy lambda lifting in the presence of `let(rec)`s

Writing a fully lazy lambda lifter, as outlined in the previous section, is surprisingly difficult. 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 an 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` is only 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 – described in [Peyton Jones 1987, Chapter 15] – namely to 'float' each `let(rec)` definition outward until it is outside any lambda abstraction in which it is free. 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 `letrec`s 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 will look at this in Section 6.8.

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.

6.6.3 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`:

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

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

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

Now the `let` expression is floated outward:

```
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-abstraction, 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` expressions. 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.

6.6.4 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 each `ELam` constructor and supercombinator definition binds only a single argument, 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 :: CoreProgram -> CoreProgram
  ```

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

```
> type Level = Int
> addLevels :: CoreProgram -> AnnProgram (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 :: AnnProgram (Name, Level) Level -> Program (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. Sadly, this means that we cannot use our current `rename` function because it works on a `coreProgram`, whereas `identifyMFEs` has produced a `program (name, level)`. We invent a new function `renameL` for the purpose:

```
> renameL ::  Program (Name, a) -> Program (Name, a)
```

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

```
> float :: Program (Name,Level) -> CoreProgram
```

- Finally, ordinary lambda lifting can be carried out, using `lambdaLift` from Section 6.3.1.

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

```
> fullyLazyLift = float . renameL . identifyMFEs . addLevels . separateLams
> runF          = pprint . lambdaLift . fullyLazyLift . parse
```

As before, we leave most of the equations for `case` expressions as an exercise.

6.6.5 Separating the lambdas

We define `separateLams` in terms of an auxiliary function `separateLams_e`, which recursively separates variables bound in lambda abstractions in expressions:

```
> separateLams_e :: CoreExpr -> CoreExpr
> separateLams_e (EVar v) = EVar v
> separateLams_e (EConstr t a) = EConstr t a
> separateLams_e (ENum n) = ENum n
> separateLams_e (EAp e1 e2) = EAp (separateLams_e e1) (separateLams_e e2)
> separateLams_e (ECase e alts)
>  = ECase (separateLams_e e) [ (tag, args, separateLams_e e)
>                             | (tag, args, e) <- alts
>                             ]
>
> separateLams_e (ELam args body) = mkSepLams args (separateLams_e body)
```

```
>
> separateLams_e (ELet is_rec defns body)
>  = ELet is_rec [(name, separateLams_e rhs) | (name,rhs) <- defns]
>               (separateLams_e body)

> mkSepLams args body = foldr mkSepLam body args
>                       where mkSepLam arg body = ELam [arg] body
```

Now we return to the top-level function `separateLams`. The interesting question is what to do about supercombinator definitions. The easiest thing to do is to turn them into the equivalent lambda abstractions!

```
> separateLams prog = [ (name, [], mkSepLams args (separateLams_e rhs))
>                     | (name, args, rhs) <- prog
>                     ]
```

6.6.6 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 expression 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 :: ASSOC Name Level -> Set Name -> Level
> freeSetToLevel env free
>  = foldll max 0 [aLookup env n 0 | n <- setToList free]
>    -- 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
>                  ASSOC Name Level          -> -- Level of in-scope names
>                  AnnExpr Name (Set Name)  -> -- Input expression
>                  AnnExpr (Name, Level) Level -- Result expression
```

We represent the name-to-level mapping as an association list, with type `assoc name level`. The interface of association lists is given in Appendix A, 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.

Now we can define `freeToLevel`, using an auxiliary function to process each supercombinator definition. Remember that `separateLams` has removed all the arguments from supercombinator definitions:

```
> freeToLevel prog = map freeToLevel_sc prog
>
> freeToLevel_sc (sc_name, [], rhs) = (sc_name, [], freeToLevel_e 0 [] rhs)
```

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 (free, ANum k) = (0, ANum k)
> freeToLevel_e level env (free, AVar v) = (aLookup env v 0, AVar v)


> freeToLevel_e level env (free, AConstr t a) = (0, AConstr t a)


> freeToLevel_e level env (free, AAp e1 e2)
>  = (max (levelOf e1') (levelOf e2'), AAp e1' e2')
>     where
>     e1' = freeToLevel_e level env e1
>     e2' = freeToLevel_e level env e2
```

The same cannot be done for lambda abstractions; so we must 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 (Section 6.7.3).

