Chapter 9
Translation into Clean

In this chapter the most important advanced language concepts offered by a lazy functional language such as Miranda are transformed into an FGRS-based language such as Clean (Figure 9.1). The transformation schemes take Miranda explicitly as the source language, but they can also serve as a guideline for the translation of other lazy functional languages.

As a 'side-effect', this chapter gives a better insight in the semantics of the treated language concepts, especially with respect to sharing.

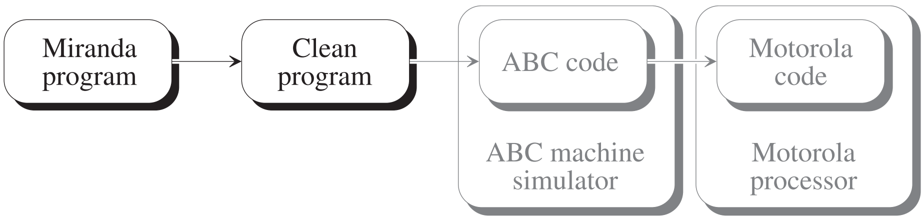

Figure 9.1 The first translation step: translating a Miranda script into an equivalent Clean program.

The goal of the transformation schemes is to obtain Clean code that is as efficient as possible. The transformation schemes presented in this chapter are based on the standard transformation techniques (Peyton Jones, 1987). However, some changes have been made to generate *better code* and to employ *sharing* as much as possible (Koopman and Nöcker, 1988) to avoid redundant computation.

During the transformation auxiliary functions are inevitably generated. The transformation schemes are designed in such a way that generation of new functions is reduced to a minimum.

A significant increase of execution speed can be obtained when a function is known to be strict in an argument of basic type (see Section 8.3). The corresponding analysis is not performed during the translation from Miranda to Clean but it is left to the strictness analyser of the Concurrent Clean System (see Chapter 7).

The transformation is performed in two phases. In the first phase a Miranda program is desugared by translating it into an equivalent program in a Miranda subset that is very Clean-like. This approach has the advantage that most transformations can be defined by using the semantics of Miranda itself (Section 9.1). As much as possible of Miranda is desugared in this way: list comprehensions (Section 9.3), local function definitions (Section 9.4) and various other small syntactical constructs (Section 9.2).

Phase two (Section 9.5) is relatively straightforward: an almost trivial syntactical transformation is needed to convert a program in the desugared language into an equivalent Clean program. Finally we discuss how sharing can be exploited in the transformation of Miranda to Clean (Section 9.6).

9.1 About the desugaring of the language

In the first phase Miranda programs are desugared by transforming them into a Miranda subset that is syntactically similar to Clean. These transformations can be split into relatively simple changes such as the transfer from infix into prefix notation and the more complex transformation schemes, such as removing list comprehensions. Miranda constructs that are more or less part of the programming environment, such as the module system, I/O facilities and predefined library modules, are not considered in this chapter.

An important aspect in the transformations concerns a special kind of local definition: the **local constant** definition, i.e. a local function definition with *zero* arguments or a local *tuple* definition. In the following a strict distinction will be made between local *constant* definitions and local *function* definitions (the other local definitions). Local constants can be defined in Clean as well as in Miranda and therefore they will *not* be moved to a global level (see Section 9.4). In the second phase (see also Section 9.5) the local constant definitions are translated directly into the equivalent node-id definitions in Clean. In Clean they will be *shared* automatically such that duplication of computation is avoided. In several transformation schemes local constant definitions are introduced to avoid duplication of work.

Most of the program transformations are easy, others are a bit more difficult. It is not always evident which transformation scheme will yield the most efficient code. In some cases several alternatives for a particular transformation are given. The following remarks hold for all transformations:

- The transformation schemes produce standard Miranda expressions (e.g. infix notation). This is merely done for readability. Actually, the expressions thus generated in their turn also have to be transformed into the wanted Miranda subset.
- Some transformations require the insertion of new local definitions. In Miranda local definitions are only allowed at particular places. The transformation schemes do not take this into account, simply because it would make the description unreadable.
- Often it does not matter very much in which order the transformations are being performed. But rule lifting, i.e. moving all local function definitions (including the ones inserted by the transformations) to a global level, should be done at the end.
- In some transformations (new) names are introduced. It is assumed that these names do not conflict with already existing names.

Trivial transformations are explained with the help of examples. In other cases the transformation schemes (called MirSMir) are more formally specified in the usual manner.

