Part 2
Models of computation

This part contains an overview of three important models of computation: λ-calculus, term rewriting systems and graph rewriting systems. In three consecutive chapters these models are compared with respect to their ability to serve as the basic model for functional languages and their implementation.

The best-known model is the **λ-calculus** introduced by Church (1932; 1933). This calculus is a model of computation based on consistent substitution of variables in nameless functions. Evaluation is done by two rules only: α-conversion and β-reduction. Many semantic properties that are deduced for the λ-calculus immediately carry over to functional languages. Therefore, the λ-calculus is studied intensively and used as *the* basic model of computation. But, although the model is very suitable as a model to study the basic semantics of functional languages, it is in our opinion not very suitable to study implementation aspects of functional languages.

Term rewriting systems (Huet and Oppen, 1980; Klop, 1992) form a model of computation based on pattern matching on terms that do not contain variables. In term rewriting systems evaluation is not done using a fixed set of reduction rules but in every term rewriting system a set of rules is specified that define how terms can be rewritten. Semantically, a functional program can be seen as a specific term rewriting system in a special class of term rewriting systems. Implementations of functional languages are usually more close to term rewriting systems than to the λ-calculus.

Graph rewriting systems (Barendregt *et al.*, 1987a; 1987b) extend term rewriting systems with the notion of sharing such that a model of computation is obtained based on pattern matching on graphs not containing variables. Every *efficient* implementation of functional languages avoids duplication of work by *sharing* subexpressions. Therefore, graph rewriting systems are considered to be best suited to serve as the basic model for functional languages and their implementation.

Chapter 3
The λ-calculus

The theory of the *λ-calculus* has been introduced by Church (Church, 1932; 1933) to describe the properties of mathematical functions using a textual substitution model, and has since been investigated further. All functions that are effectively computable can be expressed in the calculus (see also Section 1.1.2). Another important property of the calculus is the *Church–Rosser property*, which states that alternative evaluation orders cannot yield different results. Many of the theoretical properties of functional programming can be deduced from the (known) theoretical properties of the λ-calculus. The semantics of a high-level functional language can be explained by giving the translation schemes to the calculus. The close relation between the (relatively old) calculus and (relatively new) functional languages makes it clear why anyone interested in the underlying basic concepts of functional languages should study the λ-calculus.

In Sections 3.1–3.6 a short overview is given of the theory of the λ-calculus. Readers interested in a detailed overview of the λ-calculus (including proofs of the theorems discussed in this chapter) are referred to the book of Barendregt (1984).

One can regard the λ-calculus also as a very simple low-level functional programming language that is as equally powerful as a high-level functional language such as Miranda. One often regards these high-level functional programming languages as sugared versions of the λ-calculus. It is possible to translate programs written in an arbitrary functional programming language into an equivalent expression in the λ-calculus. In Section 3.7 it is shown how, in principle, Miranda functions can be translated into equivalent λ-expressions.

Since the λ-calculus can be considered as a basic functional language, a λ-calculus machine should be suited to executing all functional programs. However, as discussed in Section 3.8 an efficient realization of a λ-calculus machine is not so easy to achieve. The computational models that are introduced in Chapters 4 and 5 are more closely related to functional languages and the way they are generally implemented.

3.1 λ-expressions

In the λ-calculus one can write down expressions (**λ-terms** or **λ-expressions**) in which anonymous functions can be defined together with the expressions on which these functions have to be applied. Furthermore there are a couple of evaluation rules called **reduction rules** (or **rewrite rules**) defined in the calculus that specify how a λ-expression can be transformed into an equivalent λ-expression, i.e. an expression denoting the same value. In this section the λ-expressions are introduced, in the next section reduction rules are discussed.

Syntax definition

Syntax of λ-expressions in BNF-like notation (literals between quotes):

```
Lambda-expression   = Lambda-abstraction    | Application
                    | Variable              | Constant ;
Lambda-abstraction  = '(' 'λ' Variable '.' Lambda-expression ')' ;
Application         = '(' Lambda-expression Lambda-expression ')' ;
```

Further syntactical aspects are the following: variables will always start with a lower case character and constants with an upper case character or some special character (a digit, an asterisk, etc.); in general, the outermost parentheses are left out; as in Miranda, application (juxtaposition) is left associative; λ-abstraction (.) is right associative; application has a stronger binding than abstraction; and parentheses can be omitted when they are redundant (see Section 3.1.2).

3.1.1 λ-abstraction

With λ-abstraction **anonymous functions** are specified, i.e. functions that have no name. A **λ-abstraction** is a λ-expression with which one can define an anonymous function of a certain variable. This variable can be considered as the **formal argument** of the function. The variable is called the **bound variable** of the abstraction. The general form of a λ-abstraction is:

λ-abstraction:

λ	v	.	expression
↑	↑	↑	↑
a function	of variable v	that	returns this value

The expression behind the dot is again a λ-expression that expresses the **body** of the function. In general, the value denoted by the expression will depend on the bound variable.

Examples of λ-abstraction. It is assumed that Inc and * are externally defined functions (see the end of this section on constants):

λ	x	.	Inc x
↑	↑	↑	↑
the function of	x	that	increments x

λ	x	.	* x 2
↑	↑	↑	↑
the function of	x	that	multiplies x by 2

In the λ-calculus only functions with one argument can be specified. Functions with more than one argument are simulated by using a cascade of (higher order) functions each with exactly one argument (see also Section 1.7.2). A higher order function yielding a function can be defined by specifying another λ-abstraction as function body.

Defining higher order functions:

λ	x	.	(λ	y	.	+ x y)
↑	↑	↑	↑	↑	↑	↑
the function of	x	that	yields			
			the function of	y	that	adds x and y

This is equivalent to the Miranda definition:

```
f x y  =  x + y
```

In Miranda most functions have a name, whereas in the λ-calculus all functions are anonymous; in the λ-calculus all functions are specified in prefix notation.

3.1.2 Function application

As in functional programming languages, application is fundamental in the λ-calculus and it is denoted simply by juxtaposition.

Function application:

```
f                          a
↑                          ↑
f   has to be applied to   a
```

If f is a function (λ-abstraction), the expression to which the function is applied is called the **actual argument** of the function.

Example of function application:

```
(λ x.Inc x)                                              5
↑                                                        ↑
this function of x   has to be applied   to the constant 5

((λx.(λy.+ x y))                         4)                      5
↑                                        ↑                       ↑
this function of x   has to be applied to 4
and the result which is a function of y        has to be applied to   5
```

Notational conventions

Parentheses can be left out when they are redundant. The main cases are given below with other ways to abbreviate λ-expressions. The symbol '≡' means: 'is an abbreviation for'.

$\lambda x.exp_1\ exp_2$	≡	$\lambda x.(exp_1\ exp_2)$
$exp_1\ exp_2\ exp_3$	≡	$(exp_1\ exp_2)\ exp_3$
$\lambda x.\lambda y.exp$	≡	$\lambda x.(\lambda y.exp)$
$\lambda xy.exp$	≡	$\lambda x.\lambda y.exp$

Although functions have no name in the λ-calculus it is sometimes convenient to use an *abbreviation* as a synonym for a complex λ-expression. Such an abbreviation is generally an identifier in upper case characters. Do not confuse this notation with a (possibly recursive) function call in Miranda. The identifier is just an abbreviation for the corresponding λ-expression and nothing else.

A common abbreviation for a specific λ-expression:

TWICE ≡ (λf.λx.f (f x))

3.1.3 Constants

In the original λ-calculus model, **pure** λ-calculus, constants were not included. Nevertheless, any computable function can be expressed in pure λ-calculus. For notational convenience however, we do include **constants** in the model to represent constant values (such as numbers) or externally defined functions (such as arithmetical functions). Such a function is called a **δ-rule** or a **δ-function** (see Section 3.2.3).

Some λ-expressions containing constant values (2, 3 and 5) and externally defined functions (+, *, –):

```
5
+ 2 3
λx.* (+ 2 x) (– y x)
```

3.1.4 Representing λ-expressions as trees

Sometimes a λ-expression is also represented as a tree, which clarifies the structure of an expression.

Figure 3.1 A tree representation of, respectively, λ-abstraction (λv.E) and application (F A).

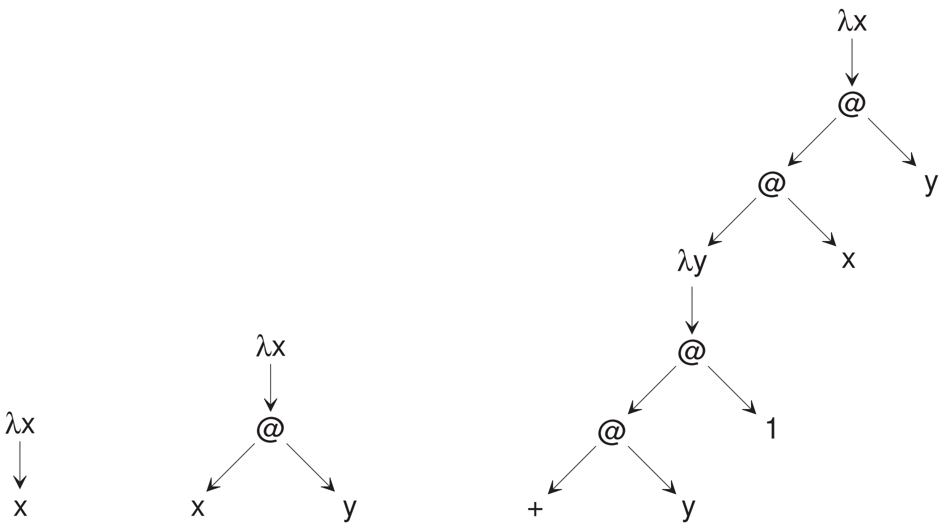

Figure 3.2 Tree representations of λx.x, λx.x y and λx.(λy.+ y 1) x y.

As shown in Figures 3.1 and 3.2, a λ-abstraction λv.E is represented in such a tree by a vertical line with on the top the λv and on the bottom the tree representing E, the function body of the anonymous function of v. An application of an expression F applied to an expression A is represented by a binary tree with @ (indicating an application) as root, a left subtree representing F and a right subtree representing A.

3.1.5 Bound and free variables

A λ-expression can contain variables. Traditionally, a one-character identifier (x, y, z, v, w, a, b, c, ...) is used to denote a variable. There are *free* variables and *bound* variables. Informally it can be stated that a variable occurs *bound* in a λ-expression if and only if there is an enclosing λ-abstraction that *binds* it, i.e. the variable is the formal argument of an enclosing λ-abstraction. Otherwise, the variable occurs *free* in this λ-expression. The notion of free and bound variables plays an important role in the reduction rules (see Section 3.2).

