

## Chapter 5

# Implementing the Lambda Calculus

This chapter provides the machinery to translate a program in applied lambda calculus (aka MiniCaml) into a program scheme where all functions are top-level functions. The language of program schemes is already considerably simplified and implementing a program scheme means to implement the lambda calculus because our transformation preserves the meaning of the program.

As simple as an applicative recursive program scheme is, it is still very different from the machine language of real-world processors. Implementing it still requires a number of intermediate steps until it is suitable for native code generation.

The first step is to name all intermediate results that must end up in registers. This transformation step introduces *let* expressions with fresh names for all those intermediate results. The second step is to make the sequence of computation step explicit by flattening the *let* expressions. The third step introduces control flow management.

The result is a program in a special form called *A-normal form*. It is well suited to various compile-time transformations as demonstrated with the example of the inlining transformation.

For illustration, a program for computing the *n*th power of a base value serves as a running example.

```
power x =  
  letrec g = lambda n. if n=0 then 1 else x * g (n-1) in  
  g
```

### 5.1 Lambda lifting

While locally defined functions make programs succinct, they complicate their semantic analysis and their mapping to memory structures. Hence, we strive to transform programs so that all functions are defined in one giant *letrec* at the toplevel and their only free variables are references to other toplevel functions. Such a recursive applicative program schema is much easier to implement because each toplevel function corresponds to a code pointer.

The transformation that lifts all functions to toplevel is called *lambda lifting*.

A good way to understand lambda lifting is to look at a few examples first and develop the general strategy from the insights gained on the way.

### 5.1.1 Strategies for removing *letrec*

Consider the following term involving *letrec* (ignoring for a moment that it does not make much sense operationally):

$$\text{let } i = 5 \text{ in } \text{letrec } f = \lambda x.f (+ i i) \text{ in } f (* i i)$$

Since this term involves “interior” recursion, and vanilla program schemes only provide top-level recursion, the basic approach to remove the *letrec* must be to “lift”  $f$  to the top level:

$$\begin{aligned} \text{letrec } f &= \lambda x.f (+ i i) \text{ in} \\ \text{let } i &= 5 \text{ in } f (* i i) \end{aligned}$$

Unfortunately, the resulting program does not work because  $\lambda x.f (+ i i)$  has a free variable  $i$ , and the lifting has removed the term from the scope of  $i$ . Therefore, it is necessary to pass  $i$  as an additional parameter to  $f$ :

$$\begin{aligned} \text{letrec } f &= \lambda i.\lambda x.f i (+ i i) \text{ in} \\ \text{let } i &= 5 \text{ in } f i (* i i) \end{aligned}$$

As an afterthought, it may have been more systematic to add the additional parameters first, and then lift them to the top level, thus avoiding an incorrect program at an intermediate stage:

$$\begin{aligned} \text{let } i &= 5 \text{ in } \text{letrec } f = \lambda x.f (+ i i) \text{ in } f (* i i) \\ \implies \\ \text{let } i &= 5 \text{ in } \text{letrec } f = \lambda i.\lambda x.f i (+ i i) \text{ in } f i (* i i) \\ \implies \\ \text{letrec } f &= \lambda i.\lambda x.f i (+ i i) \text{ in} \\ \text{let } i &= 5 \text{ in } f i (* i i) \end{aligned}$$

In the presence of mutual recursion, it is not always sufficient to abstract over all free variables of a *letrec*-bound term. Consider the following, somewhat more involved example:

$$\begin{aligned} \text{let } a &= \dots \text{ in} \\ \text{let } b &= \dots \text{ in} \\ \text{letrec } f &= \lambda x.\dots a \dots g \dots \\ \text{and } g &= \lambda y.\dots b \dots f \dots \\ \text{in } \dots f \dots g \dots \end{aligned}$$

Here,  $a$  is free in  $f$ , and  $b$  is free in  $g$ . A naive application of the parameter expansion strategy produces the following result:

$$\begin{aligned} \text{let } a &= \dots \text{ in} \\ \text{let } b &= \dots \text{ in} \\ \text{letrec } f &= \lambda a.\lambda x.\dots a \dots g b \dots \\ \text{and } g &= \lambda b.\lambda y.\dots b \dots f a \dots \\ \text{in } \dots f a \dots g b \dots \end{aligned}$$

The result still has the same meaning as the original, but lifting is not possible yet: The parameter expansion has introduced new free variables in the bodies of  $f$  and  $g$ :  $b$  is now free in  $f$ , and  $a$  is free in  $g$ . Consequently, another application of the expansion step is necessary:

$$\begin{aligned} \text{let } a &= \dots \text{ in} \\ \text{let } b &= \dots \text{ in} \\ \text{letrec } f &= \lambda b.\lambda a.\lambda x.\dots a \dots g a b \dots \\ \text{and } g &= \lambda a.\lambda b.\lambda y.\dots b \dots f b a \dots \\ \text{in } \dots f b a \dots g a b \dots \end{aligned}$$

Now, finally, the bodies both contain no more free variables, and lifting is possible:

$$\begin{aligned} \text{letrec } f &= \lambda b.\lambda a.\lambda x.\dots a\dots g a b\dots \\ \text{and } g &= \lambda a.\lambda b.\lambda y.\dots b\dots f b a\dots \text{ in} \\ \text{let } a &= \dots \text{ in} \\ \text{let } b &= \dots \text{ in} \\ \dots f b a\dots g a b\dots \end{aligned}$$

Thus, repeatedly abstracting the free variables from the letrec-bound functions and applying them to their values results in an expression where all letrec-bound functions are closed. The sequence of abstraction steps terminates because there are only finitely many bound variables in a program and the transformation does not introduce new bound variables.