9.2 Simple transformations

Since pattern matching is available in Miranda as well as in Clean, pattern matching can generally be translated directly without any changes. However, some special patterns are allowed in Miranda and not in Clean. In particular, *irrefutable patterns*, *comparing rules*, *constructor definitions* and *omitted arguments in function definition*s have to be transformed into Clean-like definitions. Furthermore, Miranda allows *infix* notation and Clean generally allows only *prefix* notation.

Infix notation

In Miranda there are many predefined infix operators with various binding powers. User-defined infix operators are also possible. Furthermore, mathematical expressions like a < b < c can occur. In Clean, the prefix notation is generally used and all functions have the same binding power. Infix notations can be removed easily. The general transformation is (in which i_1, i_2 stand for infix symbols, and e_1, e_2 and e_3 stand for arbitrary expressions):

$$\text{MirSMir} [\![e_1\ i_1\ e_2\ i_2\ e_3]\!] = (\&)\ ((i_1)\ e_1\ s)\ \text{MirSMir} [\![s\ i_2\ e_3]\!] \ \text{where} \quad s = e_2$$
$$\text{MirSMir} [\![e_1\ i_1\ e_2]\!] = (i_1)\ e_1\ e_2$$

For instance, a < b < c is translated as:

```
MirSMir ⟦ a < b < c ⟧  →  (&) ((<) a s) MirSMir ⟦ s < c ⟧   where  s = b
                        →  (&) ((<) a s) ((<) s c)           where  s = b
```

In the transformation new local constant definitions are generated to achieve sharing in Clean.

The match on syntactic constructs automatically takes care of the various binding powers, as illustrated in the example below:

```
MirSMir ⟦ 2 < 3 + 1 ⟧  →  (<) 2 MirSMir ⟦ 3 + 1 ⟧  →  (<) 2 ((+) 3 1)
```

Comparing rules

In Miranda a variable can occur multiple times on a left-hand side. Such a function alternative is called a **comparing rule**, which is only chosen if the values corresponding to this variable are equal.

The following rule will only be chosen if the two arguments are equal:

```
f x x  =  x
```

Comparing is not allowed in FGRSs and hence it is also prohibited in Clean. However, a δ-rule is defined that tests the equality of normal forms. The transformation is therefore the following:

Redefine the rule above by:

```
f x y  =  x,  if x = y
```

Since the guard is evaluated after the pattern is matched, in particular cases (e.g. when the left-hand side of the rule is f x x 2) one has to take care that the order in which the tests and matches are performed is not changed by the transformation. This can be realized by generating auxiliary function definitions.

Irrefutable patterns

In Miranda no evaluation is forced for an argument corresponding to a pattern if the type checker guarantees that the evaluated argument will always match the pattern. Such patterns are called **irrefutable patterns**.

Consider the following Miranda function definition:

```
f:: (*, **) -> num
f (x, y)   =  3
```

In this function a tuple is specified as pattern, but the tuple elements are not used in the function body. The type system guarantees that the function is always applied on a tuple. So pattern matching is in this case not necessary.

Clean would force evaluation for these kinds of pattern. This may lead to unintended non-termination if the function is not strict in its argument and the argument does not have a root normal form.

The problem is solved by eliminating the pattern match and including some way in which the components of the pattern can be accessed in a lazy manner. This is done by adding local constructor definitions (see below) or projection functions.

A correct translation of the example above is

```
f:: (*, **) -> num
f z    =  3  where (x, y)   =  z
```

Constructor definitions

A **constructor definition** is a (possibly local) definition in which the defined symbol is not a function but a constructor. Such a definition is effectively used to define projection functions yielding the arguments of the constructor.

```
C 2 x y  =  e
```

C is a Miranda constructor. In this way projection functions x and y are defined and a test on the value of the first argument of C is induced.

A possible translation involves the explicit definition of the corresponding objects using projection functions.

The example above is translated into:

$$
\begin{array}{lll}
x & = & f_1\ z \\
y & = & f_2\ z \\
z & = & e \\
f_1\ (C\ 2\ a\ b) & = & a \\
f_2\ (C\ 2\ a\ b) & = & b
\end{array}
$$

If the definitions are local, then after the transformations of the second phase e will be shared in Clean.

A disadvantage of this method is the generation of new projection functions for each occurrence of a constructor definition. In practice the same projection functions can be used for many different occurrences.

Omitted arguments in function definitions

In Miranda it is not required to specify the same number of arguments in all alternatives of a function definition.

