Chapter 4
Term rewriting systems

Term rewriting systems (TRSs) form an important computational paradigm for which applications are found in several fields, including formal software specification, analysis of algebraic specifications, design and implementation of functional languages, computing by means of equations and mechanizing deduction systems. Surveys of theory of TRSs are given in Huet and Oppen (1980) and Klop (1992).

A TRS consists of a collection of rewrite rules. Rewrite rules have many similarities with function definitions in a functional language but it is only possible to define rules on one global level. The rules in a TRS are used to rewrite a given term in the same way as function definitions in a functional language are used to evaluate a given initial expression. In TRSs tests on syntactic equality (using pattern matching) form a key concept in the semantics of reduction. The left-hand side of a rewrite rule consists of a pattern that determines whether or not the rule can be applied to rewrite a (sub)term.

One can regard TRSs as the extension of combinatory logic with pattern matching. TRSs are also related to the λ-calculus. The descriptive power of TRSs is greater than of these systems, e.g. non-determinism can be expressed in a TRS but not in λ-calculus nor in combinatory logic. TRSs are a subset of *Graph rewriting systems* (GRSs) which will be explained in the next chapter. A good understanding of TRSs is a prerequisite for a good understanding of GRSs.

An important difference between TRSs and functional languages is that in the basic semantics of TRSs there is no ordering in the rewrite rules (all rules have equal priority) and furthermore no strategy is predefined that prescribes in which order redexes in the term are rewritten. So in a certain sense one can say that TRSs are 'more declarative' than functional languages. They just contain a collection of rules that can be applied to a given term. No reduction order is specified explicitly that determines how a given term has to be evaluated.

The bad news is that in contrast with the λ-calculus, there does not exist a decidable normalizing strategy for a general TRS (see Section 4.5). Since the basic semantics of TRSs is quite simple (Sections 4.1, 4.2 and 4.3), the emphasis of this chapter lies in a quest for an efficiently implementable, intuitively clear, reduction strategy that is normalizing for a large class of TRSs. One of the problems encountered in this quest is caused by the possibility of defining ambiguous rules (partly due to the fact that all rewrite rules have equal priority in the basic semantics). On one hand this has the advantage that TRSs allow the expression of non-deterministic computations. But on the other hand this has the disadvantage that (of course) the uniqueness of normal forms is no longer guaranteed (Section 4.4). With certain restrictions imposed on TRSs (*orthogonal* TRSs, see Sections 4.4 and 4.6), uniqueness of normal forms can be guaranteed. Also, there exists a normalizing strategy (*parallel outermost*) for such orthogonal systems. Unfortunately, this normalizing strategy is certainly not efficiently implementable. Efficiently implementable normalizing strategies are possible, but they will only work for certain subclasses of orthogonal TRSs. Furthermore, it is not always intuitively clear how these strategies proceed nor for which subclass they can or cannot be applied (see Section 4.6).

It appeals to the intuition to take priority of rules into account. The basic problems encountered in a TRS with standard semantics also appear in TRSs extended with a functional language-like priority in their rules (Section 4.7). Forcing evaluation of arguments to tell whether or not they match a corresponding pattern in a rule definition now seems to be the only sensible thing to do. The disadvantage of forcing evaluation is that infinite computations can occur, even if a normal form exists according to the priority semantics. As a special case of this general approach of forcing evaluation the *functional strategy* is presented. When reduced with the functional strategy *priority rewrite systems* (PRSs) capture priority of rules similar to a functional language.

A functional language is easily translated to various subclasses of TRSs (Section 4.8). Implementations of functional languages are usually more close to term rewriting systems than to the λ-calculus. An implementation based on TRSs causes fewer problems than an implementation based on the λ-calculus. Efficient implementations of functional language-like PRSs can be obtained (Section 4.9). If graphs are used instead of terms even state-of-the-art efficiency is possible (see Part 4).

4.1 TRSs

Term rewriting systems consist of a collection of *rewrite rules*. These rewrite rules specify how a given *term* can be translated (*reduced*, *rewritten*) to another term.