The precise definition of bound and free variables can easily be given using the tree representation of a λ-expression:

- An occurrence of a variable, say x, is **bound** if on the path starting from the occurrence of the variable (this will always be a leaf in the tree) going up to the root of the tree a λ is encountered with a variable with the same name you started with, i.e. λx. The first occurrence of λx in the path from the leaf up to the root is said to **bind** the occurrence of x.
- An occurrence of a variable is **free** if it is not bound.

Follow this algorithm for the variables in the trees given in Figure 3.2 and compare the results with the examples below.

Illustrating free and bound variables:

```
x                        λx.  x      y
↑                             ↑      ↑
free variable                 bound  free

λx.  x                   λx.(λy.  +  y  1)   x      y
     ↑                               ↑       ↑      ↑
     bound variable                  bound   bound  free
```

Note that a variable can occur bound as well as free in one and the same expression.

3.1.6 α-conversion

In the λ-calculus the term λx.x is considered to be equivalent to λz.z. A λ-expression can be converted into another equivalent one by renaming *bound* variables. It can be compared to a function definition in Miranda in which the names of the formal arguments can be chosen freely (without changing the semantics) as long as they do not conflict with other names. Expressions that can be made textually equivalent by renaming bound variables are called **α-convertible** or **alphabetically equivalent**. The renaming process is called **α-conversion**.

If names are changed, a new name which is chosen must not be in *conflict* with one of the old names. Therefore α-conversion only allows us to change the name of a bound variable under the following conditions:

Definition of α-conversion

$$\lambda x.E =_\alpha (\lambda y.E\ [x := y])$$

where E [x := A] means E in which all free occurrences of x (which means occurrences of x not bound by other λs inside E) are replaced by y. The variable y has to be a **fresh variable**, i.e. y does not occur free in E *and* y does not get bound in E by the substitution.

Here are some examples of valid and invalid α-conversions on λ-expressions (the α-converted variables are underlined):

Examples of valid α-conversions:

$\underline{\lambda x}$.Succ x	$=_\alpha$	λy.Succ y
$\underline{\lambda x}$.(λy.y) x	$=_\alpha$	λy.(λy.y) y
(λx.$\underline{\lambda y}$.* x y) y	$=_\alpha$	(λx.λz.* x z) y

Examples of invalid α-conversions:

$\underline{\lambda x}$.Succ x y	$\neq_\alpha$	λy.Succ y y, because y occurs free in Succ x y
$\underline{\lambda x}$.(λy.y x)	$\neq_\alpha$	λy.(λy.y y), because y will get bound in λy. y x
λy.y $\underline{x}$	$\neq_\alpha$	λy.y z, because x is not bound

One has to be careful because the same name does not always stand for the same variable. Whether or not the same name stands for the same variable is determined by the context: when the variables are bound by the same λ or when they are both free then they also stand for the same variable. From now on, two λ-expressions are considered to be **equal** if they are α-convertible to each other.

3.2 Reduction rules for λ-expressions

There are a number of *reduction rules* or *rewrite rules* that define how λ-terms can be converted to other, equivalent, λ-terms. Two transformation rules are treated in this section: *β-reduction* and *δ-reduction.*

3.2.1 β-reduction

β-reduction is a reduction rule that describes how to evaluate an application of a function (a λ-abstraction) to its actual argument. In the λ-calculus, β-reduction states that any function applied to an argument forms a **β-redex**, a **red**ucible (sub)**ex**pression according to the β-reduction rule. The result of β-reduction is the body of the λ-abstraction in which all occurrences of the formal argument are *textually substituted* (replaced, reduced, rewritten) by the actual argument on which the function is applied. Hence, all occurrences of the variable that are bound by the corresponding λ are replaced by the actual argument. The replacement is also called a **β-reduction step** or a **rewrite step**.

Definition of β-reduction

$$(\lambda x.E)\ A \quad \rightarrow_\beta \quad E\ [x := A]$$

where E [x := A] means E in which all free occurrences of x are replaced by expression A. Furthermore, as explained below, it is not allowed that one of the free variables in A becomes bound when substituted in E.

As an example some λ-terms are given together with some β-reduction steps. As usual, the chosen redexes are underlined.

β-reduction on a simple λ-term:

$$\underline{(\lambda x.x)\ 0} \quad \rightarrow_\beta \quad 0$$

Using currying to simulate functions with more than one argument:

$$\underline{(\lambda x.(\lambda y.x)\ )\ 0}\ 1 \quad \rightarrow_\beta \quad \underline{(\lambda y.0)\ 1} \quad \rightarrow_\beta \quad 0$$

In the example above an anonymous function is returned after application of the function to the first argument. When λ-calculus is used to model currying, a new anonymous function is explicitly created each time a curried function is applied to another one of its arguments.

A function with another function as its argument:

$$\underline{(\lambda f.f\ 0)\ (\lambda x.x)} \quad \rightarrow_\beta \quad \underline{(\lambda x.x)\ 0} \quad \rightarrow_\beta \quad 0$$

A β-redex can easily be recognized in a tree representation. It is a function application with as left subtree a λ-abstraction (see Figure 3.3).

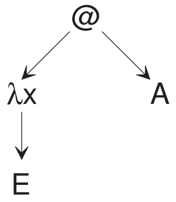

Figure 3.3 A β-redex, the application (@) of a λ-abstraction (λx.E) to an expression (A).

3.2.2 α-conversion and β-reduction

According to the definition, β-reduction is only valid if the free variables that are present in the actual argument remain free after substitution.

Example of an illegal β-reduction: after substitution, the free variable y has become a bound variable. The name of the free variable 'conflicts' with the name of the bound variable.

```
(λx.λy.x   y)      y  ↛β  λy.  y  y
           ↑       ↑           ↑  ↑
           bound   free        both bound
```

The name conflict can be avoided by performing α-conversion first, replacing the function of y by an equivalent function of, say z.

$$(\lambda x.\lambda \underline{y}.x\ y)\ y \ =_\alpha\ \underline{(\lambda x.\lambda \underline{z}.x\ z)\ y} \ \rightarrow_\beta\ \lambda z.y\ z$$

Note that the illegal substitution above would have changed the meaning of the expression: $\lambda y.y\ y \neq_\alpha \lambda z.y\ z$. For the same reason it is not allowed to avoid the name conflict by renaming the free variable y.

The examples show why β-reduction is only allowed when the substitution is 'safe', i.e. when no names of free variables get bound through the substitution. From now on it will be assumed that α-conversion will always be performed where needed to make β-reduction possible.

α-conversion needed during β-reduction

At first sight it may seem possible to avoid name conflicts by choosing different names for the different variables in the initial λ-expression. However, name conflicts can arise during β-reduction even if in the initial expression all different variables had different names.

Consider the following λ-expression: $\mathsf{TWICE} \equiv (\lambda f.\lambda x.f\ (f\ x))$. α-conversion is necessary during β-reduction to avoid name conflicts:

$$
\begin{array}{llll}
\underline{(\lambda y.y\ y)\ \mathsf{TWICE}} & & & \\
\to_\beta & \mathsf{TWICE}\ \mathsf{TWICE} & \equiv & \underline{(\lambda f.\lambda x.f\ (f\ x))\ \mathsf{TWICE}} \\
\to_\beta & \lambda x.\mathsf{TWICE}\ (\mathsf{TWICE}\ x) & \equiv & \lambda x.\mathsf{TWICE}\ (\underline{(\lambda f.\lambda x.f(f\ x))\ x})
\end{array}
$$

In this term the underlined subterm is not allowed to be β-reduced since otherwise the free variable x would get bound. So first the names have to be changed: α-conversion is applied, renaming all occurrences of x that are bound by the innermost λx:

$$
\begin{array}{llll}
\lambda x.\mathsf{TWICE}\ ((\lambda f.\underline{\lambda x}.f(f\ x))\ x) & =_\alpha & \lambda x.\mathsf{TWICE}\ (\underline{(\lambda f.\lambda y.f(f\ y))\ x}) \\
\to_\beta\quad \lambda x.\mathsf{TWICE}\ (\lambda y.x(x\ y)) & &
\end{array}
$$

The example shows that changing of names can in general be necessary after any β-reduction because as a consequence of (self-)application new conflicts can come up. Note that α-conversion is not considered to be an explicit step in the reduction process: it is a renaming process that does not change the structure of an expression and it is assumed to be performed implicitly where needed.

Practical ways to deal with α-conversion are explained in Section 3.8. Furthermore, in that section it will be shown that there are certain conditions under which α-conversion is not needed at all.

3.2.3 δ-reduction

As explained in Section 3.1, the impure λ-calculus can contain externally defined functions which are called **δ-rules** or **δ-functions**. Such δ-functions can be defined on predefined constants, such as, for instance, constants that represent numbers. Although it is possible to define most of these functions and constants directly in the pure λ-calculus (see Section 3.7), it is sometimes more convenient for readability and illustration purposes to assume the availability of these external functions. δ-rules are sometimes also introduced to increase efficiency: with one *δ-reduction* one can obtain the same result that otherwise would have cost many *β-reductions* in the pure λ-calculus.

If a δ-function is applied to the number of arguments it needs and these arguments are of the correct type, the corresponding subexpression is called a **δ-redex**. If the external function is applied it is called a **δ-reduction step**.

For example, assume that an external δ-rule + is defined which requires that its arguments are constants of a specific type, e.g. constant numbers. An application of the function + is not a δ-redex until this restriction is satisfied. This implies that one first has to reduce the arguments of + to obtain the required constant numbers, before this δ-rule can be applied.

Example of a δ-reduction step, assuming that +, 2, 3 and 5 are externally defined δ-rules:

$\underline{+\ 2\ 3} \quad \rightarrow_{\delta+} \quad 5$

The function + cannot be applied until its arguments are constant numbers:

$+\ (\underline{*\ 1\ 2})\ 3 \quad \rightarrow_{\delta*} \quad \underline{+\ 2\ 3} \quad \rightarrow_{\delta+} \quad 5$

In principle, it is allowed that external functions investigate the internal structure of their arguments. This means that, for instance, it is possible to have δ-rules that can determine whether two arbitrary λ-expressions are syntactically equal. Such a function cannot be defined in the pure λ-calculus. Hence with δ-rules one can introduce arbitrary computational power in the λ-calculus (see also Section 3.7).