In the definition of f some of its arguments are omitted:

```
f 1    =  fac
f a 2  =  a * 3
f b    =  (*) 5,     if b < 0
       =  (–) 2,     if b > 5
f      =  plus
```

In Clean function definitions have fixed arity so such definitions have to be transformed. The omitted arguments are added on the left-hand side as well as to corresponding expressions on the right-hand side.

The example above with added arguments becomes:

```
f 1 x  =  fac x
f a 2  =  a * 3
f b x  =  (*) 5 x,   if b < 0
       =  (–) 2 x,   if b > 5
f x y  =  plus x y
```

9.3 List comprehensions

In this section we shall formally specify how dotdot expressions and non-diagonalized *ZF*-expressions can be desugared into function definitions. Diagonalized *ZF*-expressions are not treated. For non-diagonalized *ZF*-expressions besides the standard transformation method (Peyton Jones, 1987) a method that generates more efficient code is described. This method makes use of patterns and compile-time continuations (Koopman and Nöcker, 1988).

9.3.1 Dotdot expressions

In Section 2.2 the semantics of dotdot expressions was explained with the help of two auxiliary functions, inbetween and from. The general translation of such dotdot expressions is as follows (in which a, b and c stand for arbitrary numeral expressions):

```
MirSMir ⟦ [a..b] ⟧     =  inbetween a b 1
MirSMir ⟦ [a..] ⟧      =  from a 1
MirSMir ⟦ [a, b..c] ⟧  =  inbetween s c (b – s)   where   s  =  a
MirSMir ⟦ [a, b..] ⟧   =  from s (b – s)          where   s  =  a
```

New local constant definitions are generated for the use of sharing.

9.3.2 *ZF*-expressions

The translation of non-diagonalized *ZF*-expressions is more difficult. Two methods are given. The first scheme is primarily given to make the semantics of non-diagonalized *ZF*-expressions clear, but it is not very efficient. At run-time a lot of time is wasted concatenating empty lists. The second scheme presented is more complex, but it also generates more efficient code.

Simple transformation of ZF-expressions

The following transformations are essentially the same as the basic transformation schemes given in Peyton Jones (1987). As with the transformation of the dotdot expressions, the original expression is not duplicated during the transformation. Instead new local definitions are generated in such a way that, after the transformations of the second phase, the original expression is shared in Clean.

ZF-expressions can be translated as follows (in which e stands for an arbitrary expression, b for a Boolean-valued filter, q for a list of qualifiers, p, p_1 and p_n for patterns, x, y and z for simple variables, and list and l_1 for a list-valued expression):

```
MirSMir ⟦ [e | b] ⟧                      =  cond b [e] [ ]
MirSMir ⟦ [e | b;q] ⟧                    =  cond b MirSMir ⟦ [e | q] ⟧ [ ]
MirSMir ⟦ [e | p <- list] ⟧              =  flatmap f list
                                             where  f p   =  [e]
                                                    f x   =  [ ]
MirSMir ⟦ [e | p <- list;q] ⟧            =  flatmap f list
                                             where  f p   =  MirSMir ⟦ [e | q] ⟧
                                                    f x   =  [ ]
MirSMir ⟦ [e | p1, ..., pn <- list] ⟧    =  MirSMir ⟦ [e | p1 <- l1;...;pn <- l1] ⟧
                                             where  l1    =  list
MirSMir ⟦ [e | p1, ..., pn <- list;q] ⟧  =  MirSMir ⟦ [e | p1 <- l1;...;pn <- l1;q] ⟧
                                             where  l1    =  list
```

The function cond is the conditional with boolean, 'then' part and 'else' part arguments. This function is used instead of guards, since guards cannot be nested easily. It is sensible to replace, at a later stage, the applications of cond as much as possible by guards.

The local constant definitions for l_1 are generated to create sharing in Clean. The local function definition for f is generated in order to achieve the proper projection from the pattern p to the expression e. The second alternative of f is only needed for the case where the pattern p is not just a simple variable.

The function flatmap is a variant of map (see Chapter 2). It is defined below:

```
flatmap:: (* -> [**]) -> [*] -> [**]
flatmap f [ ]        =  [ ]
flatmap f (x : r)    =  f x ++ flatmap f r
```

In spite of the introduced sharing the transformation scheme given above is not very efficient because at run-time many empty lists are concatenated.