Performing this step repeatedly in a compiler is a costly matter just to cater to such a non-consequential language construct. The obvious idea to avoid the repetition is to compute the free variables of all functions bound in one letrec at once. However, there is one caveat which we illustrate with a small modification of the last example. In this variation, function  $g$  does not invoke  $f$  but  $f$  invokes itself:

$$\begin{aligned} \text{let } a &= \dots \text{ in} \\ \text{let } b &= \dots \text{ in} \\ \text{letrec } f &= \lambda x.\dots a\dots g\dots f\dots \\ \text{and } g &= \lambda y.\dots b\dots \\ \text{in } \dots f\dots g\dots \end{aligned}$$

But now, the free variables of  $f$  and  $g$  are  $a$  and  $b$  and willy-nilly application of the abstraction step would lead to:

$$\begin{aligned} \text{let } a &= \dots \text{ in} \\ \text{let } b &= \dots \text{ in} \\ \text{letrec } f &= \lambda a.\lambda b.\lambda x.\dots a\dots g a b\dots f a b\dots \\ \text{and } g &= \lambda a.\lambda b.\lambda y.\dots b\dots \\ \text{in } \dots f a b\dots g a b\dots \end{aligned}$$

Thus, the code now passes  $a$  to  $g$  although  $g$  does not require  $a$  at all. The cause of this redundancy is the overly generous use of the letrec, where the functions in it are not all mutually dependent on each other. Rectifying this problem requires analyzing the dependency graph with nodes the set of letrec-defined functions and an edge from  $f$  to  $g$  if  $g$  occurs free in  $f$ 's defining expression. Each strongly connected component in this graph is collected in its own letrec with the dependences between the components determining the overall order of the letrecs.

In the example, the components are the singletons  $\{f\}$  and  $\{g\}$  so that the program first transforms into:

$$\begin{aligned} \text{let } a &= \dots \text{ in} \\ \text{let } b &= \dots \text{ in} \\ \text{letrec } g &= \lambda y.\dots b\dots \text{ in} \\ \text{letrec } f &= \lambda x.\dots a\dots g\dots f\dots \\ \text{in } \dots f\dots g\dots \end{aligned}$$

Then, lambda lifting first applies to  $g$  and then to  $f$  resulting in:

$$\begin{aligned} \text{let } a &= \dots \text{ in} \\ \text{let } b &= \dots \text{ in} \\ \text{letrec } g &= \lambda b.\lambda y.\dots b\dots \text{ in} \\ \text{letrec } f &= \lambda a.\lambda b.\lambda x.\dots a\dots g b\dots f a b\dots \\ \text{in } \dots f a b\dots g b\dots \end{aligned}$$

### 5.1.2 An algorithm for lambda lifting

The main goal for a lambda lifting algorithm is that it should transform the program only once. Hence, the algorithm computes the set of variables over which abstraction is necessary by formulating the constraints on these free variable sets as set equations and then solve those. It assumes that all functions in one *letrec* are mutually dependent.

A prerequisite is that all identifiers in the program scheme must be unique: The program may bind no identifier twice. Unique identifiers simplify the formulation of the algorithm, and generally prevent a few implementation headaches that have to do with inadvertent name capture problems.

The renaming pass also introduces a (unique) name for each lambda expression that does not occur at the top-level of a right-hand side expression in a *letrec*. This introduction step transforms every  $\lambda x.e$  which does not occur next to a  $=$  to *letrec*  $f = \lambda x.e$  in  $f$ .

The second pass proceeds top-down through the expression. It carries with it a set  $F$  which contains the *letrec*-bound identifiers which are currently in scope. Initially, the set  $F$  is empty (in a realistic setting it contains all predefined functions).

The subject of the algorithm is a subterm of the following form where none of the  $e_i$  is a lambda expression:

$$\begin{array}{l} \textit{letrec} \quad f_1 = \lambda x_1.e_1 \\ \qquad \qquad \vdots \\ \qquad \qquad f_n = \lambda x_n.e_n \\ \textit{in} \quad \quad e \end{array}$$

First, let  $F' = F \cup \{\overline{f_n}\}$ . To prepare the  $f_i$  for lifting, abstraction is necessary over all variables that occur free in one of the  $\lambda x_j.e_j$  (assuming full mutual dependency). That is, each definition must be abstracted over the set of free variables in all bodies except those that are *letrec*-bound:

$$\{\overline{y_m}\} = \left( \bigcup_i \text{free}(\lambda x_i.e_i) \right) \setminus F'$$

To ease the subsequent transformation steps, we wrap the abstracted free variables in a vector. This wrapping is indicated by bracketing the free variables as in  $\langle \overline{y_m} \rangle$  and by putting a special mark “@” on the newly introduced lambdas and function applications. Hence, we transform the expression to

$$\begin{array}{l} \textit{letrec} \quad \dots \\ \qquad \quad f_j = \lambda^@z.\lambda x_1.\textit{let} \langle \overline{y_m} \rangle = z \textit{ in} e_j[\overline{f_i} \mapsto \overline{f_i}@z] \\ \qquad \quad \dots \\ \textit{in} \quad \quad \textit{let} z = \langle \overline{y_m} \rangle \textit{ in} e[\overline{f_i} \mapsto \overline{f_i}@z] \end{array}$$

Next, recursively apply lambda lifting to the expressions  $e_1, \dots, e_n$ , and  $e$  using  $F'$  in place of  $F$ .