4.1.1 Terms

A collection of **terms** over a set of *constant symbols* and a set of *variables* is defined by the following syntactic definition (the curly braces specify *zero or more* occurrences of the enclosed objects):

```
Term    =  Variable    |   Constant   |   '(' Constant {Term} ')' ;
```

Variables always begin with a lower case character; constant symbols with an upper case character or some special character (a digit, an asterisk etc.). The outermost brackets of a term are usually omitted. The (zero or more) terms following a constant are called the **arguments** of the constant. The fixed **arity** of an occurrence of a particular constant is equal to its number of arguments. A term that does not contain any variables is called a **closed term**, otherwise it is an **open term.**

Examples of terms:

x	Reverse (Cons 2 Nil)
1000	Select 3 (Array 1 2 3 4 5 6)
Fac x	Sqrt (– (* b b) (* 4 (* a c)))
Fac 1000	Ap (Ap x z) (Ap y z)

A non-example:

x z (y z)

A term τ' is a **subterm** of a term τ iff (i.e. if and only if) τ' is contained in τ and τ' is a term in the syntactic construction of τ. A subterm τ' of τ is a **proper subterm** if τ' and τ are not the same. Subterms do differ from subexpressions in Miranda and in the λ-calculus. This is due to the absence of implicit application. Implicit application would allow any two expressions to be put together by juxtaposition to form a new expression. With explicit application one has to specify a function that defines application (see Section 4.1.4).

Subterms of the term Ap (Ap x K) K are (see Figure 4.1):

Ap (Ap x K) K	x
Ap x K	K (occurs twice)

The following are *not* subterms of the term Ap (Ap x K) K:

Ap	x K
Ap (Ap x K)	(Ap x K) K
Ap x	

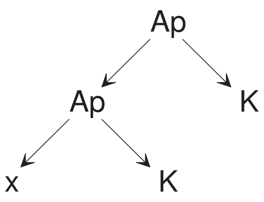

Figure 4.1 Tree representation (see Section 4.1.2) of Ap (Ap x K) K.

4.1.2 Representing terms as trees

A term can be represented as a tree (Figure 4.2).

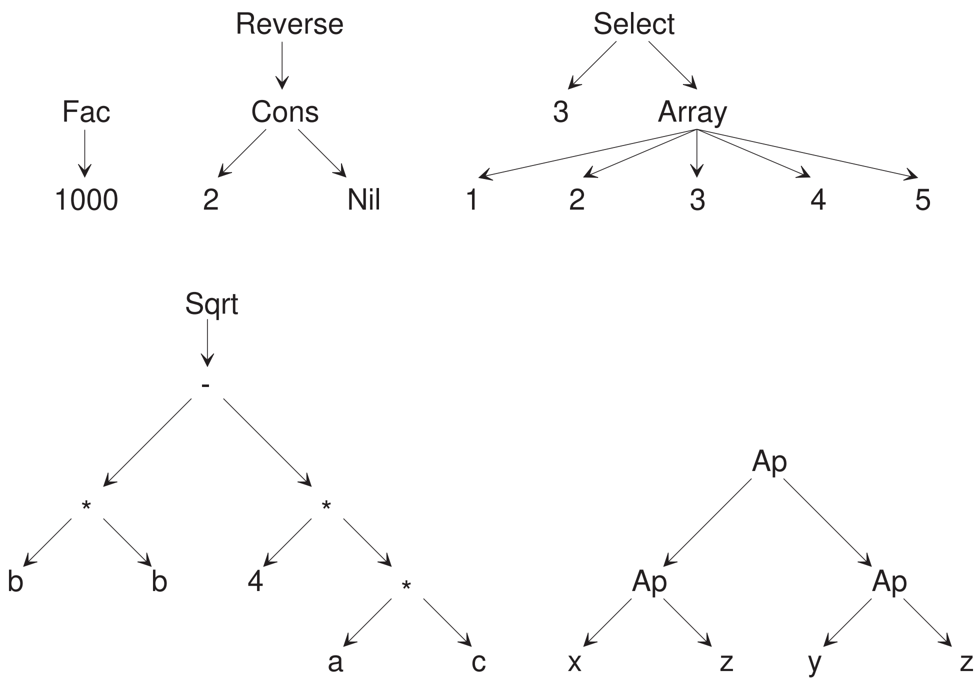

Figure 4.2 Tree representation of the term Fac 1000, Reverse (Cons 2 Nil), Select 3 (Array 1 2 3 4 5 6), Sqrt (– (* b b) (* 4 (* a c))), Ap (Ap x z) (Ap y z).