Take the following Miranda expression computing all Pythagorean triangles with sides less than or equal to n and a<b<c:

```
pyth n   =  [ (a, b, c) | a, b, c <- [1..n]; a<b<c; a^2 + b^2 = c^2 ]
```

According to the simple transformation scheme given above, this *ZF*-expression will be translated into:

```
pyth n   =  flatmap f1 list
            where
            list   =  [1..n]
            f1 a   =  flatmap f2 list
                      where
                      f2 b   =  flatmap f3 list
                                where
                                f3 c   =  cond (a<b<c)
                                                (cond (a^2+b^2=c^2) [(a, b, c)] [ ])
                                                [ ]
```

The code yields the concatenation of almost n^3 empty lists.

More efficient transformation of ZF-expressions

There are several ways to improve the transformation schemes given above (see also Peyton Jones, 1987). We will treat here the scheme of Koopman and Nöcker (1988). It presents an alternative transformation by using patterns. Furthermore, an additional argument is used in the transformation scheme to remember where the generation of list elements has to be continued. The advantage of the scheme is that it does not generate empty lists nor append functions. The proposed scheme also contains a slight optimization in the sequence of filters.

```
MirSMir ⟦ [e | q] ⟧                  =  MirSMir' ⟦ [e | q] ⟧ [ ]

MirSMir' ⟦ [e | b1;b2] ⟧ r           =  MirSMir' ⟦ [e | b1&b2] ⟧ r
MirSMir' ⟦ [e | b1;b2;q] ⟧ r         =  MirSMir' ⟦ [e | (b1&b2);q] ⟧ r
```

```
MirSMir' ⟦ [e | b] ⟧ r                     = cond b (e : r) r
MirSMir' ⟦ [e | b;q] ⟧ r                   = cond b (MirSMir' ⟦ [e | q] ⟧ r) r
MirSMir' ⟦ [e | p <- list] ⟧ r             = f list
                                              where
                                              f [ ]      = r
                                              f (p : y)  = e : f y
                                              f (x : y)  = f y
MirSMir' ⟦ [e | p <- list;q] ⟧ r           = f list
                                              where
                                              f [ ]      = r
                                              f (p : y)  = MirSMir' ⟦ [e | q] ⟧ (f y)
                                              f (x : y)  = f y
MirSMir' ⟦ [e | p1, ..., pn <- list] ⟧ r   = MirSMir' ⟦ [e | p1 <- l1;...;pn <- l1] ⟧ r
                                              where  l1 = list
MirSMir' ⟦ [e | p1, ..., pn <- list;q] ⟧ r = MirSMir' ⟦ [e | p1 <- l1;...;pn <- l1;q] ⟧ r
                                              where  l1 = list
```

r is used to remember where the evaluation has to be continued if a generator has come to the end of the generated list. This is specified in the first alternative of the locally defined function f in the scheme. The last alternative of this function f is only needed when p is a pattern that is not just a simple variable.

An empty list is passed as an additional argument to the new transformation scheme. The previous transformation schemes are only slightly changed to pass the extra argument. Furthermore, two rules are inserted to obtain better code in the case that multiple filters are specified. Finally, the two rules for generators are replaced by two rules that employ the continuation passed in the extra argument.

With this new scheme the Pythagorean triangles will be translated into:

```
pyth n  =  f1 list
           where
           list      =  [1..n]
           f1 [ ]    =  [ ]
           f1 (a:x)  =  f2 list
                        where
                        f2 [ ]    =  f1 x
                        f2 (b:x)  =  f3 list
                                     where
                                     f3 [ ]    =  f2 x
                                     f3 (c:x)  =  cond (a<b<c & a^2+b^2=c^2)
                                                       ((a, b, c) : f3 x)
                                                       (f3 x)
```

ZF-expressions are very concise constructs. A disadvantage is that even with the most efficient transformation scheme they are transformed into many function definitions and applications. So a functional program with many *ZF*-expressions might be a beautiful specification, but it is in many cases an inefficient program.