3.3 Reduction sequences and normal forms

A **reduction sequence** of a λ-expression consists of zero or more reduction steps (rewrite steps) performed on that expression. If expression E_1 can be reduced to expression E_2 by a reduction sequence, E_2 is called a **reduct** of E_1. This is denoted as: $E1 \rightarrow_* E2$. So $\rightarrow_*$ is the reflexive transitive closure of $\rightarrow$.

A λ-expression that does not contain a redex is in **normal form**. A λ-expression *has* a normal form if there exists some sequence of reduction steps starting with this expression that results in an expression in normal form. Not all expressions have a normal form. In that case there exists an infinite reduction sequence (see the example below).

A reduction sequence (of length 1), where the reduct is in normal form:

$\lambda x.\underline{(\lambda y.y)\ (\lambda x.x)} \quad \rightarrow_\beta \quad \lambda x.(\lambda x.x)$

Example of an expression that has no normal form :

$\underline{(\lambda x.x\ x)\ (\lambda x.x\ x)} \quad \rightarrow_\beta \quad \underline{(\lambda x.x\ x)\ (\lambda x.x\ x)} \quad \rightarrow_\beta \quad \cdots$

Note the difference between being in normal form and having a normal form:

x	is in normal form, because it does not contain a redex.
(λx.x)(λx.x)	is not in normal form, but it has a normal form: λx.x.
(λx.x x)(λx.x x)	is not in normal form, and it has no normal form.

3.4 Properties of the λ-calculus

3.4.1 Church–Rosser property

A λ-expression can contain several redexes:

Example: the following expression:

$(\lambda x.(\lambda y.y)\ (\lambda z.z))\ (\lambda u.u)$

contains two redexes, namely: $(\lambda x.\underline{(\lambda y.y)\ (\lambda z.z)})\ (\lambda u.u)$
and $\underline{(\lambda x.(\lambda y.y)\ (\lambda z.z))\ (\lambda u.u)}$

If more than one redex occurs in an expression, different reduction sequences are possible. Hence, the available redexes can be reduced in different **reduction orders**. An interesting question is: is it possible that an expression has more than one normal form depending on the chosen order of reduction?

Two alternative reduction orders leading to the same result. First method:

$$(\lambda x.\underline{(\lambda y.y)\ (\lambda z.z)})\ (\lambda u.u) \quad \rightarrow_\beta \quad \underline{(\lambda x.\lambda z.z)\ (\lambda u.u)} \quad \rightarrow_\beta \quad \lambda z.z$$

or, with an alternative reduction order

$$\underline{(\lambda x.(\lambda y.y)\ (\lambda z.z))\ (\lambda u.u)} \quad \rightarrow_\beta \quad \underline{(\lambda y.y)\ (\lambda z.z)} \quad \rightarrow_\beta \quad \lambda z.z$$

Two different reduction sequences: one leads to a normal form, the other is an infinite reduction sequence:

$$\underline{(\lambda y.\lambda z.z)((\lambda x.x\ x)(\lambda x.x\ x))} \rightarrow_\beta \quad \lambda z.z$$

$$(\lambda y.\lambda z.z)(\underline{(\lambda x.x\ x)(\lambda x.x\ x)}) \rightarrow_\beta \quad (\lambda y.\lambda z.z)(\underline{(\lambda x.x\ x)(\lambda x.x\ x)}) \quad \rightarrow_\beta \quad \ldots$$

The examples show that not all reduction orders lead to a normal form, not even if the initial expression has a normal form.

A reduction system (such as the λ-calculus) has the **Church–Rosser property** (Church and Rosser, 1936) or is **confluent** if and only if for every two 'divergent' reduction sequences $E \rightarrow_* E_1$ and $E \rightarrow_* E_2$ there are two 'convergent' reduction sequences $E_1 \rightarrow_* E_3$ and $E_2 \rightarrow_* E_3$. The Church–Rosser property is also called the **diamond property** (Figure 3.4).

Theorem: The λ-calculus satisfies the Church–Rosser property.

The Church–Rosser theorem has a very important consequence: in a reduction system, in which the Church–Rosser property holds, every expression has at most one normal form: *the normal form is unique*. Assume that an expression has two normal forms. The Church–Rosser property implies that there must be a common reduct to which both these normal forms can be reduced. Since both expressions are already in normal form, they must have been the same.

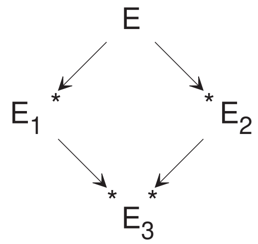

Figure 3.4 The Church–Rosser property.

If a λ-term can be reduced by several reduction sequences and several of these sequences lead to a normal form, the result is always the same, independent of the reduction order chosen. λ-expressions that have a common reduct are ***β*-convertible** ($=_\beta$). Due to the Church–Rosser property the following holds: if $E_1 \rightarrow_* E_3$ and $E_2 \rightarrow_* E_3$ then $E_1 =_\beta E_2 =_\beta E_3$ and if $E_1 =_\beta E_2$ then there is an E_3 such that $E_1 \rightarrow_* E_3$ and $E_2 \rightarrow_* E_3$.

3.4.2 Computability

Conjecture (Church's thesis)**:** The class of effectively computable functions, i.e. functions that intuitively can be computed, is the same as the class of functions that can be defined in λ-calculus (Church, 1936).

Theorem: The class of Turing computable functions is the same as the class of λ-definable functions (Turing, 1937).

So the power of Turing machines is the same as the power of the λ-calculus. Both models capture the intuitive idea of computation.

3.5 Reduction strategies

A **reduction strategy** in the λ-calculus is a mathematical function that prescribes which redex in a given, arbitrary λ-expression has to be rewritten. When it is repeatedly applied to a λ-expression and its reducts, a strategy defines the **order** in which redexes are reduced. In this way a (possibly infinite) reduction sequence can be defined. A **normalizing reduction strategy** is a reduction strategy that results, after repetitive application, in the normal form, if this normal form exists.

3.5.1 Normal order reduction

In the **normal order** reduction strategy the leftmost-outermost redex in a given expression is rewritten and therefore it is also known as the **leftmost-outermost** reduction strategy, or as the **leftmost** strategy. The **leftmost-outermost redex** of a λ-expression is the redex that is not surrounded by any other redex (outermost) and is to the left of all the other outermost redexes in that expression (leftmost).

> The notion leftmost-outermost redex:
>
> ((λx.x x) (λy.y)) ((λx.x x) (λx.x x))
> $\uparrow r_1$ $\uparrow r_2$
>
> r_1 is the leftmost redex, because this redex is to the left of all other redexes in that expression. Furthermore this redex is also an outermost redex, because there is no other redex in this expression that surrounds r_1. Note that r_2 is also an outermost redex, but not the leftmost one.
>
> (λz.(λx x) z) (λz.z)
> $\uparrow r_1$
>
> (λz.(λx x) z) (λz.z)
> $\uparrow r_2$
>
> r_1 is the leftmost-outermost redex. r_2 is not an outermost redex.

Theorem: In the λ-calculus the normal order strategy is a normalizing strategy (Curry and Feys, 1958).
So reducing normal order will always yield the normal form if it exists.

3.5.2 Applicative order reduction

The **applicative order** reduction strategy prescribes rewriting the leftmost-innermost redex in a given expression and therefore it is also known as the **leftmost-innermost** reduction strategy, or the **innermost** strategy. The **leftmost-innermost redex** of a λ-expression is the redex that does not contain any other redex (innermost) and is to the left of all the other innermost redexes in that expression (leftmost). Hence, the innermost reduction strategy will always rewrite the argument of a function first, before a β-reduction is performed on that function and its argument.

> Innermost reduction: The third example shows that the innermost reduction strategy is not normalizing (the expression does have a normal form: 0).
>
> (λx.x) ((λz.z z) y) $\rightarrow_\beta$ (λx.x) (y y) $\rightarrow_\beta$ y y
>
> (λx.x) ((λx.x) (λx.x)) $\rightarrow_\beta$ (λx.x) (λx.x) $\rightarrow_\beta$ λx.x

$$(\lambda x.0)\ (\underline{(\lambda x.x\ x)\ (\lambda x.x\ x)}) \quad \rightarrow_\beta \quad (\lambda x.0)\ (\underline{(\lambda x.x\ x)\ (\lambda x.x\ x)}) \quad \rightarrow_\beta \quad \cdots$$

Analogous to leftmost-outermost and leftmost-innermost one can define the notions *rightmost-outermost* and *rightmost-innermost*. To find leftmost, rightmost, innermost or outermost redexes, again the tree representation is very helpful (Figure 3.5).

λ-term as tree: (λx.λy.x y) ((λz.z) ((λx.x) z)) ((λx.x) ((λx.x) ((λx.x) z)))

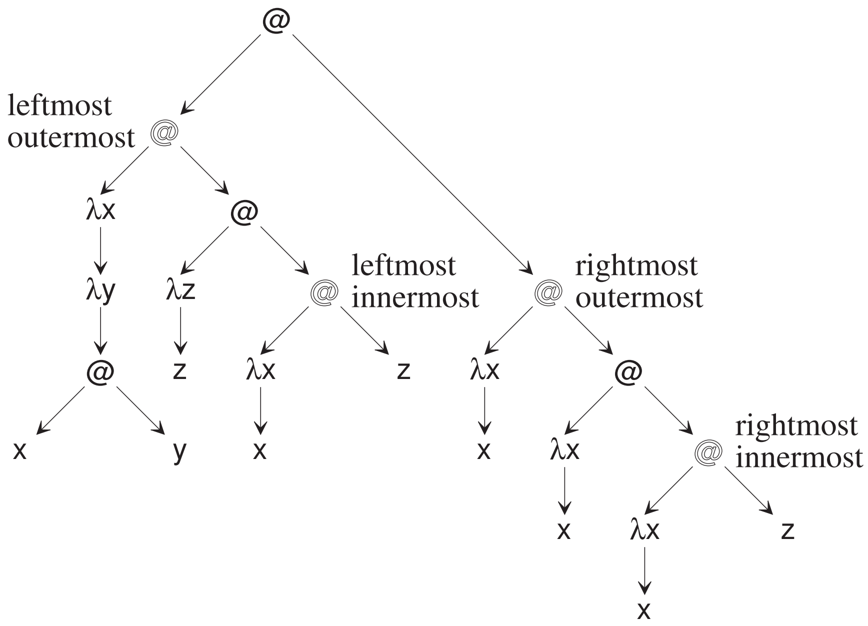

Figure 3.5 Tree showing leftmost, rightmost, innermost and outermost redexes (marked with @).