This tree representation makes the structure of a term clearer in most cases. A constant symbol is represented by a node labelled with that constant. A directed arc is drawn to each argument of a constant. Hence, the arity of a constant determines the number of outgoing arcs from the corresponding node. A subtree of a tree consists of a node with all nodes that are reachable (following the arcs) from that node. Any subtree represents a subterm.

4.1.3 Term rewriting systems

A **term rewriting system** (TRS) is a collection of **rewrite rules** $\alpha_i \to \beta_i$ over sets of constants and variables, where α_i, β_i are terms, such that all variables in β_i also occur in α_i and none of the α_i is a term consisting of a single variable.

Example of a TRS:

```
Fac 0            →   1
Fac n            →   * n (Fac (– n 1))

Cond True x y    →   x
Cond False x y   →   y
```

Non examples:

```
x            →   1

Illegal x    →   y
```

Syntax of TRSs

```
TRS                = {TRS-Rule} ;
TRS-Rule           = Left-Hand-Side '→' Right-Hand-Side ;
Left-Hand-Side     = Constant {Term} ;
Right-Hand-Side    = Term ;
```

The constants used in a TRS are divided in two classes. All constant symbols in a TRS that appear as the leftmost constant on the left-hand side of a rule are called **(recursive) functions**. The corresponding rule is then called **a rule for** the function and the corresponding occurrence of the symbol is called a **defining occurrence** of the symbol. All other constants are called **constructors**. The term on the left-hand side is called a **pattern**. A subterm of a left-hand side is called a **(sub)pattern**.

4.1.4 Notation used in this book

The notation for TRSs used in this textbook is a variant of the *functional style*. We use an alternative bracketing, resulting in a LISP-like notation. In the literature this notation is very unusual. Either an *applicative style* (like in Miranda, with implicit application functions) or a *functional style* is used. In a **functional style** application is written down *ex-*

plicitly as a function (say the function Ap which, of course, stands for Application). However, traditionally, in a functional style the arguments of functions and constructors are placed between brackets and separated by commas.

> Example of the notation used in this book (the combinatory logic example; see also Section 3.6.1):
>
> Ap (Ap (Ap S x) y) z → Ap (Ap x z) (Ap y z)
> Ap (Ap K x) y → x
>
> The SK example in the traditional functional notation (compare this with our notation in the example above):
>
> Ap (Ap (Ap (S, x), y), z) → Ap (Ap (x, z), Ap (y, z))
> Ap (Ap (K, x), y) → x

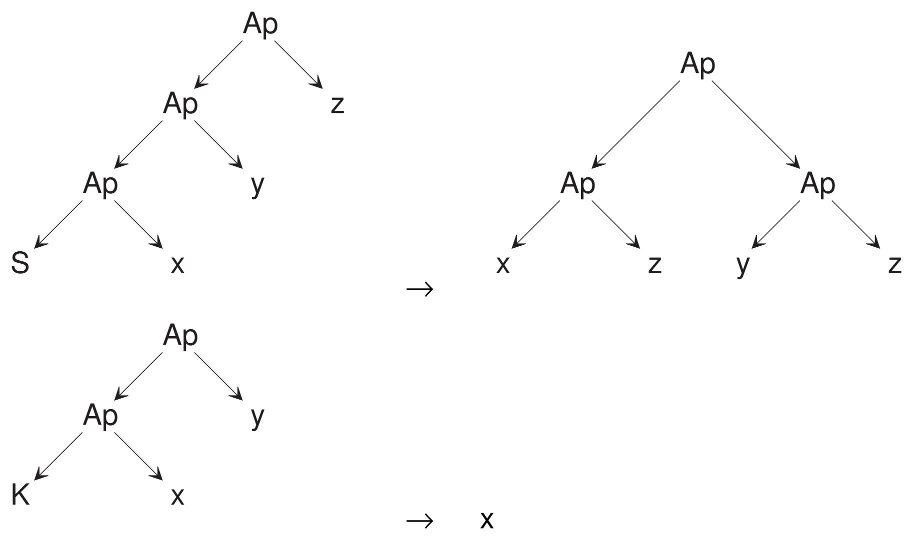

Figure 4.3 Tree representation of SK rules in (alternative) functional style.