Recurrent generators

The previous scheme for *ZF*-expressions is further extended such that it can also deal with recurrent generators. In Chapter 2 we explained that the meaning of a recurrent generator can be expressed with the help of the function iterate. So a recurrent generator can be translated by adding the following rules:

```
MirSMir' ⟦ [e | p <- e0, en..] ⟧ r      =  MirSMir' ⟦ [e | p <- iterate g e0] ⟧ r
                                          where   g p   =  en
MirSMir' ⟦ [e | p <- e0, en..;q] ⟧ r    =  MirSMir' ⟦ [e | p <- iterate g e0;q] ⟧ r
                                          where   g p   =  en
```

9.4 Local function definitions

A functional program containing local function definitions has to be transformed in such a way that all function definitions are on the global level, such as is the case in Clean. This transformation is called **rule lifting**. The general idea is to make the local functions self-contained by adding parameters for certain variables (*free* variables). The extra arguments have to be added in front of the original parameters, since for curried applications of functions generally not all original arguments are available (see also the example below).

Take the definition of a local function p in the following definition of filter:

```
filter a (b:x)    =  p a,        if b mod a = 0
                  =  g b p a,  otherwise
                     where p y   =  filter y x
g x y z           =  x : y z
```

It will be first transformed into:

```
filter a (b:x)    =  p x a,          if b mod a = 0
                  =  g b (p x) a,    otherwise
                     where p x y    =  filter y x
g x y z           =  x : y z
```

p is used in a curried way. Therefore it is necessary that the extra argument x is added in front of the original argument. This extra argument is added both to the definition of p as well as to the applications of p. Now the definition of p is self-contained and it can be lifted:

```
filter a (b:x)    =  cond ((b mod a) = 0) (p x a) (g b (p x) a)
p x y             =  filter y x
g x y z           =  x : y z
```

A self-contained local definition can be moved freely to the global level. Of course, when moving a local definition to a global level, names have to be changed if the name of the function already occurred at that level. In this section rule lifting is discussed in general and an example is given of a rule lifter for Miranda.

λ-lifters are well-known lifting algorithms (Johnsson, 1985) that are defined for λ-calculus-based expressions. The lifting process defined in this section basically follows the same algorithm but there are some differences. In particular, *rule lifters* are able to handle patterns and multiple alternatives as well.

Rule lifting is not so easy. First of all, all rule alternatives of one function definition can be mutually dependent, and therefore they all have to be lifted together. In Miranda, even functions defined on different levels can depend on each other (see the example after the specification of the lifting algorithm). Therefore, all local functions of a global function have to be lifted simultaneously, taking the scope rules of Miranda into account. Furthermore, local *constant* definitions should *not* be lifted to the global level. However, local constant definitions can be nested (in contrast to node-id definitions in Clean) so they *do* have to be lifted to one and the same *local* level.

A rule lifter for Miranda

First of all, it is assumed that all names are (made) unique. Then, let f be a function, defined on the global level, and g_i be functions (not constants) defined locally at some level of that rule. Furthermore, where in the following definitions **right-hand side** is used, it stands for the expression on the right-hand side of a function definition *with* its local *constant* definitions but *without* the local *function* definitions. With these assumptions define:

$V(g_i)$: the set of *variables* on the right-hand sides of the rule-alternatives of g_i: i.e. all identifiers on the right-hand sides that are *neither defining names* of global definitions nor of local function definitions;

$LF(g_i)$: the set of names of *local functions* applied on a right-hand side of a rule-alternative of g_i;

$BV(g_i)$: the set of *bound variables* of the rule-alternatives of g_i; **bound variables** are the names that occur as (part of the) arguments on the left-hand side of a rule-alternative or as a defining name of a local constant definition that is contained within g_i;

$FV(g_i)$: the complete set of **free variables** of a local function g_i. This set is defined by the following equation:

$FV(g_i) = (V(g_i) \cup All\_FV(g_j)) - BV(g_i)$

in which $All\_FV(g_j)$ is the union of all $FV(g_j)$, $g_j \in LF(g_i)$ and – denotes set difference.

Due to mutual dependencies in LF, the defining equation of FV can be recursive. So possibly a collection of recursive set equations has to be solved. This can be done by repeatedly substituting approximations in the equations until a fixed point is reached, i.e. all resulting sets of two subsequent approximations are the same. When empty sets are used as the initial approximation, the fixed point will be the least solution.

With these definitions the algorithm for lifting is the following:

(1) for each local function g_i determine the sets V, LF, BV and FV by solving the equations;
(2) add the elements of the set of free variables $FV(g_i)$ as extra parameters (in front of the original parameters) to each occurrence of a function g_i and to its definition;
(3) move all local *function* definitions to the global level;
(4) lift all remaining local *constant* definitions to one local level in the rule in which they are defined.