3.6 Properties of subclasses of the λ-calculus

3.6.1 Combinatory logic

A **combinator** is a λ-expression (a λ-abstraction) without free variables. The theory of such λ-terms is the **combinatory logic** (Curry, 1930).

The combinators S and K:

S ≡ λx.λy.λz.x z (y z)
K ≡ λx.λy.x

Combinators can be regarded as function constants, e.g. S and K are functions with the following property:

S x y z = x z (y z) (distributor)
K x y = x (cancellator)

With these combinators **applicative expressions** can be constructed, i.e. expressions in which only applications (of combinators) occur. It turns out that with only the two combinators, S and K, one can construct a system that has the same power as the pure λ-calculus: one can construct all effectively computable functions with it:

With an applicative expression of the combinators S and K one can build for example the identity combinator (often referred to as I), with the property

I x = x (identity)

Generally, for convenience, the I-combinator is added to the system, although it can be expressed in S and K (e.g. as S K S or S K K).

$\mathsf{I\ e} \equiv \underline{\mathsf{S\ K\ S\ e}} =_{\mathsf{S}} \underline{\mathsf{K\ e\ (S\ e)}} =_{\mathsf{K}} \mathsf{e}$

It is amazing that with only two combinators, any computable function can be expressed. Intuitively this can be explained as follows. The pure λ-calculus consists of variables, functions and applications. When a function is applied to an argument, the actual argument is substituted for all occurrences of the formal argument, the variable bound by the corresponding λ. So actually all that matters is to get a copy of an expression in the right place. With an S combinator an expression can be duplicated and inserted into two subexpressions. With a K combinator an expression can be removed. With a proper combination of the two one can copy any expression into another until it is in the right place. If a copy arrives at a place where it is not needed, it is simply removed by a K.

Transforming λ-expressions to combinators

An algorithm for translating λ-expressions into an applicative expression of SKI-combinators is presented below.

All translation rules used in this textbook are presented in a functional style. These functions define how a (part of a) language can be translated into another language. The objects of the translation are syntactic objects. To make a translation description in a functional formalism as short and as readable as possible a notational convenience is added that allows specification of arbitrary syntactic objects. To emphasize that these objects are special they are placed between double square brackets (⟦ and ⟧). If one would like to specify these transformations directly in Miranda, one would actually need a data structure (an abstract syntax tree) that is able to represent any sentence in the language one wants to translate. The algorithm for translating λ-expressions into combinators, known as **bracket-abstraction** (probably due to the use of the double square brackets), is defined below.

Algorithm for bracket-abstraction: x and y are variables, c is any constant, E, E_1 and E_2 are λ-expressions. The rules have to be tried in textual order.

$$
\begin{array}{lcl}
\mathrm{LamToSKI}\llbracket \lambda x.x \rrbracket & = & \mathrm{I} \\
\mathrm{LamToSKI}\llbracket \lambda x.y \rrbracket & = & \mathrm{K}\ y \\
\mathrm{LamToSKI}\llbracket \lambda x.c \rrbracket & = & \mathrm{K}\ c \\
\mathrm{LamToSKI}\llbracket \lambda x.E_1\ E_2 \rrbracket & = & \mathrm{S}\ (\mathrm{LamToSKI}\llbracket \lambda x.E_1 \rrbracket)\ (\mathrm{LamToSKI}\llbracket \lambda x.E_2 \rrbracket) \\
\mathrm{LamToSKI}\llbracket \lambda x.E \rrbracket & = & \mathrm{LamToSKI}\llbracket \lambda x.\mathrm{LamToSKI}\llbracket E \rrbracket \rrbracket \\
\mathrm{LamToSKI}\llbracket E_1\ E_2 \rrbracket & = & (\mathrm{LamToSKI}\llbracket E_1 \rrbracket)\ (\mathrm{LamToSKI}\llbracket E_2 \rrbracket) \\
\mathrm{LamToSKI}\llbracket E \rrbracket & = & E
\end{array}
$$

For example, consider the following λ-expression:

$$\mathrm{INV} \equiv \lambda x.\lambda f.f\ x$$

The bracket-abstraction algorithm applied to this λ-expression:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathrm{LamToSKI}\llbracket \underline{\mathrm{INV}} \rrbracket \equiv \underline{\mathrm{LamToSKI}\llbracket \lambda x.\lambda f.f\ x \rrbracket}} \\
\rightarrow & \mathrm{LamToSKI}\llbracket \lambda x.\underline{\mathrm{LamToSKI}\llbracket \lambda f.f\ x \rrbracket} \rrbracket \\
\rightarrow & \mathrm{LamToSKI}\llbracket \lambda x.\mathrm{S}\ (\underline{\mathrm{LamToSKI}\llbracket \lambda f.f \rrbracket})\ (\underline{\mathrm{LamToSKI}\llbracket \lambda f.x \rrbracket}) \rrbracket \\
\rightarrow_* & \underline{\mathrm{LamToSKI}\llbracket \lambda x.\mathrm{S}\ \mathrm{I}\ (\mathrm{K}\ x) \rrbracket} \\
\rightarrow & \mathrm{S}\ (\underline{\mathrm{LamToSKI}\llbracket \lambda x.\mathrm{S}\ \mathrm{I} \rrbracket})\ (\mathrm{LamToSKI}\llbracket \lambda x.\mathrm{K}\ x \rrbracket) \\
\rightarrow & \mathrm{S}\ (\mathrm{S}\ (\underline{\mathrm{LamToSKI}\llbracket \lambda x.\mathrm{S} \rrbracket})\ (\underline{\mathrm{LamToSKI}\llbracket \lambda x.\mathrm{I} \rrbracket}))(\underline{\mathrm{LamToSKI}\llbracket \lambda x.\mathrm{K}\ x \rrbracket}) \\
\rightarrow_* & \mathrm{S}\ (\mathrm{S}\ (\mathrm{K}\ \mathrm{S})(\mathrm{K}\ \mathrm{I}))\ (\mathrm{S}\ (\underline{\mathrm{LamToSKI}\llbracket \lambda x.\mathrm{K} \rrbracket})\ (\underline{\mathrm{LamToSKI}\llbracket \lambda x.x \rrbracket})) \\
\rightarrow_* & \mathrm{S}\ (\mathrm{S}\ (\mathrm{K}\ \mathrm{S})(\mathrm{K}\ \mathrm{I}))\ (\mathrm{S}\ (\mathrm{K}\ \mathrm{K})\ \mathrm{I})
\end{array}
$$

This complicated expression will, when it is applied to two arguments (say X and F), yield the same result as the original λ-expression:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\underline{\mathrm{S}\ (\mathrm{S}\ (\mathrm{K}\ \mathrm{S})(\mathrm{K}\ \mathrm{I}))\ (\mathrm{S}\ (\mathrm{K}\ \mathrm{K})\ \mathrm{I})\ \mathrm{X}}\ \mathrm{F}} \\
=_{\mathrm{S}} & (\underline{\mathrm{S}\ (\mathrm{K}\ \mathrm{S})(\mathrm{K}\ \mathrm{I})\ \mathrm{X}})\ (\mathrm{S}\ (\mathrm{K}\ \mathrm{K})\ \mathrm{I}\ \mathrm{X})\ \mathrm{F} \\
=_{\mathrm{S}} & (\underline{\mathrm{K}\ \mathrm{S}\ \mathrm{X}})\ (\underline{\mathrm{K}\ \mathrm{I}\ \mathrm{X}})\ (\mathrm{S}\ (\mathrm{K}\ \mathrm{K})\ \mathrm{I}\ \mathrm{X})\ \mathrm{F} \\
=_{\mathrm{K,K}} & \underline{\mathrm{S}\ \mathrm{I}\ (\mathrm{S}\ (\mathrm{K}\ \mathrm{K})\ \mathrm{I}\ \mathrm{X})\ \mathrm{F}} \\
=_{\mathrm{S}} & \underline{\mathrm{I}\ \mathrm{F}}\ (\underline{\mathrm{S}\ (\mathrm{K}\ \mathrm{K})\ \mathrm{I}\ \mathrm{X}}\ \mathrm{F}) \\
=_{\mathrm{I,S}} & \mathrm{F}\ ((\underline{\mathrm{K}\ \mathrm{K}\ \mathrm{X}})\ (\mathrm{I}\ \mathrm{X})\ \mathrm{F}) \\
=_{\mathrm{K}} & \mathrm{F}\ (\underline{\mathrm{K}\ (\mathrm{I}\ \mathrm{X})\ \mathrm{F}}) \\
=_{\mathrm{K}} & \mathrm{F}\ (\underline{\mathrm{I}\ \mathrm{X}}) \\
=_{\mathrm{I}} & \mathrm{F}\ \mathrm{X}
\end{array}
$$

The example above shows that there is a slight disadvantage when only these few combinators are used: the resulting expression is of exponential length with respect to the original expression. This can be improved by the following optimization rules (Curry and Feys, 1958):

$$
\begin{array}{lcl}
\mathrm{S}\ (\mathrm{K}\ E_1)\ (\mathrm{K}\ E_2) & = & \mathrm{K}\ (E_1\ E_2) \\
\mathrm{S}\ (\mathrm{K}\ E)\ \mathrm{I} & = & E
\end{array}
$$

These equalities can be proven as follows:

$$\underline{\mathrm{S}\ (\mathrm{K}\ E_1)\ (\mathrm{K}\ E_2)\ \mathit{anything}} =_{\mathrm{S}} (\underline{\mathrm{K}\ E_1\ \mathit{anything}})\ (\underline{\mathrm{K}\ E_2\ \mathit{anything}}) =_{\mathrm{K,K}} E_1\ E_2$$

$$\underline{\mathrm{K}\ (\mathrm{E}_1\ \mathrm{E}_2)\ \mathrm{anything}} \quad =_{\mathrm{K}} \quad \mathrm{E}_1\ \mathrm{E}_2$$

Since the combinators yield the same result for any argument, they are equal. Likewise

$$\underline{\mathrm{S}\ (\mathrm{K}\ \mathrm{E})\ \mathrm{I}\ \mathrm{anything}} \quad =_{\mathrm{S}} \quad (\underline{\mathrm{K}\ \mathrm{E}\ \mathrm{anything}})\ (\underline{\mathrm{I}\ \mathrm{anything}}) \quad =_{\mathrm{K,I}} \quad \mathrm{E}\ \mathrm{anything}$$