On return from this recursion, each *letrec* definition has only *letrec*-bound free variables and can be lifted to the toplevel.

## 5.2 Recursive Applicative Program Schemes

Once the program has been lambda lifted, it can be written in a special style called *recursive applicative program scheme*. Such a program scheme is essentially a set of equations, the right-hand sides of which are lambda terms.

### 5.1 Definition (Recursive applicative program schemes)

Let  $\langle \text{const} \rangle$  be a countable set of names for constants with  $\langle \text{var} \rangle \cap \langle \text{const} \rangle = \emptyset$ . Let  $\langle \text{fname} \rangle$  be a countable set of function names, disjoint from  $\langle \text{var} \rangle$  and  $\langle \text{const} \rangle$ . Distinguish between built-in operations that do not have side-effects and those that do by choosing their names from  $\langle \text{pure} \rangle$  and  $\langle \text{oper} \rangle$ , respectively.

An applied lambda term is a term generated by the following grammar:

```

⟨exp'⟩ ::= ⟨var⟩
        | ⟨fname⟩@⟨exp'⟩
        | ⟨const⟩
        | ⟨pure⟩ ⟨exp'⟩ ... ⟨exp'⟩
        | ⟨oper⟩ ⟨exp'⟩ ... ⟨exp'⟩
        | < ⟨exp'⟩ ... ⟨exp'⟩ >
        | ( ⟨exp'⟩ ⟨exp'⟩ )
        | let ⟨var⟩ = ⟨exp'⟩ in ⟨exp'⟩
        | let < ⟨var⟩* > = ⟨exp'⟩ in ⟨exp'⟩
        | if ⟨exp'⟩ then ⟨exp'⟩ else ⟨exp'⟩

```

The applied lambda terms form a set  $E'$ .

A recursive applicative program scheme is a set of equations together with a lambda term with the following form:

```

⟨scheme⟩ ::= ⟨equation⟩* ⟨exp'⟩
⟨equation⟩ ::= ⟨fname⟩ = λ@⟨var⟩. λ⟨var⟩. ⟨exp'⟩

```

The recursive applicative program schemes form a set  $\langle \text{scheme} \rangle$ . □

Here is a recursive applicative program scheme for the **power** function after lambda lifting. Observe that the recursive invocation of **g** does not have to recreate the tuple of free variables, it just takes the one passed from the outside.

```

power = \@ z. \x. let < > = z in g@<x>
g      = \@ z. \n. let <x> = z in if n=0 then 1 else x * g@z (n-1)

```

## 5.3 Naming of Intermediate Values

The first transformation step introduces names for those values that *must* end up in registers by means of inserting *let* expressions. Anything that remains unnamed need not end up in a register. To perform this step sensibly, we must examine each syntactic construct in turn.

**functions** The result of a function will end up in a register and the arguments (at least most of them) will be passed in registers.

**⟨var⟩:** A variable will usually end up in a register.

**$\langle \text{fname} \rangle @ \langle \text{var} \rangle$** : This construct may lead to memory allocation unless it is the function part of an application. If memory allocation is necessary, then its result must be in a register. Otherwise, the expression can be left as is.

**$\langle \text{const} \rangle$** : The constant should end up in a register, unless it is part of a larger pure expression (which is composed of constants, variables, and pure operations).

**$\langle \text{pure} \rangle \langle \text{exp}' \rangle \dots \langle \text{exp}' \rangle$** : A pure operation has no side effects and it usually corresponds to at most one machine instruction. Typical examples of such operations are the arithmetical operations and the bit operations. In some cases it is possible to translate chunks composed of pure operations to a single instruction (or at least to a specialized, more efficient sequence of instructions). As an example, the pure expression  $x + 1$  may be translated to a single instruction that has 1 as an immediate operand. Hence, introducing a name (*i.e.*, a register) for each pure expression at this point seems premature because it obscures the opportunities for identifying such expression chunks. For this reason, the transformation does not introduce names for the arguments of pure expressions and it only introduces a name for the result if the context requires it.

Another reason is that the naming and the subsequent flattening introduces an explicit sequential order of the operations which is not required for pure operations and may, again, lead to inferior code.

**$\langle \text{oper} \rangle \langle \text{exp}' \rangle \dots \langle \text{exp}' \rangle$** : Side-effecting operations must be kept in sequence and they are often implemented by function calls. For these reasons, their arguments must be in registers and they produce their result in a register.

**$\langle \langle \text{exp}' \rangle \dots \langle \text{exp}' \rangle \rangle$** : Allocation of a vector is treated like a side-effecting operation.

**$( \langle \text{exp}' \rangle \langle \text{exp}' \rangle )$** : Because each function call may perform a side-effect, function calls must be kept in sequence. Furthermore, the left-to-right call-by-value semantics demands that the subexpressions are evaluated from left to right and the function parameter must be passed in a register. Hence, each subexpression needs to yield its value in a register and the function's return value ends up in a register, too.

**$\text{let } \langle \text{var} \rangle = \langle \text{exp}' \rangle \text{ in } \langle \text{exp}' \rangle$** : The value of the header expression will be deposited in a variable anyway, so there is no need to force it again to do so. The treatment of the body expression depends entirely on the context of the *let* because the result of the *let* is the result of its body.

**$\text{let } \langle \text{var} \rangle^* = \langle \text{exp}' \rangle \text{ in } \langle \text{exp}' \rangle$** : Like a standard *let* expression.