When the rewrite rules of a TRS are represented graphically (see Figure 4.3) using the tree notation for terms, it is immediately clear that the difference in both notations is a matter of syntax only.

In a TRS using an **applicative style** function applications are *implicit*. The applicative style is commonly used when application is the only actual function that is being used and all other symbols have arity zero (Figure 4.4). When application is the only function, it can be left out in the notation and juxtaposition can stand for application.

In this textbook we generally want to consider also other functions than apply (Ap) but we do not like to write down too many brackets. Therefore we have chosen a functional style with a LISP-like bracketing. We will use this notation throughout this textbook.

The SK example in the applicative notation:

S x y z	$\rightarrow$	x z (y z)
K x y	$\rightarrow$	x

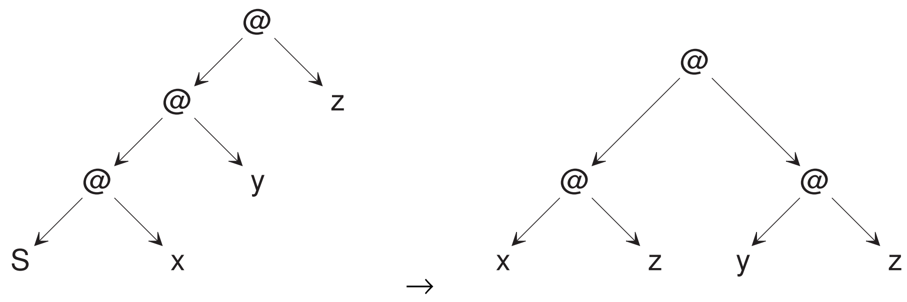

Figure 4.4 Tree representation of a TRS with implicit application functions (indicated by @ labels).

4.2 Rewriting with a TRS

A TRS can be used to rewrite a given *closed* term τ, which consists of constants only. This closed term can be compared with the initial expression to be evaluated in Miranda.

4.2.1 Redexes

A (sub)term is a redex iff there is a left-hand side of a rule (a pattern) that matches this term. A left-hand side matches a (sub)term if this term is syntactically equal with respect to all constants in the pattern.

More formally: a **redex** of a closed term τ is a subterm τ' that is syntactically equal to an *instance* of a left-hand side α_i of a rule of the TRS. An **instance** of a term α_i is a term $\sigma(\alpha_i)$ where σ is a substitution of closed terms for all variables in α_i. One also says: τ' **matches** α_i. By σ every variable of a left-hand side is bound to a subterm of the redex.

Example of pattern matching; consider again:

Ap (Ap (Ap S x) y) z	$\rightarrow$	Ap (Ap x z) (Ap y z)
Ap (Ap K x) y	$\rightarrow$	x

and the term

Ap (Ap K K) (Ap S K)

This term matches the pattern Ap (Ap K x) y. The corresponding substitution is σ(x) = K, σ(y) = Ap S K, yielding σ(Ap (Ap K x) y) = Ap (Ap K K) (Ap S K).

Consider the following TRS:

Eq x x $\rightarrow$ True

The term Eq (+ 0 1) 1 does *not* match Eq x x. There is simply no substitution for x that achieves this. The term Eq (+ 0 1) (+ 0 1) does match Eq x x with σ(x) = + 0 1.

Consequently, whether or not a subterm is a redex now depends on the *structure* of the rewrite rules in the TRS itself: a subterm is a redex iff there exists a rule in the TRS with a pattern that matches the subterm. This is in contrast with the λ-calculus where a redex is created whenever a function is applied to an argument.

4.2.2 Rewriting a term

A **rewrite** (or **reduction**) of a term τ is the replacement of a redex of τ, say τ', by the right-hand side of the rule using the substitution function of the corresponding match: σ. The redex τ', which is equal to $\sigma(\alpha_i)$, is replaced by $\sigma(\beta_i)$: the right-hand side in which the substitution function is applied on the variables (by σ every variable is bound to a subterm of the redex).