The lifting algorithm is applied in the following function definition:

```
f x y z  =  g s s
            where
            s      =  r + k r
                      where
                      r      =  y + z
            g a b  =  h a a + k t
                      where
                      h 0 c  =  10 + t
                      h d e  =  g (b + e) (y + 1)
                      t      =  a + x
            k q    =  q + z
```

There are three local functions defined (g, h and k) and three local constants (s, r and t). All names are already unique.

Step 1.

$$\begin{array}{lll} V(g) = \{a, t, x\} & V(h) = \{b, y, e, t\} & V(k) = \{z, q\} \\ LF(g) = \{h, k\} & LF(h) = \{g\} & LF(k) = \{\} \\ BV(g) = \{a, b, t\} & BV(h) = \{c, d, e\} & BV(k) = \{q\} \end{array}$$

$$\begin{array}{lcl} FV(g) & = & (V(g) \cup FV(h) \cup FV(k)) - BV(g) \\ FV(h) & = & (V(h) \cup FV(g)) - BV(h) \\ FV(k) & = & V(k) - BV(k) \end{array}$$

So the following recursive set equations have to be solved:

$$\begin{array}{lcl} FV(g) & = & (\{a, t, x\} \cup FV(h) \cup FV(k)) - \{a, b, t\} \\ FV(h) & = & (\{b, y, e, t\} \cup FV(g)) - \{c, d, e\} \\ FV(k) & = & \{z, q\} - \{q\} \end{array}$$

Starting with the empty set as initial approximation for the recursive equation:

$$\begin{array}{lcl} FV_1(g) & = & (\{a, t, x\} \cup FV_0(h) \cup \{z\}) - \{a, b, t\} \\ & = & (\{a, t, x\} \cup \{z\}) - \{a, b, t\} \\ & = & \{x, z\} \\ FV_1(h) & = & (\{b, y, e, t\} \cup FV_0(g)) - \{c, d, e\} \\ FV_1(h) & = & (\{b, y, e, t\}) - \{c, d, e\} \\ & = & \{b, y, t\} \\ FV(k) & = & \{z\} \end{array}$$

The second subsequent approximation will be:

$$\begin{array}{lcl} FV_2(g) & = & (\{a, t, x\} \cup FV_1(h) \cup \{z\}) - \{a, b, t\} \\ & = & (\{a, t, x\} \cup \{b, y, t\} \cup \{z\}) - \{a, b, t\} \\ & = & \{x, y, z\} \\ FV_2(h) & = & (\{b, y, e, t\} \cup FV_1(g)) - \{c, d, e\} \\ & = & (\{b, y, e, t\} \cup \{x, z\}) - \{c, d, e\} \\ & = & \{b, y, t, x, z\} \\ FV(k) & = & \{z\} \end{array}$$

The result of the third subsequent approximation equals the second:

$$\begin{array}{lcl} FV_3(g) & = & (\{a, t, x\} \cup FV_2(h) \cup \{z\}) - \{a, b, t\} \\ & = & (\{a, t, x\} \cup \{b, y, t, x, z\} \cup \{z\}) - \{a, b, t\} \\ & = & \{x, y, z\} \\ & = & FV_2(g) \\ FV_3(h) & = & (\{b, y, e, t\} \cup FV_1(g)) - \{c, d, e\} \\ & = & (\{b, y, e, t\} \cup \{x, y, z\}) - \{c, d, e\} \\ & = & \{b, y, t, x, z\} \\ & = & FV_2(h) \\ FV(k) & = & \{z\} \end{array}$$

So the fixed point solution is:

```
FV (g)  =  {x, y, z}
FV (h)  =  {b, y, t, x, z}
FV (k)  =  {z}
```

Step 2. To all occurrences of g, h and k the variables of the corresponding FV set are added as parameters in front of the original parameters:

```
f x y z  =  g x y z s s
            where
            s           =  r + k z r
                           where
                           r              =  y + z
            g x y z a b =  h b y t x z a a + k z t
                           where
                           h b y t x z 0 c =  10 + t
                           h b y t x z d e =  g x y z (b + e) (y + 1)
                           t               =  a + x
            k z q       =  q + z
```

Step 3. All local function definitions are made global; the result will be:

```
f x y z         =  g x y z s s
                   where  s  =  r + k z r
                                where  r  =  y + z
g x y z a b     =  h b y t x z a a + k z t
                   where  t  =  a + x
h b y t x z 0 c =  10 + t
h b y t x z d e =  g x y z (b + e) (y + 1)
k z q           =  q + z
```