**$\text{if } \langle \text{exp}' \rangle \text{ then } \langle \text{exp}' \rangle \text{ else } \langle \text{exp}' \rangle$** : The condition must end up in a variable (perhaps corresponding to a condition register on the machine). The result of both branches must be collected so that it can be retrieved from the same place. Hence, both branches must return their values in a variable and the result of the conditional is also in a variable.

Clearly, the transformation function needs to propagate information about the context in which an expression occurs, as witnessed by the cases for  $\langle \text{fname} \rangle @ \langle \text{var} \rangle$ , impure primitives, function calls, and the subexpressions of a conditional. The transformation distinguishes four different contexts  $q \in \{v, f, l, t\}$ :

*v* context demands variable,

$$\begin{aligned}
\llbracket f = \lambda^{\textcircled{z}} \lambda x. e \rrbracket &= f = \lambda^{\textcircled{z}} \lambda x. \llbracket e \rrbracket^v \\
\llbracket x \rrbracket^q &= x \\
\llbracket f @ e \rrbracket^q &= \mathcal{W}^{q \notin \{f, l\}} \llbracket f @ \llbracket e \rrbracket^v \rrbracket \\
\llbracket \langle e_1 \dots e_n \rangle \rrbracket^q &= \mathcal{W}^{q \neq l} \llbracket \langle \llbracket e_1 \rrbracket^v \dots \llbracket e_n \rrbracket^v \rangle \rrbracket \\
\llbracket c \rrbracket^q &= \mathcal{W}^{q \notin \{l, t\}} \llbracket c \rrbracket \\
\llbracket p \ e_1 \dots e_n \rrbracket^q &= \mathcal{W}^{q \notin \{l, t\}} \llbracket p \ \llbracket e_1 \rrbracket^t \dots \llbracket e_n \rrbracket^t \rrbracket \\
\llbracket o \ e_1 \dots e_n \rrbracket^q &= \mathcal{W}^{q \neq l} \llbracket o \ \llbracket e_1 \rrbracket^v \dots \llbracket e_n \rrbracket^v \rrbracket \\
\llbracket e_1 \ e_2 \rrbracket^q &= \mathcal{W}^{q \neq l} \llbracket \llbracket e_1 \rrbracket^f \ \llbracket e_2 \rrbracket^v \rrbracket \\
\llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket^q &= \text{let } x = \llbracket e_1 \rrbracket^l \text{ in } \llbracket e_2 \rrbracket^q \\
\llbracket \text{let } \langle \bar{x} \rangle = e_1 \text{ in } e_2 \rrbracket^q &= \text{let } \langle \bar{x} \rangle = \llbracket e_1 \rrbracket^l \text{ in } \llbracket e_2 \rrbracket^q \\
\llbracket \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rrbracket^q &= \mathcal{W}^{q \neq l} \llbracket \text{if } \llbracket e_1 \rrbracket^v \text{ then } \llbracket e_2 \rrbracket^v \text{ else } \llbracket e_3 \rrbracket^v \rrbracket \\
\mathcal{W}^{\text{true}} \llbracket e \rrbracket &= \text{let } x = e \text{ in } x \\
\mathcal{W}^{\text{false}} \llbracket e \rrbracket &= e
\end{aligned}$$

where  $x \in \langle \text{var} \rangle$ ,  $f \in \langle \text{fname} \rangle$ ,  $p \in \langle \text{pure} \rangle$ ,  $o \in \langle \text{oper} \rangle$ , and  $e, e_i \in \langle \text{exp}' \rangle$ . The auxiliary transformation  $\mathcal{W}^b$  wraps its expression argument in a freshly generated *let*, as determined by its boolean parameter  $b$ .

Figure 5.1: The *let*-introduction transformation.

$f$  context demands function (only used for function subexpression of application),

$l$  context is *let* header,

$t$  context does not place demands.

Figure 5.1 defines the transformation  $\llbracket \cdot \rrbracket^q$  for expressions and equations.

Applying the *let*-introduction step to the running example yields:

```

power = \@ z. \x. let x1 = g@<x> in x1
g      = \@ w. \n.
  let x2 = if (let x3 = n=0 in x3)
             then (let x4 = 1 in x4)
             else (let x5 = w!1 * (let x6 = (g@w)(let x7 = n-1
                                                    in x7)
                                     in x6)
                  in x5)
  in x2

```

## 5.4 Making Sequencing Explicit

The next transformation step makes all sequencing explicit by flattening the highly nested *let* expressions that arise in the previous step. This step is also called *let flattening*. For this purpose, define a set of *trivial terms* which require no further transformation steps. The evaluation of these terms terminates and has no side effects.

```

<triv> ::= <var>
        | <fname>@<var>
        | <const>
        | <pure> <triv> ... <triv>

```

The transformation rules are local rewrite rules that may be applied anywhere in the term. We let  $v$  range over terms generated by  $\langle \text{triv} \rangle$ . All rules for  $\text{let } x = e \text{ in } \dots$  equally apply to  $\text{let } \langle \bar{x} \rangle = e \text{ in } \dots$ . All rules furthermore require that the binding of  $x$  in the outward floating  $\text{let}$  does not capture any free variables. However, this capture avoidance is always granted because we have previously renamed all variables so that each variable has one unique binding location.