```
> freeToLevel_e level env (free, ALam args body)
>  = (freeSetToLevel env free, ALam args' body')
>    where
>    body' = freeToLevel_e (level + 1) (args' ++ env) body
>    args' = [(arg, level+1) | arg <- args]
```

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

```
> freeToLevel_e level env (free, ALet is_rec defns body)
>  = (levelOf new_body, ALet is_rec new_defns new_body)
>    where
>    binders = bindersOf defns
>    rhss = rhssOf defns
>
>    new_binders = [(name,max_rhs_level) | name <- binders]
>    new_rhss = map (freeToLevel_e level rhs_env) rhss
>    new_defns = zip2 new_binders new_rhss
>    new_body = freeToLevel_e level body_env body
>
>    free_in_rhss = setUnionList [free | (free,rhs) <- rhss]
>    max_rhs_level = freeSetToLevel level_rhs_env free_in_rhss
>
>    body_env      = new_binders ++ env
>    rhs_env | is_rec          = body_env
>            | otherwise       = env
>    level_rhs_env |  is_rec    = [(name,0) | name <- binders] ++ env
>                  |  otherwise = 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; the dependency analysis which we look at in the next section will eliminate such definitions.

case expressions are deferred:

```
> freeToLevel_e level env (free, ACase e alts)
>  = freeToLevel_case level env free e alts
> freeToLevel_case free e alts = error "freeToLevel_case: not yet written"
```

Lastly the auxiliary functions levelOf extracts the level from an expression:

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

6.6.7 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                               -- Level of context
>                   -> AnnExpr (Name, Level) Level      -- Input expression
>                   -> Expr (Name, Level)               -- Result
```

```
> identifyMFEs prog = [ (sc_name, [], identifyMFEs_e 0 rhs)
>                     | (sc_name, [], rhs) <- prog
>                     ]
```

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 in Section 6.7.3. We encode this knowledge of which MFEs deserve special treatment in a function `notMFECandidate`.

```
> notMFECandidate (AConstr t a) = True
> notMFECandidate (ANum k)      = True
> notMFECandidate (AVar v)      = True
> notMFECandidate ae            = 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 sub-expressions; otherwise we use `transformMFE` to perform the appropriate transformation.

```
> identifyMFEs_e cxt (level, e)
>  | level == cxt || notMFECandidate e = e'
>  | otherwise = 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.

```
> identifyMFEs_e1 :: Level                                 -- Level of context
>                 -> AnnExpr' (Name,Level) Level     -- Input expressions
>                 -> Expr (Name,Level)               -- Result expression
```

```
> identifyMFEs_e1 level (AConstr t a)          = EConstr t a
> identifyMFEs_e1 level (ANum n)               = ENum n
> identifyMFEs_e1 level (AVar v)               = EVar v
> identifyMFEs_e1 level (AAp e1 e2)
>  = EAp (identifyMFEs_e level e1) (identifyMFEs_e level e2)
```

When `identifyMFEs_e1` encounters a binder it changes the 'current' level number carried down as its first argument, as we can see in the equations for lambda abstractions and `let(rec)` expressions:

```
> identifyMFEs_e1 level (ALam args body)
>  = ELam args (identifyMFEs_e arg_level body)
>    where
>    (name, arg_level) = hd args

> identifyMFEs_e1 level (ALet is_rec defns body)
>  = ELet is_rec defns' body'
>    where
>    body' = identifyMFEs_e level body
>    defns' = [ ((name, rhs_level), identifyMFEs_e rhs_level rhs)
>             | ((name, rhs_level), rhs) <- defns
>             ]
```

`case` expressions are deferred:

```
> identifyMFEs_e1 level (ACase e alts) = identifyMFEs_case1 level e alts
> identifyMFEs_case1 level e alts = error "identifyMFEs_case1: not written"
```

6.6.8 Renaming variables

As we remarked above, it would be nice to use the existing `rename` function to make the binders unique, but it has the wrong type. It would be possible to write `renameL` by making a copy of `rename` and making some small alterations, but it would be much nicer to make a single generic renaming function, `renameGen`, which can be specialised to do either `rename` or `renameL`.

What should the type of `renameGen` be? The right question to ask is: *'what use did we make in* `rename` *of the fact that each binder was a simple* `name`*?'* or, alternatively, 'what operations did we perform on binders in `rename`?'.

There is actually just one such operation, which constructs new binders. In `rename_e` this function is called `newNames`; it takes a name supply and a list of names, and returns a depleted name supply, a list of new names and an association list mapping old names to new ones:

```
> newNames :: nameSupply -> [name]  -> (nameSupply, [name], assoc name name)
```

Since `renameGen` must be able to work over any kind of binder, not just those of type `name`, *we must pass the new-binders function into* `renameGen` *as an extra argument.* So the type of `renameGen` is:

```
> renameGen :: (NameSupply -> [a]  -> (NameSupply, [a], ASSOC Name Name))
>                                      -- New-binders function
>              -> Program a            -- Program to be renamed
>              -> Program a            -- Resulting program
```

Notice that the type of the binders is denoted by the type variable `*`, because `renameGen` is polymorphic in this type. Using `renameGen`, we can now redefine the original `rename` function, by passing `newNames` to `renameGen` as the new-binders function.