Consider the TRS

Hd (Cons a b) $\rightarrow$ a

The term Hd (Hd (Cons (Cons 0 Nil) Nil)) can be rewritten as follows using the only rewrite rule in the TRS:

Hd (<u>Hd (Cons (Cons 0 Nil) Nil)</u>)	‖	σ(a) = Cons 0 Nil,	σ(b) = Nil
$\rightarrow_{\text{Hd}}$ <u>Hd (Cons 0 Nil)</u>	‖	σ(a) = 0,	σ(b) = Nil
$\rightarrow_{\text{Hd}}$ 0			

where the subscript Hd on the arrow indicates that the rule for Hd is used.

This rewriting of terms is the TRS equivalent of β-reduction in the λ-calculus. The main difference is that a λ-expression like (λx.M) N is always a redex and that it binds exactly one variable. For TRSs it depends on the set of rules in a concrete TRS whether a (sub)term is a redex or not. Furthermore, in TRSs functions can be specified with more than one argument. These rules can only be applied if *all* the arguments are present and match the pattern. Therefore, no currying is needed to simulate functions with more than one argument.

Example of functions with more than one argument:

```
G x y    →   x
H x y z  →   G z x
```

The term G 2 3 $\rightarrow_G$ 2, but the term G 2 will not be reduced: it is not a redex because it has only one argument and not two.

Name conflicts, as they occurred in the λ-calculus, cannot arise in TRSs. The initial term is a closed term and free variables cannot be introduced in the rewrite rules. So the term that is rewritten will never contain variables, therefore no name conflicts occur.

4.2.3 δ-rules

As in the λ-calculus there can exist externally defined functions and constants. Such an externally defined function is called a **δ-rule** or a **δ-function**. If a δ-function is applied to the number of arguments it needs, the corresponding subexpression is called a **δ-redex**. If the external function is applied it is called a **δ-reduction step.** Although it is possible to define rewrite rules with the same behaviour directly in TRSs, it is sometimes more convenient for readability and illustration purposes to assume the availability of these external functions.

Example of a δ-reduction step, assuming that + is an externally defined δ-rule and 2 and 3 are externally defined constants:

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

An external function like + requires that its arguments are indeed constants of a specific type, in this case the arguments should be constant numbers. If this is not the case, the corresponding subexpression is not a δ-redex until this restriction is satisfied. This implies that one first has to reduce the arguments of + before this δ-rule can be applied.

4.3 Reduction sequences and normal forms

One reduction is often called a **reduction step** (or **rewrite step**). A **reduction sequence** (or **rewrite sequence**) of a term is a sequence of zero or more reduction steps (or rewrites) performed on that term. If term τ_1 can be reduced by a sequence of reductions to expression τ_2, τ_2 is called a **reduct** of τ_1. This is denoted as $\tau_1 \rightarrow^* \tau_2$. So $\rightarrow^*$ is the reflexive transitive closure of $\rightarrow$.

Given a TRS, a term is in **normal form** if no rule in the TRS matches the term or one of its subterms. Note that whether or not a term is in normal form depends on the concrete TRS. Given a TRS, a term has a normal form if there exists some sequence of reduction steps that results in a term in normal form.

A term is in **head normal form** if the term as a whole is not a redex and the term as a whole will never become a redex. Such a term need not be in normal form since it may contain subterms that are redexes. Whether or not a term as a whole will ever become a redex is in general undecidable, so it is also undecidable whether a term is in head normal form. In many cases, however, the head normal form property is decidable, e.g. when the term begins with a constructor it is trivially in head normal form.

Examples of head normal forms; take the following rules:

```
H (Cons a b) 0   →   a
H Nil 1          →   Nil
```

Cons (H Nil 1) 1	is clearly in head normal form because the head of the term is a constructor
H (H Nil 1) 0	is in head normal form although the head is a function and the second argument matches. However, the first argument is a redex that reduces to Nil
H (H Nil 1) 1	is not in head normal form

Since the head normal form property is in general undecidable, the following notion is used to define a decidable approximation. A **rule partially matches** a term if its pattern partially matches the term. A **pattern α_i partially matches** a term τ if firstly the first constant of α_i equals the first constant of τ and secondly for each argument τ' of the first constant of τ corresponding to a non-variable argument α_i' of the first constant of α_i, α_i' also partially matches τ' or there exists a rule with a pattern partially matching τ'. Intuitively, a rule partially matches a term if by just looking at the left-hand sides the rule might after some reductions match the term. In Klop (1992), a term for which there exists a partially matching rule is called a **soft term**.