$$\begin{array}{ll}
f@(let\ x = e\ in\ e') & \Rightarrow\ let\ x = e\ in\ f@e' \\
p\ v_1 \dots v_i (let\ x = e\ in\ e') e_1 \dots e_k & \Rightarrow\ let\ x = e\ in\ p\ v_1 \dots v_i e' e_1 \dots e_k \\
o\ v_1 \dots v_i (let\ x = e\ in\ e') e_1 \dots e_k & \Rightarrow\ let\ x = e\ in\ o\ v_1 \dots v_i e' e_1 \dots e_k \\
\langle v_1 \dots v_i (let\ x = e\ in\ e') e_1 \dots e_k \rangle & \Rightarrow\ let\ x = e\ in\ \langle v_1 \dots v_i e' e_1 \dots e_k \rangle \\
(let\ x = e\ in\ e') e'' & \Rightarrow\ let\ x = e\ in\ (e'\ e'') \\
v(let\ x = e\ in\ e') & \Rightarrow\ let\ x = e\ in\ (v\ e') \\
let\ y = (let\ x = e\ in\ e')\ in\ e'' & \Rightarrow\ let\ x = e\ in\ let\ y = e'\ in\ e'' \\
if\ (let\ x = e\ in\ e')\ then\ e_2\ else\ e_3 & \Rightarrow\ let\ x = e\ in\ if\ e'\ then\ e_2\ else\ e_3
\end{array}$$

The strategy of the transformation repeatedly applies any of the rules until no further rule is applicable. This application can be shown to terminate always.

The structure of the rules is derived directly from the call-by-value evaluation contexts. Primitive operations evaluate their arguments from left to right. The evaluation of an application evaluates the function expression first and then the argument expression. In a  $\text{let}$  expression, the evaluator first processes the header and then starts evaluation of the body. In a conditional the evaluation first evaluates the condition, then it chooses one of the branches to continue, based on the outcome of the condition.

Applying the  $\text{let}$ -flattening step to the running example yields:

```

power = \@ z. \x.
  let <x> = z in
  let x0 = <x> in
  let x1 = g@x0 in
  x1
g      = \@ w. \n.
  let <x> = w in
  let x3 = n=0 in
  let x2 = if x3
    then (let x4 = 1
          in x4)
    else (let x7 = n-1 in
          let x6 = (g@w) x7 in
          let x5 = x * x6
          in x5)
  in x2

```

## 5.5 Making Control Transfers Explicit

The next step of the transformation specializes the syntax so that only  $\text{let}$ -flattened programs can be expressed and it makes control transfers explicit by introducing special expressions that indicate function returns and jumps out of the branches of a conditional. The revised syntax splits expressions into serious ones and trivial ones. Serious expressions are allowed to do the interesting things like impure operations, function calls, conditionals, returning from functions, and jumping out of conditionals. Trivial expressions are entirely composed of variables, constants, and pure functions as indicated in the  $v$  terms in the previous section.

```

⟨triv⟩ ::= ⟨var⟩
        | ⟨fname⟩@⟨var⟩
        | ⟨const⟩
        | ⟨pure⟩ ⟨triv⟩ ...⟨triv⟩

⟨hdr⟩ ::= ⟨triv⟩
        | ⟨oper⟩ ⟨var⟩ ...⟨var⟩
        | < ⟨var⟩ ...⟨var⟩ >
        | ( ⟨triv⟩ ⟨var⟩ )

⟨ser⟩ ::= return ⟨var⟩
        | let ⟨vars⟩=⟨hdr⟩ in ⟨ser⟩
        | ( ⟨triv⟩ ⟨var⟩ )
        | if ⟨var⟩ then ⟨ser⟩ else ⟨ser⟩
        | letcont k@⟨vars⟩ = λ⟨var⟩. ⟨ser⟩ in ⟨ser⟩
        | yield k ⟨var⟩

⟨equation⟩ ::= ⟨fname⟩ = λ@⟨var⟩. λ⟨var⟩. ⟨ser⟩
⟨vars⟩ ::= < ⟨var⟩* >

```

The serious expression  $\text{letcont } k@⟨x_1 \dots x_n⟩ = \lambda x.s_1 \text{ in } s_2$  serves to introduce the target address of the jump out of the branches of a conditional. The function  $k$  represents this target address and it is invoked with the *yield* expression. The value  $v$  passed to  $\text{yield } k \ v$  is bound to  $x$ . The use of this feature is best demonstrated by looking at the running example after applying the transformation.

```

power = \@ z. \x.
  let <> = z in
  let x0 = <x> in
  let x1 = g@x0 in
  return x1
g     = \@ w. \n.
  let <x> = w in
  let x3 = n=0 in
  letcont k@<> = \x2. return x2 in
  if x3
  then (let x4 = 1 in
        yield k x4)
  else (let x7 = n-1 in
        let x6 = (g@w) x7 in
        let x5 = x * x6 in
        yield k x5)

```

The example clearly exhibits the tasks of the transformation. Wherever a function returns to its caller, the transformation inserts a *return* applied to a variable. Wherever a conditional appears, the transformation inserts an abstraction over the context of the conditional, gives it a name  $k$ , and recursively transforms the branches of the conditional now using *yield k* instead of *return* to pass the result to the context. The “context of the conditional” means the code that processes the result of the conditional. In the example, this code is just

```
let x2 = [ ... ] in x2
```

$$\begin{aligned}
\mathcal{R}[f = \lambda^@z.\lambda x.e] &= f = \lambda^@z.\lambda x.\mathcal{R}_0[e] \\
\mathcal{R}_k[x] &= \begin{cases} \text{return } x & \text{if } k = 0 \\ \text{yield } k \ x & \text{otherwise} \end{cases} \\
\mathcal{R}_k[\text{let } x = \text{if } y \text{ then } e_1 \text{ else } e_2 \text{ in } e] &= \text{letcont } k'@(\text{free}(e) \setminus \{x\}) = \lambda x.\mathcal{R}_k[e] \text{ in} \\
&\quad \text{if } y \text{ then } \mathcal{R}_{k'}[e_1] \text{ else } \mathcal{R}_{k'}[e_2] \\
\mathcal{R}_k[\text{let } \bar{x} = e \text{ in } e'] &= \text{let } \bar{x} = e \text{ in } \mathcal{R}_k[e']
\end{aligned}$$

Figure 5.2: Introduction of explicit control transfers.