So the following rules can be added to the translation scheme that *Tr*ansforms an SKI-expression inside out into a more optimal one:

$$\begin{aligned}
&\mathrm{Tr}\ \llbracket\ \mathrm{E}_1\ \mathrm{E}_2\ \rrbracket &&= \mathrm{OptiSKI}\ \llbracket\ (\mathrm{Tr}\ \llbracket\ \mathrm{E}_1\ \rrbracket)\ (\mathrm{Tr}\ \llbracket\ \mathrm{E}_2\ \rrbracket)\ \rrbracket \\
&\mathrm{Tr}\ \llbracket\ \mathrm{c}\ \rrbracket &&= \mathrm{c}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{OptiSKI}\ \llbracket\ \mathrm{S}\ (\mathrm{K}\ \mathrm{E}_1)\ (\mathrm{K}\ \mathrm{E}_2)\ \rrbracket &&= \mathrm{K}\ (\mathrm{OptiSKI}\ \llbracket\ \mathrm{E}_1\ \mathrm{E}_2\ \rrbracket) \\
&\mathrm{OptiSKI}\ \llbracket\ \mathrm{S}\ (\mathrm{K}\ \mathrm{E})\ \mathrm{I}\ \rrbracket &&= \mathrm{E} \\
&\mathrm{OptiSKI}\ \llbracket\ \mathrm{E}\ \rrbracket &&= \mathrm{E}
\end{aligned}$$

These optimizations can be applied to our example expression

$$\begin{aligned}
&\underline{\mathrm{Tr}\ \llbracket\ \mathrm{S}\ (\mathrm{S}\ (\mathrm{K}\ \mathrm{S})(\mathrm{K}\ \mathrm{I}))\ (\mathrm{S}\ (\mathrm{K}\ \mathrm{K})\ \mathrm{I})\ \rrbracket} \\
\rightarrow\ \ &\mathrm{OptiSKI}\ \llbracket\ (\mathrm{Tr}\ \llbracket\ \mathrm{S}\ (\mathrm{S}\ (\mathrm{K}\ \mathrm{S})(\mathrm{K}\ \mathrm{I}))\ \rrbracket)\ (\underline{\mathrm{Tr}\ \llbracket\ \mathrm{S}\ (\mathrm{K}\ \mathrm{K})\ \mathrm{I}\ \rrbracket})\ \rrbracket \\
\rightarrow\ \ &\mathrm{OptiSKI}\ \llbracket\ (\mathrm{Tr}\ \llbracket\ \mathrm{S}\ (\mathrm{S}\ (\mathrm{K}\ \mathrm{S})(\mathrm{K}\ \mathrm{I}))\ \rrbracket)\ (\mathrm{OptiSKI}\ \llbracket\ (\mathrm{Tr}\ \llbracket\ \mathrm{S}\ (\mathrm{K}\ \mathrm{K})\ \rrbracket)(\underline{\mathrm{Tr}\ \llbracket\ \mathrm{I}\ \rrbracket})\ \rrbracket)\ \rrbracket \\
\rightarrow\ \ &\mathrm{OptiSKI}\ \llbracket\ (\mathrm{Tr}\ \llbracket\ \mathrm{S}\ (\mathrm{S}\ (\mathrm{K}\ \mathrm{S})(\mathrm{K}\ \mathrm{I}))\ \rrbracket)\ (\mathrm{OptiSKI}\ \llbracket\ (\underline{\mathrm{Tr}\ \llbracket\ \mathrm{S}\ (\mathrm{K}\ \mathrm{K})\ \rrbracket})\ \mathrm{I}\ \rrbracket)\ \rrbracket \\
\rightarrow^*\ &\mathrm{OptiSKI}\ \llbracket\ (\mathrm{Tr}\ \llbracket\ \mathrm{S}\ (\mathrm{S}\ (\mathrm{K}\ \mathrm{S})(\mathrm{K}\ \mathrm{I}))\ \rrbracket)\ (\underline{\mathrm{OptiSKI}\ \llbracket\ \mathrm{S}\ (\mathrm{K}\ \mathrm{K})\ \mathrm{I}\ \rrbracket})\ \rrbracket \\
\rightarrow\ \ &\underline{\mathrm{OptiSKI}\ \llbracket\ (\mathrm{Tr}\ \llbracket\ \mathrm{S}\ (\mathrm{S}\ (\mathrm{K}\ \mathrm{S})(\mathrm{K}\ \mathrm{I}))\ \rrbracket)\ \mathrm{K}\ \rrbracket} \\
\rightarrow^*\ &\mathrm{S}\ (\mathrm{K}\ (\mathrm{S}\ \mathrm{I}))\ \mathrm{K}
\end{aligned}$$

and indeed:

$$\begin{aligned}
&\underline{\mathrm{S}\ (\mathrm{K}\ (\mathrm{S}\ \mathrm{I}))\ \mathrm{K}\ \mathrm{X}}\ \mathrm{F} \quad =_{\mathrm{S}} \quad (\underline{\mathrm{K}\ (\mathrm{S}\ \mathrm{I})\ \mathrm{X}})\ (\mathrm{K}\ \mathrm{X})\ \mathrm{F} \quad =_{\mathrm{K}} \quad \underline{\mathrm{S}\ \mathrm{I}\ (\mathrm{K}\ \mathrm{X})\ \mathrm{F}} \\
&=_{\mathrm{S}} \quad (\underline{\mathrm{I}\ \mathrm{F}})\ (\underline{\mathrm{K}\ \mathrm{X}\ \mathrm{F}}) \quad =_{\mathrm{I,K}} \quad \mathrm{F}\ \mathrm{X}
\end{aligned}$$

Further optimizations involve the introduction of new combinators, e.g.:

$$\begin{aligned}
&\mathrm{B} \equiv \lambda f.\lambda g.\lambda x.f\ (g\ x) \\
&\mathrm{C} \equiv \lambda f.\lambda x.\lambda y.f\ y\ x
\end{aligned}$$

with corresponding new optimization rules (note that only two new rules are added):

$$\begin{aligned}
&\mathrm{OptiSKI}\ \llbracket\ \mathrm{S}\ (\mathrm{K}\ \mathrm{E}_1)\ (\mathrm{K}\ \mathrm{E}_2)\ \rrbracket &&= \mathrm{K}\ (\mathrm{OptiSKI}\ \llbracket\ \mathrm{E}_1\ \mathrm{E}_2\ \rrbracket) \\
&\mathrm{OptiSKI}\ \llbracket\ \mathrm{S}\ (\mathrm{K}\ \mathrm{E})\ \mathrm{I}\ \rrbracket &&= \mathrm{E} \\
&\mathrm{OptiSKI}\ \llbracket\ \mathrm{S}\ (\mathrm{K}\ \mathrm{E}_1)\ \mathrm{E}_2\ \rrbracket &&= \mathrm{B}\ \mathrm{E}_1\ \mathrm{E}_2 \\
&\mathrm{OptiSKI}\ \llbracket\ \mathrm{S}\ \mathrm{E}_1\ (\mathrm{K}\ \mathrm{E}_2)\ \rrbracket &&= \mathrm{C}\ \mathrm{E}_1\ \mathrm{E}_2 \\
&\mathrm{OptiSKI}\ \llbracket\ \mathrm{E}\ \rrbracket &&= \mathrm{E}
\end{aligned}$$

These optimizations are easily proven to be correct. With the help of these new combinators and optimization rules the complexity can be improved to $O(n^2)$, where n is the length of the original expression.

In this way INV can be expressed as B (S I) K. But on the other hand there is nothing wrong with introducing INV itself as an extra combinator. Many other combinators can be defined. It is an interesting line of research to find out which set is the best in a certain sense.

3.7 λ-calculus as a basis for functional languages

As stated before, the λ-calculus can be seen as a desugared functional language to which all existing functional languages can be translated. The same is possible for the other models of computation treated in this part of the book. An exhaustive transformation scheme from Miranda to a graph rewriting system is given in Chapter 9. The intention of this section is to illustrate how, in principle, the basic concepts of functional languages (see Chapter 1) can be expressed in the *pure* λ-calculus. This will give the reader an intuitive idea how it can be proven that in the pure λ-calculus all recursive functions can be expressed. The transformation of the more advanced language concepts can be found, for example, in Peyton Jones (1987).

3.7.1 Functions and applications

A simple Miranda function definition without patterns and guards is easy to translate to the λ-calculus since application is a basic operation in both languages. Furthermore, all Miranda functions are considered to be curried so they are actually functions with one argument.

$$
\begin{aligned}
&\text{MirToLam} [\![\, \text{f } a_1\ a_2 \ldots a_n = \text{expression} \,]\!] \\
&\qquad\qquad = \lambda a_1.\lambda a_2 \ldots \lambda a_n.\ (\text{MirToLam} [\![\, \text{expression} \,]\!]) \\
&\text{MirToLam} [\![\, \text{f a} \,]\!] = (\text{MirToLam} [\![\, \text{f} \,]\!])\ (\text{MirToLam} [\![\, \text{a} \,]\!])
\end{aligned}
$$

The notation means: a function definition in a Miranda program is translated into a λ-expression with a λ for each argument of the function definition. The body of the λ-expression is the translation of the body of the function. The resulting λ-expression is replaced for every application of f, possibly abbreviated by a symbolic name. Furthermore, any application of operands is translated directly to an application of the translated operands.

For instance,

```
twice f x    =  f (f x)
```

will be translated into

TWICE $\equiv$ λf.λx.f (f x)

Since all functions are anonymous, an application of the function twice in Miranda results in a textual substitution of the λ-expression TWICE in the corresponding λ-expression.

3.7.2 Booleans and the conditional

Boolean values can be translated as follows:

MirToLam ⟦ True ⟧ = λx.λy.x
MirToLam ⟦ False ⟧ = λx.λy.y

The translation of True and False is not a value, but a function that is able to select the first or second argument to which it is applied. This function corresponds to the conditional in Miranda. So the conditional itself disappears in the translation. Hence a conditional expression can be translated as follows:

MirToLam ⟦ cond boolexpr thenpart elsepart ⟧
= MirToLam ⟦ boolexpr ⟧ (MirToLam ⟦ thenpart ⟧) (MirToLam ⟦ elsepart ⟧)

because

$\underline{\text{TRUE}}$ THENPART ELSEPART $\equiv$ $\underline{(\lambda x.\lambda y.x)\ \text{THENPART}}$ ELSEPART
$\rightarrow_\beta$ $\underline{(\lambda y.\text{THENPART})\ \text{ELSEPART}}$ $\rightarrow_\beta$ THENPART

and

$\underline{\text{FALSE}}$ THENPART ELSEPART $\equiv$ $\underline{(\lambda x.\lambda y.y)\ \text{THENPART}}$ ELSEPART
$\rightarrow_\beta$ $\underline{(\lambda y.y)\ \text{ELSEPART}}$ $\rightarrow_\beta$ ELSEPART

The names in upper case characters are again abbreviations that represent λ-expressions that are the translation of corresponding Miranda expressions, e.g. TRUE $\equiv$ λx.λy.x, FALSE $\equiv$ λx.λy.y

3.7.3 Recursion

In the λ-calculus functions have no name. So recursion cannot be achieved by using recursive function calls, as in Miranda.