Illustrating the definition of a partial match:

```
H (Cons a b) 0   →   a
H Nil 1          →   Nil
```

H Nil 0	there is no rule with a pattern partially matching this term
H (H Nil 1) 0	partial match for the first rule of H; H Nil 1 is a redex
H (H (H Nil 1) 0) 0	partial match for the first rule of H; H (H Nil 1) 0 itself partially matches the first rule of H

A term is in **strong head normal form** if none of the rules partially matches this term. It is decidable whether or not a term is in strong head normal form. A term in strong head normal form does not partially match any rule so it is also in head normal form. A term in head normal form need not be in strong head normal form (see the examples above).

It is important to note that because of the rewrite semantics of TRSs there is a difference between TRSs and functional languages with respect to normal forms for the case of partial functions:

> Difference between functional languages and rewriting system for a partial function:
>
> ```
> F 1  →  2
> F 2  →  3
> ```
>
> Suppose the term to be reduced is `F 3`: in a functional language this would result in an error because it does not match any of the rules; in a rewriting system it is just not rewritten because it does not match, and the normal form is then simply `F 3`.

A function that is applied with a number of arguments that does not correspond to the number of arguments specified on a left-hand side, will never be a redex. This is why one can consider occurrences of constants with different numbers of arguments as different constants. This has no influence on the semantics.

4.4 Properties of TRSs

In this section some properties of TRSs are considered that are independent of the specific term one wants to rewrite and independent of the specific reduction strategy (see Section 4.5) that is used.

4.4.1 Ambiguity

The patterns specified in a TRS determine whether or not a (sub)term is a redex. However, it is possible that a term matches several left-hand sides of the rewrite rules of the TRS. In that case there are several alternative rewrite rules by which the term can be rewritten. A TRS is **ambiguous** iff it contains rules of one of the following types:

- **non-deterministic** rules, i.e. their left-hand sides are totally overlapping;
- **partially ambiguous** rules, i.e. their left-hand sides are partially overlapping.

Two left-hand sides of two *different* rules are **totally overlapping** if a term exists that is an instance of both left-hand sides. Two left-hand sides α_1 and α_2 of two (not necessarily different) rules are **partially overlapping** if a term exists that is an instance of α_1 and that is also an instance of a non-variable, *proper* subterm of α_2. Two such (partially) overlapping left-hand sides are also called a **critical pair**. Depending on the right-hand sides the existence of a critical pair can invalidate uniqueness of normal forms.

Example of non-deterministic rules:

```
Choose x y   →   x
Choose x y   →   y
```

Given this TRS, the term Choose 1 2 can be reduced to either 1 or to 2: there is no unique normal form.

The following very familiar looking set of definitions forms a TRS that is non-deterministic. The term Fac 0 matches both left-hand sides. Unlike function definitions in Miranda, in ordinary TRSs all rules have equal priority. Choosing the wrong rule will lead to unexpected and unwanted results.

```
Fac 0   →   1
Fac n   →   * n (Fac (– n 1))
```

Example of partial ambiguity:

```
G ( K n 1 ) →   1
K x y       →   x
```

Given this TRS, the term K 0 1 is a redex and an instance of the subterm K n 1 in G (K n 1). The term G (K 0 1) can be reduced to either 1 or to G 0.

A more subtle example of a partial ambiguous TRS:

```
F ( F x )   →   A
```

This TRS is ambiguous with itself. The term F (F 1) is a redex and an instance of the subterm F x in F (F x). Given this TRS the following term can be reduced as follows:

F (<u>F (F 1)</u>) $\rightarrow_F$ F A	<u>F (F (F 1))</u> $\rightarrow_F$ A

So the term F (F (F 1)) has two different normal forms in this TRS.

4.4.2 Comparing

A TRS is **comparing** iff one or more left-hand sides of the rules of the TRS is **comparing**, i.e. it contains a multiple occurrence of the same variable, such as

$$\text{Eq x x} \quad \rightarrow \quad \text{True}$$

A comparing rule is also called **non-left-linear** because of the correspondence with multiple occurrences of variables in multiplicative terms of polynomial expressions.