```
> rename :: CoreProgram -> CoreProgram
> rename prog = renameGen newNames prog
```

renameL is rather more interesting. Its binders are (name,level) pairs so we need to define a different new-binders function:

```
> renameL :: Program (Name,Level) -> Program (Name,Level)
> renameL prog = renameGen newNamesL prog
```

The function newNamesL does just what newNames does, but it does it for binders whose type is a (name,level) pair:

```
> newNamesL ns old_binders
>  = (ns', new_binders, env)
>    where
>    old_names = [name | (name,level) <- old_binders]
>    levels        = [level | (name,level) <- old_binders]
>    (ns', new_names) = getNames ns old_names
>    new_binders =  zip2 new_names levels
>    env = zip2 old_names new_names
```

Now we can turn our attention to writing renameGen. As usual we need an auxiliary function renameGen_e which carries around some extra administrative information. Specifically, like rename_e, it needs to take a name supply and old-name to new-name mapping as arguments, and return a depleted supply as part of its result. It also needs to be passed the new-binders function:

```
> renameGen_e :: (NameSupply -> [a]  -> (NameSupply, [a], ASSOC Name Name))
>                                               -- New-binders function
>                -> ASSOC Name Name             -- Maps old names to new ones
>                -> NameSupply                  -- Name supply
>                -> Expr a                      -- Expression to be renamed
>                -> (NameSupply, Expr a)        -- Depleted name supply
>                                               -- and result expression
```

Using renameGen_e we can now write renameGen. Just like rename, renameGen applies a local function rename_sc to each supercombinator definition.

```
> renameGen new_binders prog
>  = second (mapAccuml rename_sc initialNameSupply prog)
>     where
>     rename_sc ns (sc_name, args, rhs)
>      = (ns2, (sc_name, args', rhs'))
>        where
>       (ns1, args', env) = new_binders ns args
>       (ns2, rhs') = renameGen_e new_binders env ns1 rhs
```

Exercise 6.9. Write the function `renameGen_e`. It is very like `rename_e`, except that it takes the binder-manipulation functions as extra arguments. In the equations for `ELet`, `ELam` and `ECase` (which each bind new variables), the function `newBinders` can be used in just the same way as it is in `rename_sc` above.

Test your definition by checking that the simple lambda lifter still works with the new definition of `rename`.

Exercise 6.10. The type signature we wrote for `renameL` is actually slightly more restrictive than it need be. How could it be made more general (without changing the code at all)? Hint: what use does `renameL` make of the fact that the second component of a binder is of type `level`?

This section provides a good illustration of the way in which higher-order functions can help us to make programs more modular.

6.6.9 Floating `let(rec)` expressions

The final pass floats `let(rec)` expressions out to the appropriate level. The auxiliary function, which works over expressions, has to return an expression together with the collection of definitions which should be floated outside the expression.

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

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, [(Name, Expr 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.*

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

```
> float_e (EVar v) = ([], EVar v)
> float_e (EConstr t a) = ([], EConstr t a)
> float_e (ENum n) = ([], ENum n)
> 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 (recall from Section 6.6.6 that all variables bound by a single `ELam` construct are given the same level):

```
> float_e (ELam args body)
>  = (fd_outer, ELam args' (install fd_this_level body'))
>    where
>    args' = [arg | (arg,level) <- args]
>    (first_arg,this_level) = hd args
>    (fd_body, body') = float_e body
>    (fd_outer, fd_this_level) = partitionFloats this_level fd_body
```

The equation 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 is_rec defns body)
>  = (rhsFloatDefns ++ [thisGroup] ++ bodyFloatDefns, body')
>    where
>    (bodyFloatDefns, body') = float_e body
>    (rhsFloatDefns, defns') = mapAccuml float_defn [] defns
>    thisGroup = (thisLevel, is_rec, defns')
>    (name,thisLevel) = hd (bindersOf defns)
>
>    float_defn floatedDefns ((name,level), rhs)
>     = (rhsFloatDefns ++ floatedDefns, (name, rhs'))
>       where
>       (rhsFloatDefns, rhs') = float_e rhs
```

We defer `case` expressions:

```
> float_e (ECase e alts) = float_case e alts
> float_case e alts = error "float_case: not yet written"
```

The auxiliary function `partitionFloats` takes a `floatedDefns` and a level number, and separates it into two: those belonging to an outer level and those belonging to the specified level (or an inner one):

```
> partitionFloats :: Level -> FloatedDefns -> (FloatedDefns, FloatedDefns)
> partitionFloats this_level fds
>  = (filter is_outer_level fds, filter is_this_level fds)
>    where
>    is_this_level  (level,is_rec,defns) = level >= this_level
>    is_outer_level (level,is_rec,defns) = level <  this_level
```

The function `install` wraps an expression in a nested set of `let(rec)`s containing the specified definitions:

```
> install :: FloatedDefns -> Expr Name -> Expr Name
> install defnGroups e
>  = foldr installGroup e defnGroups
>    where
>    installGroup (level, is_rec, defns) e = ELet is_rec defns e
```

Finally, we can define the top-level function, `float`. It uses `float_sc` to apply `float_e` to each supercombinator, yielding a list of supercombinators, in just the same way as `collectSCs` above.

```
> float prog = concat (map float_sc prog)
```

The function `float_sc` takes a supercombinator definition to a list of supercombinator definitions, consisting of the transformed version of the original definition together with the level-zero definitions floated out from its body:

```
> float_sc (name, [], rhs)
>  = [(name, [], rhs')] ++ concat (map to_scs fds)
>    where
>    (fds, rhs') = float_e rhs
>    to_scs (level, is_rec, defns) = map make_sc defns
>    make_sc (name, rhs) = (name, [], rhs)
```

The top level of a program is implicitly mutually recursive, so we can drop the `isRec` flags. We also have to give each floated definition an empty argument list, since it is now a supercombinator definition.

6.7 Mark 5: Improvements to full laziness

That completes the definition of the fully lazy lambda lifter. Its output is always correct, but it is larger and less efficient than it need be. In this section we discuss some ways to improve the full laziness transformation.

6.7.1 Adding `case` expressions

Exercise 6.11. Write definitions for `freeToLevel_case`, `identifyMFEs_case1` and `float_case`. All of them work in an analogous way to lambda abstractions. Hint: in `float_case` take care with alternatives whose argument list is empty.

6.7.2 Eliminating redundant supercombinators

Consider the Core-language expression

```
\x. \y. x+y
```

Here the \y-abstraction has no MFEs apart from `x` itself, so the full-laziness pass will not affect the expression at all. Unfortunately, the simple lambda lifter, `lambdaLift`, will then generate *two* supercombinators, one for each lambda, whereas only one is needed. It would be better to combine nested `ELam` expressions into a single `ELam` before passing the program to `lambdaLift`, so that the latter would then generate just one supercombinator. We could do this in a separate pass, but it saves work to do it as part of the work of `float`.

Exercise 6.12. Modify the definition of the `ELam` case of `float` so that it combines nested `ELam` constructors. Hint: make use of the function:

```
> mkELam :: [Name] -> CoreExpr -> CoreExpr
> mkELam args (ELam args' body) = ELam (args++args') body
> mkELam args other_body        = ELam args          other_body
```

6.7.3 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 may turn into 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 sub-expressions 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 1985] and [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.

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 allow `ELam` constructors to take a list of binders.

- `identifyMFEs` uses a predicate `notMFECandidate` to decide whether to identify a particular sub-expression 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.

This concludes the presentation of the full laziness transformation.

6.8 Mark 6: Dependency analysis†

Consider the Core-language definition

```
f x = let
        g = \y. letrec
                  h = y+1 ;
                  k = x+2
                in
                h+k
      in
      g 4
```

The inner `letrec` is not recursive at all! The program is equivalent to the following:

```
f x = let
        g = \y. let h = y+1 in
                let k = x+2 in
                h+k
      in
      g 4
```

This transformation, which breaks up `let(rec)` blocks into minimal-sized groups, and always uses `let` in preference to `letrec`, is called *dependency analysis*.

We have already alluded to the fact that better code can be generated if the program has been subjected to a dependency analysis. This has shown up in two places:

- In Johnsson-style lambda lifting, we treated the free variables for a `letrec` block of definitions as a single entity. If we could in fact break the `letrec` up into smaller blocks, then there would be fewer free variables in each block. This will then reduce the number of free variables that must be added to a function definition by the `abstractJ` function.
- In the full-laziness transformation we always kept the declarations in a `let(rec)` block together. If we first do dependency analysis, to break the declarations into small groups, then perhaps some of the groups could be floated further out than before.

This section explores how to do dependency analysis on Core programs.

6.8.1 Strongly connected components

In order to discuss the dependency analysis we need to understand some graph theory. We therefore begin with a definition.

Definition 6.1 *A directed graph is a tuple $(V,\ E)$ of two components:*

- *a set of vertices (or nodes), V;*
- *a set of edges, E. Each edge is a pair: $(v_0,\ v_1)$; where $v_0 \in V$ is the source of the edge, and $v_1 \in V$ is the target.*

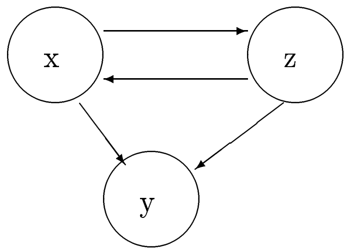
In the following expression we say that `x` depends on `y` and `z`, `z` depends on `x` and `y`, but that `y` does not depend on any other variable.

```
letrec
    x = y + 7 * tl z;
    y = 5
    z = (x,y)
in e
```

The graph we construct for this definition block has three vertices, $\{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$, and the edges:

$$\{(\mathtt{x},\mathtt{y}),\ (\mathtt{x},\mathtt{z}),\ (\mathtt{z},\mathtt{x}),\ (\mathtt{z},\mathtt{y})\}$$

The interpretation of the first edge is that `x` *depends* on `y`, and the absence of any edges from `y` means that `y` does not depend on any other variables. Pictorially, the graph is as follows:



Because we are not concerned with multiple edges between the same pairs of vertices, we can instead formulate the information about the edges as a map: *outs*.

Definition 6.2

$$\mathit{outs}\ v = \{v' \mid (v,\ v') \in E\}$$

The set *outs* v is therefore the set of vertices that are targets for an edge whose source is v. In the example we have just looked at *outs* $\mathtt{x} = \{\mathtt{y}, \mathtt{z}\}$, *outs* $\mathtt{y} = \{\}$ and *outs* $\mathtt{z} = \{\mathtt{x}, \mathtt{y}\}$.

We can construct a similar map, *ins*, which is dual to the *outs* map.

Definition 6.3

$$\mathit{ins}\ v = \{v' \mid (v',\ v) \in E\}$$

This is the set of vertices that are the source of an edge pointing to v. In the example we have just looked at $\mathit{ins}\ \mathtt{x} = \{\mathtt{z}\}$, $\mathit{ins}\mathtt{y} = \{\mathtt{x}, \mathtt{z}\}$ and $\mathit{ins}\mathtt{z} = \{\mathtt{x}\}$.

Definition 6.4 *The map r^* is the* transitive closure *of the map r. It can be defined recursively as:*

- $a \in r^*\ a$,
- *if $b \in r^*\ a$, then $r\ b \subseteq r^*\ a$.*

The set $\mathit{outs}^*\ a$ is the set of all vertices that can be reached from the vertex a, by following the edges of the graph. We say that b is *reachable* from a whenever $b \in \mathit{outs}^*\ a$. The set $\mathit{ins}^*\ a$ is the set of all vertices that can reach the vertex a by following the edges of the graph.

Using the running example, we have $\mathit{outs}^*\ \mathtt{x} = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$, $\mathit{outs}^*\ \mathtt{y} = \{\mathtt{y}\}$, and $\mathit{outs}^*\ \mathtt{z} = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$. We also have $\mathit{ins}^*\ \mathtt{x} = \{\mathtt{x}, \mathtt{z}\}$, $\mathit{ins}^*\ \mathtt{y} = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$, and $\mathit{ins}^*\ \mathtt{z} = \{\mathtt{x}, \mathtt{z}\}$.

We are now in a position to define a strongly connected component of a graph. Informally, the vertices a and b are in the same component whenever a is reachable from b and b is reachable from a.

Definition 6.5 *The strongly connected component of the graph containing the vertex a is the set of vertices $\mathit{scc}\ a$, defined as:*

$$\mathit{scc}\ a = \mathit{outs}^*\ a \cap \mathit{ins}^*\ a$$

That is: a vertex b is in $\mathit{scc}\ a$ if and only if it is reachable from a and if a is reachable from b. The graph in the running example has two strongly connected components: $\{\mathtt{x}, \mathtt{z}\}$ and $\{\mathtt{y}\}$.

Exercise 6.13. Prove that the relation 'in the same strongly connected component' partitions the set of vertices into equivalence classes.

Topological sorting

An ordering on vertices can be induced by considering the maps ins^* and outs^*. This is a partial order which we will represent as $\preceq$.

Definition 6.6 *The vertex a is topologically less than or equal to the vertex b, written $a \preceq b$, if and only if:*

$$b \in \mathit{outs}^*\ a$$

In our example $\mathtt{x} \preceq \mathtt{x}$, $\mathtt{x} \preceq \mathtt{y}$, $\mathtt{x} \preceq \mathtt{z}$, $\mathtt{y} \preceq \mathtt{y}$, $\mathtt{z} \preceq \mathtt{x}$, $\mathtt{z} \preceq \mathtt{y}$ and $\mathtt{z} \preceq \mathtt{z}$.

Because the strongly connected components are disjoint, a similar ordering is induced on them by considering the action of $\preceq$ on their elements. In our running example the two components are ordered as: $\{\mathtt{x}, \mathtt{z}\} \preceq \{\mathtt{y}\}$.

Definition 6.7 *A sequence of vertices (or strongly connected components) are topologically sorted if, for all a and b in the sequence, a precedes b whenever $a \preceq b$.*

So, one possible topologically sorted sequence would be: [x, z, y]. The other is [z, x, y].

Having now determined that the two strongly connected components are ordered as $\{\mathtt{x}, \mathtt{z}\} \preceq \{\mathtt{y}\}$, we may transform the original expression into the following one:

```
letrec
    y = 5
in letrec
       x = y + 7 + second z
       z = (x, y)
   in e
```

We next consider efficient ways to implement the strongly connected component algorithm.

6.8.2 Implementing a strongly connected component algorithm

Depth first search

We will first consider the problem of implementing a depth first search of a graph. The function `depthFirstSearch` is parameterised over the map from vertices to their offspring; this permits us to reverse the direction in which the edges are traversed. Furthermore, we choose to make the maps *ins* and *outs* into functions from vertices to sequences of vertices. The reason for this change is that we have to traverse the offspring in some order and this is easier to arrange if we have a sequence rather than a set.

```
> depthFirstSearch :: Ord a =>
>                     (a -> [a])   -> -- Map
>                     (Set a, [a]) -> -- State: visited set,
>                                     --        current sequence of vertices
>                     [a]          -> -- Input vertices sequence
>                     (Set a, [a])    -- Final state
```

The function `depthFirstSearch` updates a state as it runs down the input vertex sequence, using `foldll`. The state consists of two parts: in the first we keep track of all of the vertices that we have so far visited; in the second we construct the sequence of vertices that we will output.

```
> depthFirstSearch
>  = foldll . search
>    where
```

The key part of the depth first search is coded into `search`: if we have already visited the vertex then the state is unchanged.

```
>    search relation (visited, sequence) vertex
>     | setElementOf vertex visited = (visited,          sequence ) -- KH
>     -- KH Was: = (visited,          sequence ), setElementOf vertex visited
```

On the other hand, if this is the first time we have visited the vertex, then we must proceed to search from this vertex. When the state is returned from this search, we must add the current vertex to the sequence.

```
>     | otherwise = (visited', vertex: sequence') -- KH
>     -- KH Was: = (visited', vertex: sequence'), otherwise
>       where
```

The visited set must be updated with the current vertex, before we begin the search, and the list of vertices to search from is determined by applying the map `relation`.

```
>       (visited', sequence')
>        = depthFirstSearch relation
>                           (setUnion visited (setSingleton vertex), sequence)
>                           (relation vertex)
```

The result will be a set of vertices visited, and a sequence of these visited vertices, in topological sort order.

Exercise 6.14. Prove the following theorem:

$$\texttt{depthFirstSearch}\ \mathit{outs}\ (\{\},\ [])S = (V,\ S')$$

where S is a sequence of vertices, $V = \bigcup_{v \in S} \mathit{outs}\ v$, and S' is a topologically sorted sequence of the vertices in V.

Spanning search

The function `spanningSearch` is a slight adaptation of the function `depthFirstSearch`, in which we retain the structuring information obtained during the search.