The factorial function in Miranda:

```
fac n    =  cond (n = 0) 1 (n * fac (n – 1))
```

cannot be translated into

FAC ≡ λn. (EQUAL n 0) 1 (TIMES n (FAC (MINUS n 1)))

In the recursive abbreviation above an infinite λ-expression is actually expressed. However, the λ-calculus deals with finite terms only. There is a trick to solve this problem. There exist special λ-expressions, called **fixed point combinators**, having the following property (Y is a fixed point combinator, f is any λ-expression):

Y f $=_\beta$ f (Y f)

Recursive functions now will get an additional parameter in which such a special duplicate of their own code is filled in, but this time this will happen only when the function is applied. Hence there is a kind of lazy duplication of code. In the following Y stands for any fixed point combinator.

A correct translation of the factorial function:

FAC ≡ Y F
F ≡ λf.λn.(EQUAL n 0) 1 (TIMES n (f (MINUS n 1)))

Now,

FAC 3 ≡ Y F 3
= F (Y F) 3 ≡ (λf.λn.(EQUAL n 0) 1 (TIMES n (f (MINUS n 1)))) (Y F) 3
≡ (λf.λn.(EQUAL n 0) 1 (TIMES n (f (MINUS n 1)))) FAC 3
$\rightarrow_\beta$ λn.(EQUAL n 0) 1 (TIMES n (FAC (MINUS n 1))) 3
$\rightarrow_\beta$ (EQUAL 3 0) 1 (TIMES 3 (FAC (MINUS 3 1)))
$\rightarrow_*$ FALSE 1 (TIMES 3 (FAC (MINUS 3 1)))
≡ (λx.λy.y) 1 (TIMES 3 (FAC (MINUS 3 1))
$\rightarrow_*$ TIMES 3 (FAC (MINUS 3 1))
≡ TIMES 3 (Y F (MINUS 3 1)), …

Each time Y F is applied, F is returned, and the original expression Y F is duplicated which guarantees that this kind of lazy code duplication can be repeated in the sequel. The property of fixed point combinators may seem very magical at first sight. Therefore an example of the definition of such a fixed point combinator is given below.

θ ≡ A A
A ≡ λx.λy.y (x x y)

The following reduction steps show that θ f $\rightarrow_*$ f (θ f), which implies that θ indeed has the fixed point property:

$$\underline{\theta}\ f \equiv \underline{A}\ A\ f \equiv \underline{(\lambda x.\lambda y.y\ (x\ x\ y))\ A}\ f$$
$$\rightarrow_\beta\ \underline{(\lambda y.y\ (A\ A\ y))\ f} \rightarrow_\beta\ f\ (\underline{A\ A}\ f) \equiv f\ (\theta\ f)$$

3.7.4 Representation of natural numbers

In the λ-calculus one can represent natural numbers and arithmetical functions. In this section every numeral n is represented by the nth Church numeral. The **Church numerals** c_0, c_1, c_2, ... are defined by ($f^0(x)$ stands for x, $f^{n+1}(x)$ stands for $f(f^n(x))$):

$$c_n \equiv \lambda f.\lambda x.f^n(x)$$

Hence,

$$0 \equiv c_0 \equiv \lambda f.\lambda x.x$$
$$1 \equiv c_1 \equiv \lambda f.\lambda x.f(x)$$
$$2 \equiv c_2 \equiv \lambda f.\lambda x.f(f(x))$$

Now it is possible to represent the arithmetical functions $+$, $*$ and $\exp$ in the λ-calculus. Rosser defines the following λ-expressions:

$$A_+ \equiv \lambda x.\lambda y.\lambda p.\lambda q.\ x\ p\ (y\ p\ q)$$
$$A_* \equiv \lambda x.\lambda y.\lambda p.\ x\ (y\ p)$$
$$A_{exp} \equiv \lambda x.\lambda y.\ y\ x$$

Now for all $n,m \in N$

(i) $A_+\ c_n\ c_m \rightarrow_*\ c_{n+m}$
(ii) $A_*\ c_n\ c_m \rightarrow_*\ c_{n*m}$
(iii) $A_{exp}\ c_n\ c_m \rightarrow_*\ c_{(n^m)}$, except for $m=0$

Below it is shown that A_+ yields the Church numeral representing the arithmetical sum of the two Church numerals it is applied to:

$$\underline{A_+}\ c_n\ c_m \equiv \underline{(\lambda x.\lambda y.\lambda p.\lambda q.\ x\ p\ (y\ p\ q))\ c_n}\ c_m$$
$$\rightarrow_\beta\ \underline{(\lambda y.\lambda p.\lambda q.\ c_n\ p\ (y\ p\ q))\ c_m}$$
$$\rightarrow_\beta\ (\lambda p.\lambda q.\ \underline{c_n}\ p\ (c_m\ p\ q)) \equiv (\lambda p.\lambda q.\underline{(\lambda f.\lambda x.f^n(x))\ p}\ (c_m\ p\ q))$$
$$\rightarrow_\beta\ (\lambda p.\lambda q.\underline{(\lambda x.p^n(x))\ (c_m\ p\ q)})$$
$$\rightarrow_\beta\ (\lambda p.\lambda q.p^n(\underline{c_m}\ p\ q) \equiv (\lambda p.\lambda q.p^n(\underline{(\lambda f.\lambda x.f^m(x))\ p}\ q)$$
$$\rightarrow_\beta\ (\lambda p.\lambda q.p^n(\underline{(\lambda x.p^m(x))\ q})$$
$$\rightarrow_\beta\ (\underline{\lambda p.\lambda q.p^n(p^m(q)})) =_\alpha (\lambda f.\lambda x.f^n(f^m(x)) \equiv c_{n+m}$$

So the arithmetic operations $+$, $*$ and $\exp$ on the numerals can be represented respectively by the λ-terms A_+, A_* and A_{exp} operating on the Church numerals. It will be clear that it is much more efficient to actually use the machine representation of numerals.

3.7.5 Lists

The translation of lists is not as intuitively simple as the translation of simple function definitions. The trick is to translate every constructor with its arguments to a λ-term representing a function that given a projection as an argument produces the desired component. So the constructors are translated as follows (implicitly defining CONS and NIL):

MirToLam ⟦ x:xs ⟧ = λz.z (MirToLam ⟦ x ⟧) (MirToLam ⟦ xs ⟧)
MirToLam ⟦ [ ] ⟧ = λx.TRUE

Functions like head and tail are translated into λ-terms that given a list as an argument apply the list to the appropriate projection (implicitly defining HD and TL):

MirToLam ⟦ hd ⟧ = λx.x TRUE
MirToLam ⟦ tl ⟧ = λx.x FALSE

such that hd (x:xs) is translated into HD (CONS X XS) ≡ (λx.x TRUE) (λz.z X XS). In an analogous way, tl (x:xs) is translated into: TL (CONS X XS) ≡ (λx.x FALSE) (λz.z X XS). Below it is shown that the translations are correct:

$\underline{\text{HD}}$ ($\underline{\text{CONS}}$ X XS)		$\underline{\text{TL}}$ ($\underline{\text{CONS}}$ X XS)	
$\equiv$	$\underline{(\lambda\text{x.x TRUE})}$ $\underline{(\lambda\text{z.z X XS})}$	$\equiv$	$\underline{(\lambda\text{x.x FALSE})}$ $\underline{(\lambda\text{z.z X XS})}$
$\rightarrow_\beta$	$\underline{(\lambda\text{z.z X XS) TRUE}}$	$\rightarrow_\beta$	$\underline{(\lambda\text{z.z X XS) FALSE}}$
$\rightarrow_\beta$	$\underline{\text{TRUE}}$ X XS	$\rightarrow_\beta$	$\underline{\text{FALSE}}$ X XS
$\equiv$	(λx.λy.x) X XS	$\equiv$	(λx.λy.y) X XS
$\rightarrow_*$	X	$\rightarrow_*$	XS

3.7.6 Patterns

One can simulate pattern matching by using conditionals in combination with a test for equality.

Modelling pattern matching by conditionals:

```
fac 0   =  1
fac n   =  * n (fac (– n 1))
```

can be written as (in which the conditional is created in λ-calculus as above):

```
fac n   =  cond (= n 0) 1 (* n (fac (– n 1)))
```

A test for syntactic equality is however not easy to achieve in pure λ-calculus. It can be realized in two ways.

The first way is choose a suitable representation (λ-term) such that each constant one would like to compare can be examined via an application of a specially designed test function. The comparison is done by applying the test function to the object and reducing the application to normal form. If the normal form is the λ-term that corresponds to TRUE, then they are equal; if it is the λ-term that corresponds to FALSE they are not. The disadvantage of this method is that it is sometimes hard to come up with an appropriate representation.

For instance, define

ZERO	≡	CONS TRUE NIL
ONE	≡	CONS FALSE ZERO
TWO	≡	CONS FALSE ONE

Now a test on the equality on ZERO can be performed by taking the head of the representation of the numeral. A test on ONE can be performed by taking the head of the tail, and so on.

The second way is to add additional power to the model by the introduction of a special δ-rule (Church's δ) that is defined on normal forms. This δ-rule returns TRUE or FALSE depending on the syntactic equality of its arguments. The strategy has to ensure that the arguments are reduced to normal form before the δ-rule is applied.

3.8 λ-calculus as a basis for implementations

A **reducer** is a piece of hardware and/or software that evaluates an expression by performing reductions. Although the λ-calculus appears to be simple, an implementation of a λ-reducer is not straightforward, particularly if efficiency is important. This section discusses some of the problems that have to be solved.

First of all, for efficiency reasons δ-rules will have to be used for pattern matching and for dealing with basic values.

Then, since our prime interest is a reducer that tries to reduce to normal form, one of the problems is caused by the α-conversion. Implementation of α-conversion has a considerable impact on efficiency. Below the relevant issues related with α-conversion will be discussed.

But there are more implementation problems. During the reduction process the reducer has to look up the actual argument to substitute for a particular variable. For this substitution process a rather complicated administration is needed. Only via the proper λ that binds the variable can the corresponding argument be found. The consequence is that in the actual implementation either all arguments accessible in a function body are passed to a function or a pointer structure has to be set up that makes it possible to switch back to the appropriate environment in which an argument can be evaluated.