Restricting a TRS to be non-comparing is natural. A rewrite rule using a left-hand side that contains a multiple occurrence of the same variable induces a test on *syntactic* equality of the corresponding arguments of the term. However, it is possible that the arguments are *semantically* equal but syntactically not. In general, it is undecidable whether two terms are semantically equal. Comparing is therefore only sound if the two terms are in normal form, i.e. they do not contain any redexes (this is similar to Church's δ, see Section 3.7.6). In that case terms that are semantically the same are also syntactically equal, because the only semantics is the rewriting semantics.

In the term below the arguments are not always syntactically equal. If they are in normal form, then syntactic equality implies semantic equality.

$$\text{Eq } (\underline{+\ 1\ 1})\ (+\ 1\ 1) \quad \rightarrow_{\delta+} \quad \text{Eq } 2\ (\underline{+\ 1\ 1}) \quad \rightarrow_{\delta+} \quad \underline{\text{Eq } 2\ 2} \quad \rightarrow_{\text{Eq}} \quad \text{True}$$

4.4.3 Confluency

A TRS is **confluent** (or has the **Church–Rosser property**) iff for every two (divergent) reduction sequences $\tau \rightarrow_* \tau_1$ and $\tau \rightarrow_* \tau_2$ there is a τ_3 such that there are two (convergent) reduction sequences $\tau_1 \rightarrow_* \tau_3$ and $\tau_2 \rightarrow_* \tau_3$ (see also Section 3.4). The Church–Rosser property implies the *unique normal form property*. As in the λ-calculus, this does not imply that every term has a normal form. It states only that *if* there is a normal form, it is a unique one.

Not every term has a normal form in a confluent TRS:

$$\begin{array}{lll} \text{F x} & \rightarrow & \text{G x} \\ \text{G x} & \rightarrow & \text{F x} \\ \text{H 0} & \rightarrow & 1 \end{array}$$

This TRS is confluent. H 0 has as normal form 1, However, the term F 1 does not have a normal form.

But, in contrast with the λ-calculus, a TRS is in general *not* confluent.

The examples in the previous section show that ambiguity can interfere with confluency. However, ambiguity does not necessarily lead to non-confluency in all cases. Furthermore, comparing can also be a source of non-confluency. The following examples clarify this.

Ambiguity versus confluency:

- a non-ambiguous and confluent TRS:

K x y	→	x

- ‘parallel OR’, an ambiguous but confluent TRS:

Or	True	x	→	True
Or	x	True	→	True
Or	False	False	→	False

- an ambiguous and non-confluent TRS:

F 0	→	0
F x	→	1

The following marvellous example is due to J.W. Klop. It shows that comparing is also a source of non-confluency. This is caused by the fact that the terms that are compared need not be in normal form.

- a non-ambiguous and non-confluent TRS (Klop’s example):

D x x	→	E
C x	→	D x (C x)
A	→	C A

Explanation: the term C A can be reduced in the following ways:

$\underline{\text{C A}}$		C $\underline{\text{A}}$	
$\rightarrow_{C}$	D $\underline{\text{A}}$ (C A)	$\rightarrow_{A}$	C ($\underline{\text{C A}}$)
$\rightarrow_{A}$	$\underline{\text{D (C A) (C A)}}$	$\rightarrow_{C}$	C (D $\underline{\text{A}}$ (C A))
$\rightarrow_{D}$	E	$\rightarrow_{A}$	C ($\underline{\text{D (C A) (C A)}}$)
		$\rightarrow_{D}$	$\underline{\text{C E}}$
		$\rightarrow_{C}$	D E (C E)

The term D E (C E) cannot be reduced to E any more. Note that just before the D-reductions are done the terms can still be reduced to a common reduct. Further note that the arguments of D are not in normal form when the D-rule is applied.

Orthogonal TRSs

A TRS is **orthogonal** iff it is non-ambiguous and non-comparing.

Theorem (Klop, 1992): Every orthogonal TRS is confluent.

Hence, in an orthogonal TRS a unique normal form is guaranteed. As in the λ-calculus it depends on the chosen reduction order whether or not the reduction process will find the normal form (if it exists). In Section 4.6 a normalizing strategy for orthogonal TRSs is presented.