```
> spanningSearch   :: Ord a =>
>                     (a -> [a])       -> -- The map
>                     (Set a, [Set a]) -> -- Current state: visited set,
>                                         --  current sequence of vertice sets
>                     [a]              -> -- Input sequence of vertices
>                     (Set a, [Set a])    -- Final state
```

Again, it is defined in terms of an auxiliary function `search`

```
> spanningSearch

>  = foldll . search
>    where
```

If the current vertex has been visited already then we return the current state.

```
>    search relation (visited, setSequence) vertex
>     | setElementOf vertex visited = (visited,         setSequence ) -- KH
>     -- KH Was: = (visited,         setSequence ), setElementOf vertex visited
```

Alternatively, if this is the first time we have visited the current vertex, then we search – using `depthFirstSearch` – from the current vertex. The sequence that is returned constitutes the component associated with the current vertex. We therefore add it to the sequence of sets that we are constructing.

```
>     | otherwise = (visited', setFromList (vertex: sequence): setSequence) -- KH
>     -- KH Was: = (visited', setFromList (vertex: sequence): setSequence)
>       where
>       (visited', sequence)
>        = depthFirstSearch relation
>                            (setUnion visited (setSingleton vertex), [])
>                            (relation vertex)
```

Strongly connected components

The strongly connected component algorithm can now be implemented as the following function:

```
> scc :: Ord a =>
>        (a -> [a]) -> -- The "ins"  map
>        (a -> [a]) -> -- The "outs" map
>        [a]        -> -- The root vertices
>        [Set a]       -- The topologically sorted components
```

The function `scc` consists of two passes over the graph.

```
> scc ins outs
>  = spanning . depthFirst
```

In the first we construct a topologically sorted sequence of the vertices of the graph.

```
>    where depthFirst = second . depthFirstSearch outs (setEmpty, [])
```

In the second pass we construct the reverse of the topologically sorted sequence of strongly connected components.

```
>          spanning   = second . spanningSearch   ins  (setEmpty, [])
```

Let us consider what happens when we construct the first component. At the head of the sequence is the vertex `a`. Any other vertices that satisfy $\mathtt{a} \preceq \mathtt{b}$ will occur later in the sequence. There are no vertices satisfying $\mathtt{b} \preceq \mathtt{a}$. The call to `spanningSearch` with `ins` as its relational parameter, will construct *ins** `a`. The visited set will be augmented with each vertex in the component.

In the example we considered earlier, we will be applying `spanning` to the list [`x`, `z`, `y`]. This expands to:

```
second (search ins (search ins (search ins ({},[]) x) z) y)
```

Expanding the inner `search` we obtain:

```
second (search ins (search ins (vs, [setFromList (x:s)]) z) y)
where (vs,s) = depthFirstSearch ins ({x},[]) (ins x)
```

But `ins x` $=$ [z]; this means that `vs` $= \{$`x, z`$\}$ and `s` $=$ [z]. Hence we reduce the expression to:

```
second (search ins (search ins ({x,z}, [{x,z}]) z) y)
```

Because `z` is already in the visted set, this becomes:

```
second (search ins ({x,z}, [{x,z}]) y)
```

The `search` expands as:

```
second (vs, [setFromList (y:s), {x,z}])
where (vs,s) = depthFirstSearch ins ({x,y,z},[]) (ins y)
```

But `ins y` $=$ [`x, z`]; both of which are already visited, so `vs` $= \{$`x, z, y`$\}$ and $s = []$.

The final form of the expression becomes:

```
[{y}, {x,z}]
```

> *The visited set in* `spanningSearch` *represents those vertices that have already been assigned to a strongly connected component.*

When we come across a vertex in the input sequence that is already in the visited set, it behaves as if it had been deleted from further consideration. Suppose that the vertex `b` is the next vertex in the input sequence that has not already been visited. When we come to compute

$$\texttt{depthFirstSearch}\ \mathit{ins}\ (\{\texttt{b}\} \cup \texttt{visited}, [])\ (\mathit{ins}\ \texttt{b})$$

This will produce a new visited set (which will be scc `a` $\cup$ scc `b`) and a sequence whose elements are the vertices in scc `b`.

Note that the strongly connected components are output in reverse topological order.

Exercise 6.15. Let

$$\texttt{scc}\ \mathit{ins}\ \mathit{outs}\ R = S,$$

and $V = \bigcup\{\mathit{ins}^*\ v \mid v \in R\}$. Prove that

- firstly, that the sequence S contains all of the strongly connected components, i.e.

$$\texttt{setFromList}\ S = \bigcup\{\mathit{scc}\ v \mid v \in V\};$$

- secondly, that these components are in reverse topological order, i.e. if `a` $\preceq$ `b` then `b` occurs before `a` in the sequence S.

6.8.3 A dependency analysis

We can now perform a dependency analysis on the program. Whenever we come across a `let(rec)` block we must split it into strongly connected components. In the case of `let`, this is simple; there are no dependencies so we simply separate the list of definitions into a singleton list for each definition.

The dependency analysis is performed by the function `dependency`. This uses information from a prior pass of `freeVars` to rearrange `let(rec)`s; this information is then used by the auxiliary function `depends`.