In general, more interesting things may happen.

The choice of the name  $k$  for the context function comes from its usual denomination “continuation”. This name is generically used for all kinds of function that abstract over “the rest of a computation”. Continuations may be used to implement arbitrary control structures. For example, continuations give rise to a natural implementation of exceptions. More information may be found in the book [FWH01].

Figure 5.2 defines the transformation in terms of a function  $\mathcal{R}_k[\cdot]$  which maps an expression to a serious expression. The parameter  $k$  is either 0 to indicate a top-level continuation (*i.e.*, a function return) or it is a continuation introduced with *letcont*.

At first, it may seem surprising that there are only so few cases to the function. However, it can be proven that no other cases arise after the expression has been subjected to let-flattening.

Looking a bit more closely at the code of the running example, the reader might wonder why we introduced the continuation  $\lambda x.\text{return } x$  at all. This continuation and the inefficiency that results from first jumping out of the conditional and then returning from the surrounding function call could be avoided by replacing the *yield*  $k$ s with *returns* and by just dropping the continuation. Here is the resulting code for  $g$ :

```

g = \@ w. \n.
  let <x> = w in
  let x3 = n=0 in
  if x3
  then (let x4 = 1 in
        return x4)
  else (let x7 = n-1 in
        let x6 = (g@w) x7 in
        let x5 = x * x6 in
        return x5)

```

In a similar vein, the transformation pipeline as it is presented produces further jumps to jumps. Consider transforming the function

```
down n = if n=0 then 1 else down (n-1)
```

Even after “cleaning up” the continuation as in the `power` example, the resulting code looks as follows:

```

down = \@ z. \n.
  let x1 = n=0 in
  if x1
  then let x2 = 1 in

```

```

    return x2
  else let x3 = n-1 in
    let x4 = down@z (x3) in
      return x4

```

It seems fairly wasteful to return from the recursive function call to `down` just to immediately return the same value again.

Indeed, it is possible to optimize this code (by exploiting an as yet unused production of the definition of `<ser>`) to

```

down = \@ z. \n.
  let x1 = n=0 in
  if x1
  then let x2 = 1 in
    return x2
  else let x3 = n-1 in
    down@z (x3)

```

This optimization step is called “proper tail-recursion” and it allows function calls that occur “at the end” of the calling function to avoid having to return to the calling function but instead to return immediately to the caller of the caller. More accurately, it will return to the ancestor of the caller that performed a function call that was not in tail position.

We leave the incorporation of this tail-recursion detection into our transformation pipeline as an easy exercise. In addition, we define a transformation that implements the whole pipeline (including tail recursion) in one pass in the next section.

## 5.6 An Optimized One-pass Transformation

The main trick of the optimized one-pass transformation is to make use of a continuation as a programming device. As already mentioned, a continuation abstracts over the context of a computation. The one-pass transformation exploits this fact by generating the desired *let* expressions on-the-fly and by nesting the context inside of those *lets* to perform the flattening transformation implicitly.

Let’s start with a type definition for recursive applicative program schemes.

```

type exp =
  Var of ident
  | Const of ident
  | Pure of ident * exp list
  | Oper of ident * exp list
  | Vector of exp list
  | App of exp * exp
  | Let of ident list * exp * exp
  | If of exp * exp * exp
  | Close of exp * exp
type equation =
  ident * ident list * ident * exp
type scheme =
  equation list * exp

```

It mirrors exactly the previous definition in terms of a grammar. Most alternatives should be self-explanatory, with the non-obvious ones being: `Close (f, z)` stands

for the expression  $f@z, \text{Vector } [e_1; \dots; e_n]$  stands for  $\langle e_1, \dots, e_n \rangle$ . The equation  $(f, [x_1; \dots; x_n], x, e)$  stands for  $f = \lambda^{\textcircled{z}} \langle x_1, \dots, x_n \rangle. \lambda x. e$  and a scheme is a list of equations followed by an expression, as usual.

The next type represents the target language composed of trivial expressions and serious expressions.

```
type trivial =
  TVar of ident
  | TConst of ident
  | TPure of ident * trivial list
  | TClose of ident * ident
```

This type corresponds directly to  $\langle \text{triv} \rangle$ . There is an extra type describing the possible let headers.

```
type header =
  HTriv of trivial
  | HOper of ident * ident list
  | HApp2 of ident * ident
  | HApp3 of ident * ident * ident
  | HVect of ident list
```

```
type serious =
  SReturn of ident
  | SApp2 of ident * ident
  | SApp3 of ident * ident * ident
  | SLet of ident list * header * serious
  | SIf of ident * serious * serious
  | SLetCont of kident * ident list * ident * serious * serious
  | SYield of kident * ident
```

Here, `SApp2` and `SApp3` correspond to tail calls to an unknown function ( $x y$ ) and to a known function  $f@z(y)$ . Similarly, `SLet( ... HApp2 ... )` and `SLet( ... HApp3 ... )` denote ordinary calls to unknown and known functions. The remaining alternatives match with the definition of  $\langle \text{ser} \rangle$  in the obvious way.