3.8.1 α-conversion

It is clear that performing α-conversions after every reduction will never yield an efficient implementation. De Bruijn (1972) and Berkling and Fehr (1982) have developed solutions to the problem of name conflicts. Both their solutions are described here. They are basically equivalent to each other. The general idea is that in every variable name it is encoded via a counter to which λ the variable is bound.

Berkling and Fehr introduce the *λ-bar operator*: '#'. The number of **λ-bar operators** in front of a variable indicates the number of surrounding λs, which bind variables with the same name, to skip. A λ **surrounds** a variable in a λ-term if the λ is above the variable in the tree representation of that λ-term. In that case the variable is in the body of the function corresponding to that λ. The following example illustrates the function of this operator.

λ-bar operator:

```
λx.x                  ||  x is bound.

λx.#x                 ||  the # indicates to skip the first λ-occurrence
                      ||  that binds x. So x is free in this expression.

λx.λx.#x              ||  x is bound by the indicated λ, because #
↑                     ||  indicates to ignore the first surrounding λ.

λx.λx.λx.##x          ||  x is bound by the indicated λ.
↑

λx.λy.λx.#x           ||  x is bound by the indicated λ.
↑

λx. λx. #x (λx. ##x)  ||  both xs are bound by the same indicated λ.
↑

λx.(λx. #x) (λx. #x)  ||  both xs are bound by the same indicated λ.
↑                     ||  note that the second λ does not surround the second #x.
```

The following problem arises: doing a β-reduction implies that the number of λ-bar operators in the expression has to be adjusted in the body of the function as well as in the argument.

Examples of the adjustment of the number of λ-bar operators:

```
(λx.(λy.x y))   y   →β   λy.#y y
                ↑
```

It is not λy.y y, because the indicated y is a free variable of the expression and β-reduction should not change the meaning of variables.

λy.($\underline{(\lambda\text{y.\#y y) z}}$) $\rightarrow_\beta$ λy.y z

Not λy.#y z, because in λy.#y z, #y is a free variable whereas #y was bound in the expression before the β-reduction took place.

The solution of De Bruijn uses *nameless dummies*: instead of named variables. A **nameless dummy** is a number that replaces a named variable. The nameless dummy indicates by which λ the replaced variable is bound. The λs are counted starting from the nameless dummy going to the top of the tree representation of the term. For free occurrences of variables nameless dummies are replaced as if extra λs binding them were added in front of the term. In this way effectively also the names of the free variables are modelled by the nameless dummies. Note that De Bruijn's $\underline{n}$ means that the *n*th surrounding λ binds, while Berkling and Fehr's $\#^{n}$x means to skip *n* surrounding λs that bind x.

Nameless dummies:

λ. $\underline{1}$ || $\underline{1}$ is a dummy for a variable bound by the indicated λ.
↑

λ. $\underline{2}$ || $\underline{2}$ is a dummy for a variable that is free.

λ. λ. $\underline{2}$ (λ. $\underline{3}$) || $\underline{2}$ and $\underline{3}$ indicate the same variable, bound by indicated λ.
↑

λ. (λ. $\underline{2}$) (λ. $\underline{2}$) || $\underline{2}$ and $\underline{2}$ indicate the same variable, bound by indicated λ.
↑

As with the # operator, the numbers of the nameless dummies have to be adjusted when doing a β-reduction.

Adjusting the numbers of De Bruijn:

(λ. (λ. $\underline{2}$ $\underline{1}$)) $\underline{1}$ $\rightarrow_\beta$ λ. $\underline{2}$ $\underline{1}$ || not λ. $\underline{1}$ $\underline{1}$
λ.((λ. $\underline{2}$ $\underline{1}$) $\underline{2}$) $\rightarrow_\beta$ λ. $\underline{1}$ $\underline{2}$ || not λ. $\underline{2}$ $\underline{2}$

3.8.2 Avoiding α-conversion

It has been shown that, in general, α-conversion cannot be avoided during β-reduction. However, if redexes are reduced in a special order and the final result does not have to be in normal form, α-conversion can be avoided. This can be used in a reducer that reduces a λ-expression which is the result of a translation of a Miranda program.

Suppose, when reducing in normal order, the leftmost-outermost redex is rewritten. Starting from Miranda the initial expression is a closed term, i.e. it does not contain free variables. So if the corresponding argument is substituted in the function body it is not possible that free variables in the argument get bound. Name conflicts cannot arise if the leftmost-outermost redex is repeatedly reduced in this way and the reduction process is stopped as soon as the result is a function. As long as reduction inside the body of a function (under an abstraction) does not occur, no α-conversion is needed.

> Example: if the underlined redex inside the function body is reduced, name conflicts arise:
>
> $(\lambda y.\underline{(\lambda x.\lambda y.x\ y)\ y})\ (\lambda x.\lambda y.x\ y)$
>
> However, reducing in normal order will not give any problems:
>
> $\underline{(\lambda y.(\lambda x.\lambda y.x\ y)\ y)\ (\lambda x.\lambda y.x\ y)} \to_\beta \underline{(\lambda x.\lambda y.x\ y)\ (\lambda x.\lambda y.x\ y)} \to_\beta \lambda y.(\lambda x.\lambda y.x\ y)\ y$
>
> Now conversion problems would arise if reduction would continue inside the body of the function.

So the evaluation has to be stopped as soon as the result is known to be a function. Starting from Miranda this is all right. In general, Miranda programs do not yield a function but a value of a basic type, e.g. a string. In the case that a function is yielded, the message <function> is printed and no further reduction to normal form takes place. So it is possible to proceed with reduction in the way that is described above. Below the formal foundation for this approach will be given.

An expression E in the pure λ-calculus is in **head normal form** if it is of the form $\lambda x_1.\lambda x_2. \dots \lambda x_n.x\ E_1\ E_2 \dots E_m$ with $x_1, x_2, \dots x_n, x$ (not necessarily different) variables and $n, m \geq 0$. In that case x is the **head variable** of the expression E. If E is of the form $\lambda x_1.\lambda x_2. \dots \lambda x_n. (\lambda x.E_0)\ E_1\ E_2 \dots E_m$ with $n \geq 0$, $m \geq 1$, then $(\lambda x.E_0)\ E_1$ is called the **head redex** of E. An expression E in the pure λ-calculus is in **weak head normal form** if it is an expression of either the form $\lambda x.E$ or the form $x\ E_1\ E_2 \dots E_m$ with $m \geq 0$. A head normal form is always a weak head normal form.

A head normal form does not contain a head redex. A λ-abstraction in weak head normal form may still contain a head redex in its body. So a (weak) head normal form does not have to be a normal form.

The notion of reduction in which reduction never takes place under an abstraction is called **weak reduction** (conversely, reduction under an abstraction is sometimes called **strong reduction**). When only weak reduction is performed, no α-conversion is needed if initially the free variables are chosen to be distinct from the bound variables. Since weak reduction cannot reduce under a λ-abstraction it generally cannot yield a head normal form but only a weak head normal form.

Abramsky's work on lazy λ-calculus formalizes the notion of lazy (leftmost) reduction to weak head normal form (Abramsky, 1990; Ong, 1988). Lazy reduction is a form of weak reduction, so in lazy λ-calculus no α-conversion is necessary.

3.8.3 Sharing of expressions to avoid duplication of work

Consider the following expression:

```
(λx.+ x x) heavy-computation
```

Assume that the evaluation of the argument involves many reductions. The following reduction steps are performed if the expression is evaluated in normal order:

$\underline{\text{(λx.+ x x) heavy-computation}}$
$\rightarrow_\beta$ + $\underline{\text{heavy-computation}}$ heavy-computation
$\rightarrow_*$ + easy-number $\underline{\text{heavy-computation}}$
$\rightarrow_*$ $\underline{\text{+ easy-number easy-number}}$
$\rightarrow_{\delta+}$ twice-the-easy-number

The argument is substituted for each occurrence of the variable in the function body. But if such an argument is a complicated expression requiring many reductions then each additional substitution implies duplication of work. This is of course a terrible waste because the result of the computation will be the same for each occurrence. In the example above this problem can be solved by reducing the expression in applicative order.

(λx.+ x x) $\underline{\text{heavy-computation}}$
$\rightarrow_*$ $\underline{\text{(λx.+ x x) easy-number}}$
$\rightarrow_\beta$ $\underline{\text{+ easy-number easy-number}}$
$\rightarrow_{\delta+}$ twice-the-easy-number

In general, it is much more efficient to reduce in applicative order instead of in normal order. But unfortunately, when reduction is performed in applicative order, termination is no longer guaranteed (see Section 3.3).

There are several solutions to retain the efficiency. One solution is to analyse the arguments of a function to determine whether or not they can be reduced in applicative order instead of in normal order. This is allowed if the termination behaviour compared with normal order reduction is not changed (see Chapter 7). If such arguments can be found a mixture of normal order and applicative order evaluation is obtained. The analysis to determine whether or not arguments can be evaluated in applicative order is known as *strictness analysis* (see Chapter 7).

The problem of the duplication of work can be partly solved by choosing a different representation for a λ-expression. A computer scientist would use pointers, i.e. use a *graph* representation instead of using *strings* or *trees*. However, this changes the reduction behaviour since several redexes are now contracted at once. So it is not sufficient just to use a different representation. The pure λ-calculus has to be extended with some notion of sharing of terms. The first proposal for using λ-calculus **graph reduction** instead of the ordinary **string reduction** is due to Wadsworth (1971). Graphically this sharing can be expressed as in Figure 3.6.

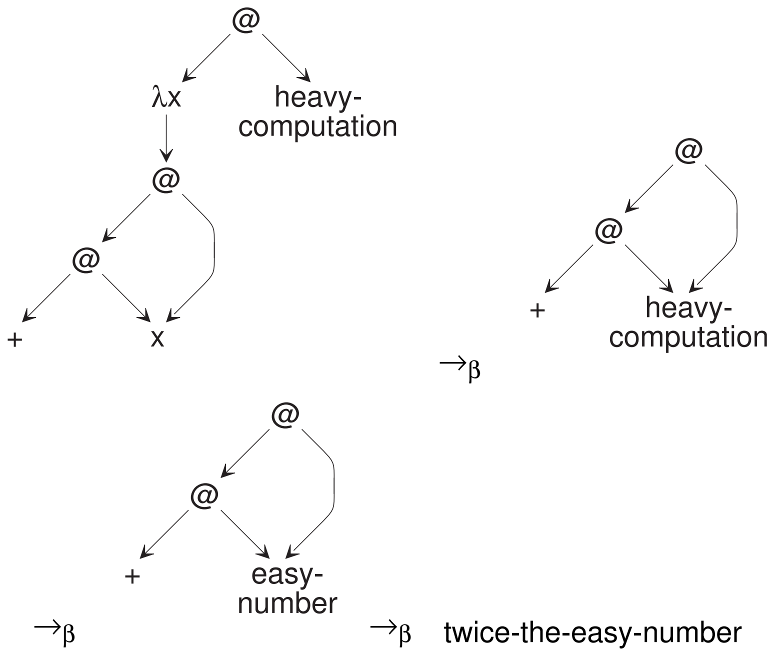

Figure 3.6 Sharing of λ-expressions to avoid duplication of work.