Step 4. Finally, all local constant definitions are lifted to the same local level:

```
f x y z         =  g x y z s s
                   where  s  =  r + k z r
                          r  =  y + z
g x y z a b     =  h b y t x z a a + k z t
                   where  t  =  a + x
h b y t x z 0 c =  10 + t
h b y t x z d e =  g x y z (b + e) (y + 1)
k z q           =  q + z
```

9.5 Translating the desugared language into Clean

The subset of Miranda resulting from the program transformations described in the previous sections is chosen in such a way that the translation of this subset into an equivalent Clean program is relatively straightforward: it is reduced to merely a syntax transformation. Expressions will be represented by graphs, and function definitions become rewrite rules. Variables in functions are represented by node identifiers. So in this way *sharing* of computation is actually introduced.

Furthermore, local constant definitions are syntactically transformed into node definitions, yielding again sharing of computation. For each kind of Miranda type there is an equivalent in Clean via a simple syntactic transformation that is not given here (see Chapter 8).

From function definitions to rewrite rules

The following syntax transformations are needed to change a function definition in the Miranda subset into a rewrite rule in Clean:

- Rename every function in such a way that it starts with an upper case character, to make it a proper Clean symbol (be careful to cause no name conflicts with constructors); also rename the standard basic functions to their Clean δ-rule equivalent.
- Add a ';' to the end of each function alternative.
- Replace the 'where' symbol by a ',', replace the '=' by a '->' where it separates left- and right-hand sides in a rule definition, but by a ':' where it is used to separate sides in a local constant definition;; in guards: replace 'if' by 'IF' and remove ',' and 'otherwise' where necessary; let introduced node definitions be preceded with a ','.
- Remove redundant brackets.

The example used in the section on rule lifting transformed into Clean:

```
F x y z         ->   G x y z s s,
                     s: + r (K z r),
                     r: + y z;
G x y z a b     ->   + (H b y t x z a a) (K z t),
                     t: + a x;
H b y t x z 0 c ->   + 10 t;
H b y t x z d e ->   G x y z (+ b e) (+ y 1);
K z q           ->   + q z;
```

Numerals

In Miranda, there is no upper bound to the numerals that can be used in arithmetic. In Clean however, the standard integer arithmetic available on the machine is used. So if the reach of this arithmetic is not sufficient one can either use Clean reals (floating-point numbers) or one has to define a suitable module that implements Miranda-like unbounded arithmetic on numbers.

9.6 The exploitation of sharing

The generation of sharing and of cycles is, in particular, very important, since it can greatly improve the efficiency of the program, effectively reducing the computational complexity of the specification. In the previous sections local constant definitions were introduced when possible such that expressions are not unnecessarily duplicated and become shared in Clean. In this section three important transformations are discussed that carry the exploitation of sharing even further: *sharing of common subexpressions*, *creating cycles* and *sharing results of partial applications*.

Find common subexpressions

For subexpressions that occur more than once on a right-hand side an extra constant definition can be inserted which induces sharing in Clean.

Sharing common subexpressions on the right-hand side:

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

This can be transformed into:

```
f x     =  s + s     where   s  =  x + 1
```

This will lead to sharing the expression x + 1 in Clean.

It is also possible to make use of the occurrence of common subexpressions on the left- and right-hand sides. This is the case when a term is used as a pattern while this same term is also used in the function body. In Clean, rebuilding of the graph for such a term can be avoided by explicitly specifying a node-id for the term on the left-hand side and by using that node-id on the right-hand side instead of duplicating the term again.

Consider the Miranda definition:

```
f (Cons 1 (Cons a Nil))     x  =  Cons 1 (Cons a Nil)
f y                         x  =  x
```

In Clean, rebuilding the graph Cons 1 (Cons a Nil) can be avoided as follows:

```
F y : (Cons 1 (Cons a Nil))   x  ->    y ;
F y                           x  ->    x ;
```

Create cycles

The transformations described in the previous sections generate cycles in Clean if a local constant definition is recursive.

A recursive local constant definition generating a cycle in Clean:

```
f g   =  map g ones    where    ones = Cons 1 ones
```

Since the local constant definition is translated directly to a node-id definition this will create a cycle in Clean:

```
F g   =  Map g ones,
         ones: Cons 1 ones;
```

Generating cycles for global constant definitions can be achieved by generating extra local constant definitions.