The representation of identifiers is not really important for the transformation. The only required feature is a facility for generating fresh names which are not yet in use.

```
type ident =
  StrId of string
  | NumId of string * int
type kident =
  ident

let fresh_counter =
  ref 0
let fresh_prefix =
  begin
    fresh_counter := !fresh_counter + 1;
    NumId (prefix, !fresh_counter)
  end
```

Furthermore, there are auxiliary functions to compute the free variables of trivial and serious expressions. Their definition is standard and therefore omitted.

The function `toSerious subst e c` implements the main part of the transformation from expressions of type `exp` to serious expressions. Its type is

```
toSerious : (string * string) list -> exp -> (ident -> serious) -> serious
```

The first argument, `subst`, is a variable renaming. The renaming is applied to all variables before outputting them. It simplifies the incorporating of existing *let* expressions. The second argument, `e`, is the expression to be transformed. The third argument, `c`, is the continuation. It takes the identifier which carries the result of the calling expression and returns a `serious` term. The notable technique used in the transformation is that the generation of serious terms may either be wrapped around a call to `c` or it may happen by abstracting the generation into a function and pass it as a continuation to `toSerious`.

Let's have a look at the code:

```
let rec
  toSerious subst e c =
  match e with
  Var i ->
    c (rename i subst)
```

A variable does not require any *let*-wrapping, so the transformation passes it directly to the continuation. Before doing so, the variable is renamed according to the renaming substitution.

```
| Const p ->
  let fr = fresh "t" in
  SLet ([fr], HTriv (TConst p), c fr)
```

Because the context of a serious term is a function return, a *let* has to be introduced at this point.

```
| Pure (p, es) ->
  toTrivialL subst es
  (fun ts ->
    let fr = fresh "t" in
    SLet ([fr], HTriv (TPure (p, ts)), c fr))
```

The introduction of the *let* at this point has the same motivation as for the constant. However, the subexpressions are treated with a different transformation `toTrivialL` that is geared towards passing a list of `trivial` terms to its continuation. Hence, the transformation builds the *let* directly in its final form and the implementation with continuation sees to it that serious computations nested within the arguments `es` of the pure operation are generated before.

```
| Oper (o, es) ->
  let fr = fresh "t" in
  toSeriousL subst es
  (fun is ->
    SLet ([fr], HOper (o, is), c fr))
```

Here the recursive call goes to `toSeriousL` which yields a list of identifiers which are bound to the variables in the substitutions.

The next two cases consider application of known and unknown functions, in this order. Both rely entirely on `toSerious` because the corresponding `serious` term requires only variables as arguments.

```
| App (Close (e1, e2), e) ->
  toSerious subst e1
  (fun x1 ->
```

```

    toSerious subst e2
      (fun x2 ->
        toSerious subst e
          (fun x3 ->
            let fr = fresh "t" in
              SLet ([fr], HApp3 (x1, x2, x3), c fr))))
  | App (f, e) ->
    toSerious subst f
      (fun ff ->
        toSerious subst e
          (fun ee ->
            let fr = fresh "t" in
              SLet ([fr], HApp2 (ff, ee), c fr)))

```

The transformation of already existing *let* expressions is the reason for having the renaming `subst` in the transformation.

```

  | Let ([x], e1, e2) ->
    toSerious subst e1
      (fun xx ->
        toSerious ((x, xx) :: subst) e2 c)

```

In transforming such a *let* expression, we must somehow arrange that the value of `e1` is bound to the variable `x` before entering `e2`. Unfortunately, the `toSerious` transformation is not equipped to take advice about the name to which it should bind its result. It rather generates this name on the fly or reuses the name of another already existing variable. Hence, the transformation intercepts the name of `e1`'s result in `xx` and transforms `e2` with a renaming that maps the previous name `x` to freshly generated name `xx`.

*Let* expressions that decompose vectors work differently. The transformation needs to leave them in place.

```

  | Let (xs, e1, e2) ->
    toSerious subst e1
      (fun ee1 ->
        SLet (xs, HTriv (TVar ee1), toSerious subst e2 c))

  | If (e1, e2, e3) ->
    toSerious subst e1
      (fun x1 ->
        let kk = fresh "k" in
          let xx = fresh "t" in
            let body = c xx in
              SLetCont
                (kk, remove xx (free_serious body), xx, body,
                 SIf
                  (x1,
                   toSerious subst e2 (fun i -> SYield (kk, i)),
                   toSerious subst e3 (fun i -> SYield (kk, i))))))

```

The transformation treats the conditional exactly as outlined in the description of  $\mathcal{R}[\llbracket \cdot \rrbracket]$  in the previous section. The important thing is that the continuation `c` generates the transformed expression for the context of the conditional. The use of `SYield` in the continuation for transforming the true- and false-branches guarantees that they pass their results to the context.

The remaining cases are just easy variations of the same theme.

```

| Close (e1, e2) ->
  toSerious subst e1
  (fun x1 ->
    toSerious subst e2
    (fun x2 ->
      let fr = fresh "t" in
      SLet ([fr], HTriv (TClose (x1, x2)), c fr)))
| Vector (es) ->
  toSeriousL subst es
  (fun xs ->
    let fr = fresh "t" in
    SLet ([fr], HVect(xs), c fr))

```

The corresponding function for building up trivial terms is

```
toTrivial : (string * string) list -> exp -> (trivial -> serious) -> serious
```

The only difference to the type of `toSerious` is in the argument of the continuation. It expects a trivial term instead of an identifier.