The following linear notation is chosen to indicate shared λ-expressions:

$\underline{(\lambda x.+\ x\ x)\ S}$, where S ≡ heavy-computation
$\rightarrow_\beta$ + S S, where S ≡ $\underline{\text{heavy-computation}}$
$\rightarrow_*$ $\underline{+\ S\ S}$, where S ≡ easy-number
$\rightarrow_{\delta+}$ twice-the-easy-number

Unfortunately, the sharing has to be broken if the shared expression is a function that is applied to different arguments (Figure 3.7).

For instance, consider the following expression:

S 3 (S 4), where S ≡ λx.+ x (Fac 1000)

If the argument 3 were to be substituted in the function S, this shared function would be corrupted. Consequently, the shared expression S has to be partially **deshared** (i.e. a copy is made of the shared structure) as shown in Figure 3.7.

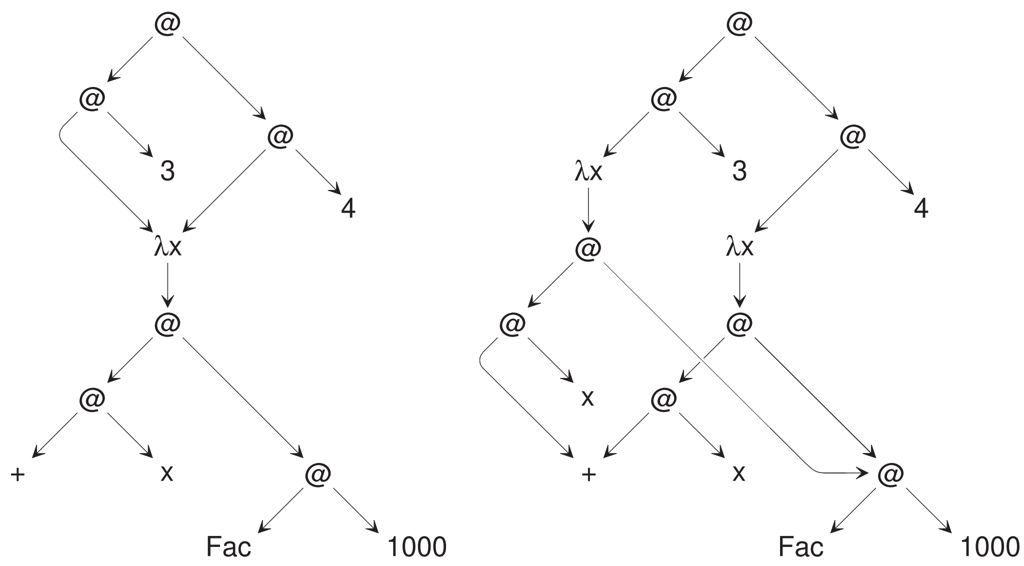

Figure 3.7 Desharing needed when the shared expression is a function.

To keep things sound, that part of the shared function has to be copied that is affected by a possible substitution of the variable. This process of partly desharing is also called **peeling off a copy** (Wadsworth, 1971). It increases the complexity of an implementation.

The example of Figure 3.8, due to J.J. Lévy, shows that, because of the forced copy action, duplication of work is sometimes hard to avoid. When the leftmost redex is reduced, there seems to be no way to deshare (parts of) the λf expression without introducing a copy of a λx-redex. This is due to the fact that when a copy is peeled off, a copy has to be made of all applications above the variable to be substituted up to the λ that binds it. So the application f 1 has to be copied but also the two applications above it including the λx redex.

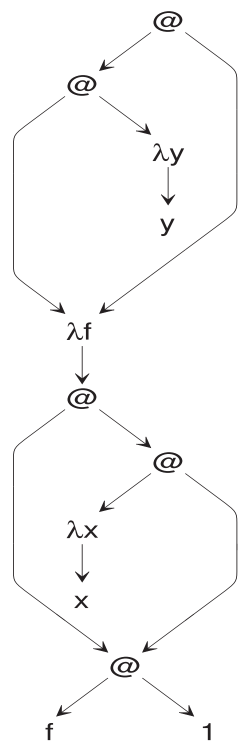

Figure 3.8 J.J. Lévy example.

The problem of avoiding duplication of work is called the optimality problem (Lévy, 1980). Recently, in Lamping (1990) an algorithm is given that solves the optimality problem. However, the algorithm is very complex. A lot of administration is needed to always avoid duplication of work. In practice, for most cases it may be more efficient to duplicate the work instead.

3.8.4 Implementations based on combinatory logic

The use of combinators has been studied very extensively in the context of implementing functional languages. Combinators do not suffer from the problems with copying shared subexpressions that are treated in the previous section. Expressions of these combinators do not contain variables. So if combinators are implemented as basic primitives, no administration for variables is needed at all.

David Turner was the first to introduce an implementation of a functional language using combinators as an intermediate language (Turner, 1979a). He used about 15 combinators including S, K, I, B and C named above and of course some δ-rules. This paper led to a lot of research on the question of which set of combinators was best for implementation purposes (e.g. super-combinators (Hughes, 1982) and categorical combinators (Curien, 1986)) and even to research on machine architectures based on such sets of combinators.

Striving for ultimate efficiency recent implementations tend to employ sharing not with a fixed set of combinators but with a dedicated set of combinator-like definitions (rewrite rules) that are deduced from the program. The next chapters cover the concepts of (graph) rewrite rules of which combinators are treated as a special case.

3.8.5 Translating evaluation orders

In the translation schemes given in Section 3.7 the evaluation order was not taken into account. Actually, when λ-terms were reduced in the examples, normal order evaluation was used. This is not remarkable since Miranda is a lazy language. *Lazy evaluation* corresponds with *normal order reduction* in the λ-calculus, whereas *eager evaluation* corresponds to *applicative order reduction*. Lazy languages can handle infinite lists as long as the evaluation of the last element is not being asked for. This corresponds with normal order reduction that will find the normal form if it exists.

Languages can also be characterized by the way parameters are passed, the **calling mechanisms**. *Call-by-value* implies *applicative order reduction*, *call-by-name* corresponds with *normal order reduction*; and *call-by-need* corresponds to *normal order reduction* in a context in which the arguments are *shared* such that they are evaluated only once.

Summary

- The *λ-calculus* can be considered to be the computational basis of all functional languages.

- In the λ-calculus *anonymous functions* are defined.
- *Reduction rules* translate λ-expressions into equivalent ones.
- The most important reduction rule is β-*reduction* that defines how a function applied to an argument is evaluated by applying *textual substitution*.
- With α-*conversion* bound variables can be renamed.
- In general α-conversion is needed when doing β-reduction.
- α-conversion can be avoided when only *weak reduction* is done.
- *Lazy evaluation* in functional languages corresponds to *normal order reduction* in the λ-calculus.
- *Eager evaluation* in functional languages corresponds to *applicative order reduction* in the λ-calculus.
- Theoretical and practical properties of functional languages can be derived directly from the λ-calculus:
  - all effectively computable functions can be defined;
  - normal forms are unique;
  - not every expression has a normal form;
  - lazy evaluation is normalizing;
  - with lazy evaluation work can be duplicated while eager evaluation avoids duplication of work;
  - eager evaluation is not normalizing.
- The λ-calculus is not very well suited as a basis for efficient implementation for the following reasons:
  - full sharing of expressions is difficult to maintain;
  - testing on syntactic equality is not possible in the pure λ-calculus. The extension of the calculus with Church's δ-rule is rather artificial. For reasoning about implementations of functional languages it seems to be a better approach to start with a computational model in which syntactical tests (pattern matching) are inherently present (see the next chapter).

EXERCISES

3.1 Make tree-representations of the following λ-expressions:

(a) `(λx.λy. x+y) ((λx. y) 1) ((λy.λz. x z) ((λx. 2*x) z) 5)`

(b) `FAC y`

where `FAC` is the abbreviation of the λ-expression for the factorial function given in Section 3.7.3.

Mark the free variables and the leftmost-outermost, leftmost-innermost, rightmost-outermost and rightmost-innermost redexes.

3.2 Consider the λ-expression λx.x ((λx.λy.x y) y) x. Give its tree representation. Reduce the expression normal order to weak head normal form and to normal form.

3.3 Reduce the following expressions in normal order, once using the λ-bar operator and once using the numbers of De Bruin.

(a) λx.((λx.λy.y z x) y x)
(b)* (λf.λx. f(f x)) x) ((λf.λx. f(f x)) x)

3.4 Show that A_{exp} c_n c_m ->* $c_{(n^m)}$, except for m = 0, where c_n and c_m are the Church numerals and A_{exp} = λx.λy.y x.

3.5 Define λ-expressions AND, OR and NOT with standard semantics.

3.6 Represent the following expression by a graph, and so build a tree but share (sub)expressions where possible.

(+) (F (F 3)) (F 300)   where F = λx. * (Fib 10) x

Fib represents the Fibonacci function. Show with the help of the graph what happens when the following redexes will be β-reduced:

(a) (+) (F (<u>F 3</u>)) (F 300)
(b) (+) (<u>F (G)</u>) (F 300)   where G stands for result of previous reduction
(c) (+) (H) (<u>F 300</u>)   where H stands for result of previous reduction

3.7* Write a reducer for λ-expressions in Miranda. This function should reduce a λ-expression to normal form using normal order reduction. The program should accept a list of characters as input and deliver the reduced λ-expression again as a list of characters. Take as internal data structures to represent λ-expressions:

```
lcalc   ::=  AP lcalc lcalc | ABS var lcalc | VAR var | CONST const
var     = =  [ char ]
const   = =  [ char ]
```

Furthermore, define a *parser* function to parse the input string to this data structure lcalc, a *reducer* (of type reduce:: lcalc -> lcalc) and a *display function* that converts the data structure back to a list of chars. Make sure that your reducer does α-conversion when necessary (this is probably the most difficult part of this exercise).