A recursive global constant definition in Miranda:

```
ham  =  1 : merge (map (* 2) ham) (map (* 3) ham)
```

A cycle will be generated in Clean if this definition is transformed into

```
ham  =  x   where    x  =  1 : merge (map (* 2) x) (map (* 3) x)
```

Sharing results of partial applications

When in a lazy evaluation of a computation sharing is applied in such a way that even every partial application is evaluated at most once, this evaluation is called **fully lazy**. In general, computation in Clean is not fully lazy since the semantics of Clean does not require any special evaluation behaviour for partial applications.

Take the following Miranda program and its standard transformation to Clean:

```
g x y =  x * x + y
f x   =  x 3 + x 5
h x   =  f (g x)

G x y     ->     + (* x x) y;
```

```
F x    ->   + (x 3) (x 5);
H x    ->   F (G x);
```

Evaluating H 35 the partial application G 35 is constructed. But no special computation is involved for such a partial application. The two calls G 35 3 and G 35 5 are separately computed and * 35 35 will be computed twice. So in some sense the computation involved with the partial application G 35 is done twice. Hence, this evaluation is not fully lazy in Clean.

It is, however, possible to transform Miranda in such a way into Clean rules that effectively the Miranda computation is performed in a fully lazy manner. This can be achieved by generating *super-combinators* (Hughes, 1982). **Super-combinators** have the property that evaluation by standard graph rewriting is fully lazy.

In Clean, all classes of combinatory expressions can be expressed. Hence it is also possible to specify or generate super-combinators, which can be done as follows. First, the largest subexpressions that only depend on one argument of the function, the **maximal free expressions**, are determined. Then, for each maximal free expression a new function is defined of one argument, with as body the maximal free expression.

In the example above x * x is a maximal free expression in the function g. So a new function is defined: g' x = x * x.

Furthermore, a new function is defined with as body the body of the original function in which every maximal free expression is replaced by a fresh variable. In the parameter list of the new function such a fresh variable replaces the argument on which the corresponding maximal free expression depends. Finally, the original function body is adjusted, calling the newly created functions.

After defining the new function g'' and adjusting the definition of g the final set of function definitions becomes:

```
g' x     =  x * x
g'' z y  =  z + y
g a      =  g'' (g' a)
f x      =  x 3 + x 5
h x      =  f (g x)
```

The main drawback of super-combinators is the huge number of combinator rules that are generated in this way. Besides that, they are only of use if there really are such partially applied shared functions. Even in that case it is doubtful whether the additional effort needed to maintain the sharing will be less than the resulting increase of execution speed. If the choice is made to transform a Miranda program into a super-combinator program this transformation has to be done before the rule lifting, and on local definitions the algorithm for generating super-combinators has to be slightly adapted. An example is given below.

Consider, for instance, the following definition:

```
f x    =  g 3 + g 5    where   g y       =  x * x + y
```

It can be translated into

```
f x    =  g 3 + g 5    where   g'        =  x * x
                               g'' z y   =  z + y
                               g         =  g'' g'
```

The definitions of g' and g are local constant definitions to increase sharing. The definition of g'' will be lifted to the global level afterwards.

Summary

- A Miranda program can be translated into an equivalent Clean program in the following way: first, a Miranda program is translated into an equivalent program in a Clean-like Miranda subset, then the program in this subset is transformed via simple syntactical rules into an equivalent Clean program.
- Most of these transformations are straightforward. The more difficult transformations are:
  - the translation of *ZF-expressions*;
  - the *lifting* of local definitions to one global level.
- The transformations give (as a side-effect) a better understanding of the precise meaning of many constructs in Miranda. One of the reasons for this is that by defining the transformation into a graph rewriting language one *automatically defines* the places where *sharing* is used in the computation.
- During the transformations *sharing* can be *introduced* in several ways: by introducing local constant definitions, by sharing common subexpressions, by creating cycles and by generating super-combinators. Sharing can greatly improve the efficiency of the computation.

EXERCISES

9.1 Use both of the given translation schemes for *ZF*-expressions to transform:

(a) `f = [ a | a,b <- [0..100]; a = b * b ]`

(b) `cartprod x y = [ (a,b) | a <- x; b <- y ]`

9.2 Apply the given rule lifting algorithm to:

```
f x   = h (a + a)
        where   a       = g x y
                          where   g      = h
                                  y      = a * a
                h p q   = p + q + x
```

9.3* Translate the exercises of Chapter 2 to Clean. Explain which transformations have been used. Measure the time–space behaviour of the original programs and the Clean programs. Explain the differences.