The implementation of `toTrivial` just checks the cases where it can continue building a trivial term and bails out to `toSerious` whenever that is impossible. Those cases correspond exactly to the definition of `trivial`.

```

and
  toTrivial subst e c =
  match e with
  | Var i ->
    c (TVar (rename i subst))
| Const p ->
  c (TConst p)
| Pure (p, es) ->
  toTrivialL subst es
  (fun ts ->
    c (TPure (p, ts)))
| Vindex (e, i) ->
  toTrivial subst e
  (fun t ->
    c (TIndex (t, i)))
| _ ->
  toSerious subst e
  (fun xx ->
    c (TVar xx))

```

The definitions of `toTrivialL` and `toSeriousL` are straightforward and therefore omitted.

The transformation of an expression `e` is initiated with

```
toSerious [] e (fun i -> SReturn i)
```

Transforming an expression in this way reveals that the transformation does not yet implement proper tail recursion. In particular, each term generated with the above call of `toSerious` will contain `SReturn i!`

For this reason, we introduce a variant of `toSerious` that only works on top-level expressions.

```

let rec toSeriousT subst e =
  let c x = SReturn x in

```

```

match e with
| App (Close (e1, e2), e) ->
  toSerious subst e1
  (fun x1 ->
    toSerious subst e2
    (fun x2 ->
      toSerious subst e
      (fun x3 ->
        SApp3 (x1, x2, x3))))))
| App (f, e) ->
  toSerious subst f
  (fun ff ->
    toSerious subst e
    (fun ee ->
      SApp2 (ff, ee)))

```

Transforming with `toSeriousT` transforms a top-level function call into a tail-call.

```

| Let (x, e1, e2) ->
  toSerious subst e1
  (fun xx ->
    toSeriousT ((x, xx) :: subst) e2)

```

If a *let* expression occurs in tail position, the header is treated as usual and then the body still occurs in tail position. The rationale for the conditional is quite similar. If the conditional has no context, then the transformation may consider each branch as top-level.

```

| If (e1, e2, e3) ->
  toSerious subst e1
  (fun x1 ->
    SIf
      (x1,
        toSeriousT subst e2,
        toSeriousT subst e3))
| _ ->
  toSerious subst e c

```

The type of this top-level function is very simple because the continuation argument is elided:

```
toSeriousT : (string * string) list -> exp -> serious
```

## 5.7 Introducing Closures

The one remaining question in the transformed code is the representation of functions and other control transfers. Recall that there are the following facilities for defining and using control transfers:

- Define a top-level function in an equation

$$f = \lambda^@z.\lambda x.s$$

- Call a known top-level function ( $f$  is the name of a top-level function)

$$f@z(x)$$

- Call an unknown top-level function ( $y$  is not a top-level function)

$$y(x)$$

- Return from a top-level function

$$\text{return } x$$

- Define a continuation

$$\text{letcont } k@ \langle x_1 \dots x_n \rangle = \lambda x.s \text{ in}$$

- Use a continuation

$$\text{yield } k \ x$$

A call to a top-level function remembers the environment in which it was called and can return to it through the *return* instruction. That is, at each time during the execution of a program there is a stack of pending top-level function activations consisting of the current environment of a function (its arguments and locally defined values) and a return address that indicates where the function is to continue when the return value becomes available. This collection of data (return address, function arguments, and local values) is called an *activation record* or *activation block*. Hence, the implementation of a top-level function call (in addition to performing the control transfer, of course) must save the previous activation record on entry to the function and the corresponding return must obtain the return address from current activation record and restore the previous one.

In contrast, a call to a continuation neither requires saving values nor return addresses and it does not create a new activation record. Hence, the representation of a continuation is just the address of a piece of code and its invocation amounts to jumping to this address after putting the value passed to the continuation in place (*e.g.*, a register). The mention of the free variables  $\langle x_1 \dots x_n \rangle$  in the definition of the continuation merely helps keep subsequent code generation a local task.

The representation of a top-level function is slightly more involved. Due to lambda lifting every top-level function has two parameters, a vector of values (from lifting the free variables) and the current argument. To invoke the function requires the presence of both parameters. Let's consider the two cases.

In a call to a known top-level function  $f@z(x)$ , the function's address is known as  $f$ , the vector of free variable values is available through  $z$ , and the argument's value is available in  $x$ . Hence, all information is available for generating the function invocation.

In a call to an unknown (top-level) function  $y(x)$ , the argument's value is available in  $x$  so that the remaining information (address and variable vector) must be available through  $y$ . This requirement dictates the representation of an unknown function as a *closure*.

A closure is a pair consisting of

- the entry point (address) of the function and
- the vector of the values of its free variables.

With this representation in place, all uses of functions may now be brought into the format of a known call:

- transform all occurrences of  $\text{let } w = f@z \text{ in } s$  to  
 $\text{let } w = \langle f, z \rangle \text{ in } s$  (create a closure as a pair of address and vector)

- transform  $y(x)$  to  
 $let \langle z_1, z_2 \rangle = y \text{ in } z_1 @ z_2(x)$
- transform  $let w = y(x) \text{ in } s$  to  
 $let \langle z_1, z_2 \rangle = y \text{ in } let w = z_1 @ z_2(x) \text{ in } s$

At this point, it is clear for each function invocation what is the address of the function body and what are the two parameters to pass to it.