```
> dependency :: CoreProgram -> CoreProgram
> dependency =  depends . freeVars
```

```
> runD = pprint . dependency . parse
```

```
> depends :: AnnProgram Name (Set Name) -> CoreProgram
> depends prog = [(name,args, depends_e rhs) | (name, args, rhs) <- prog]
```

The work is done by `depends_e`, whose only interesting case is that for `let(rec)`.

```
> depends_e :: AnnExpr Name (Set Name) -> CoreExpr
> depends_e (free, ANum n)          = ENum n
> depends_e (free, AConstr t a)     = EConstr t a
> depends_e (free, AVar v)          = EVar v
> depends_e (free, AAp e1 e2)       = EAp (depends_e e1) (depends_e e2)
> depends_e (free, ACase body alts) = ECase (depends_e body)
>                                           [ (tag, args, depends_e e)
>                                           | (tag, args, e) <- alts
>                                           ]
> depends_e (free, ALam ns body)    = ELam ns (depends_e body)
```

In the case of `letrec`s we must construct the dependency graph, and then apply the `scc` function to determine the way to split the definitions. If `defnGroups` is $[d_1, d_2, \ldots, d_n]$ – and we are processing a `letrec` – then the `letrec` will be transformed to:

`letrec` d_1 `in letrec` d_2 `in` $\ldots$ `letrec` d_n `in body`.

```
> depends_e (free, ALet is_rec defns body)
>  = foldr (mkDependLet is_rec) (depends_e body) defnGroups
>    where
>    binders    = bindersOf defns
```

The set of variables that we are interested in is derived from the `binders`, and is called the `binderSet`.

```
>    binderSet | is_rec    = setFromList binders
>              | otherwise = setEmpty
```

From this we can construct the edges of the dependency graph.

```
>    edges      = [(n, f) | (n, (free, e)) <- defns,
>                           f <- setToList (setIntersection free binderSet)]
```

And thus the functions ins and outs required by the strongly connected component algorithm.

```
>    ins  v     = [u | (u,w) <- edges, v==w]
>    outs v     = [w | (u,w) <- edges, v==u]
```

The resulting list of sets is converted into a list of lists, called components.

```
>    components = map setToList (scc ins outs binders)
```

We construct the defnGroups by looking up the expression bound to each binder in the original definitions, defns:

```
>    defnGroups = [ [ (n, aLookup defns n (error "defnGroups"))
>                     | n <- ns]
>                   | ns <- components
>                   ]
```

Finally, to join together each group in defnGroups, we define mkDependLet, which recursively does dependency analysis on each right-hand side, and then builds the results into a let(rec) expression: A simple definition is:

```
> mkDependLet is_rec dfs e = ELet is_rec [(n, depends_e e) | (n,e) <- dfs] e
```

Exercise 6.16. In addition to the definition groups from non-recursive lets we sometimes get non-recursive definitions arising in letrec's. This is the case for y in the example we have used.

Redefine mkDependLet to make such bindings with a let and not a letrec. Hint: use the free-variable information present in the right-hand sides of the definitions passed to mkDependLet.

6.9 Conclusion

It is interesting to compare our approach to full laziness 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*. For example:

- We were able to reuse existing functions on several occasions (`freeVars`, `rename`, `collectSCs` and so on).
- The multi-pass approach means that each pass has a well-defined, simple purpose, which makes it easier to modify. For example, we modified the `identifyMFEs` algorithm to be more selective about where full laziness is introduced (Section 6.7.3).
- We could use the major phases in various combinations to 'snap together' a variety of transformations. For example, we could choose whether or not to do dependency analysis and full laziness, and which lambda lifter to use, simply by composing the appropriate functions at the top level.

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 (Section 6.7.3). Even so, matters might not have fallen out so fortuitously, and our separation of concerns has certainly made some sorts of transformation rather difficult.