4.5 Reduction strategies

A **reduction strategy** is a function that, given a TRS and a term, prescribes which redexes in the term have to be rewritten next. It also prescribes for each redex, if there is more than one matching rule, which one of these rules has to be applied. It is possible that the strategy function prescribes more than one redex to be reduced next in arbitrary order. Such a strategy is called a **parallel reduction strategy**. It is also possible that the strategy function returns more than one redex or rule to choose from. In that case the strategy function is called **non-deterministic**. Note that for the λ-calculus we only considered non-parallel and deterministic strategies. In TRSs parallel and non-deterministic strategies play a more important role than in the λ-calculus.

Without knowing the reduction strategy it is in general impossible to reason about the actual evaluation of terms in a TRS: a TRS with a term specifies many possible computations. When the reduction strategy is known, a term with a TRS fully specifies the computation and then it is possible to reason about properties of this computation.

A **reducer** is a process that consecutively reduces the redexes that are indicated by some strategy. The result of a reducer is reached as soon as the reduction strategy does not indicate any more redexes. A reduction strategy is **normalizing** iff for any term, having a normal form, the reducer applying the strategy will terminate at a normal form. **Parallel reducers** are assumed to reduce the redexes indicated by a parallel reduction strategy in parallel, i.e. in arbitrary order. **Non-deterministic reducers** should choose non-deterministically one of the redexes offered by a non-deterministic strategy.

If the intention of non-deterministic rules is indeed to make a non-deterministic choice between the overlapping rules, a non-deterministic reduction strategy must be used. Such a strategy will yield both rules as candidate to rewrite a matching redex. A non-deterministic reducer will arbitrarily choose one of these rules.

The reduct depends on the chosen rewrite rule (the first or the second rule for Choose):

Choose x y	$\rightarrow$	x
Choose x y	$\rightarrow$	y

<ins>Choose 1 2</ins> $\rightarrow_{\text{Choose}_1}$ 1 | <ins>Choose 1 2</ins> $\rightarrow_{\text{Choose}_2}$ 2

Of course it is possible to define an algorithm that finds the normal form for all TRSs: just produce breadth-first the reduction tree of a term with all possible reductions. If a normal form exists it will certainly be found and then the algorithm can stop. This is not a very efficient algorithm. The algorithm is not a strategy since the algorithm does not just indicate which redexes are to be reduced next possibly in parallel, but indicates which possible reducts have to be investigated and remembered. In the following we shall only investigate proper strategies.

A redex in a term is **needed** iff for all possible reduction sequences of this term ending at a normal form this redex (or if it is copied by a rule, one of these copies) must have been reduced. In confluent TRSs a reduction strategy is trivially normalizing if it chooses needed redexes only.

Needed redex: consider the following (part of a) TRS:

If True x y	$\rightarrow$	x
If False x y	$\rightarrow$	y

Consider the term If (= 0 0) (+ 1 1) (+ 2 2) (assume that the TRS has rules for + and =). Then it contains three redexes, namely (= 0 0), (+ 1 1) and (+ 2 2). Enumerating all possible reduction sequences shows that (= 0 0) and (+ 1 1) are needed redexes and (+ 2 2) is not.

Whether or not a redex is needed is in general undecidable. It is even undecidable for orthogonal TRSs. So always choosing the needed redexes is not a practical strategy for orthogonal TRSs or indeed for confluent TRSs in general. However, for (subclasses of) orthogonal TRSs it is possible to define decidable strategies that are normalizing for that class. Sometimes it is even possible to define for a certain subclass a decidable normalizing strategy that only reduces needed redexes.

4.6 Orthogonal TRSs

Orthogonal TRSs are confluent (see Section 4.4.3). In this section various *rule-based* and *non-rule-based* reduction strategies are investigated. There is a normalizing strategy for orthogonal TRSs (parallel outermost), but this strategy is not efficient: generally too many redexes are indicated. There are other more efficient strategies, but they are only normalizing for certain subclasses of orthogonal TRSs.

4.6.1 Classes with non-rule-based normalizing strategies

In a **non-rule-based strategy** the decision of which redex to reduce depends only on the position of the redex in the term.

Given the following rule:

F x y $\rightarrow$ 1

and the following term (Figure 4.5): Triple (F 0 (F 1 2)) (F (F 3 4) 5) (F 6 7)

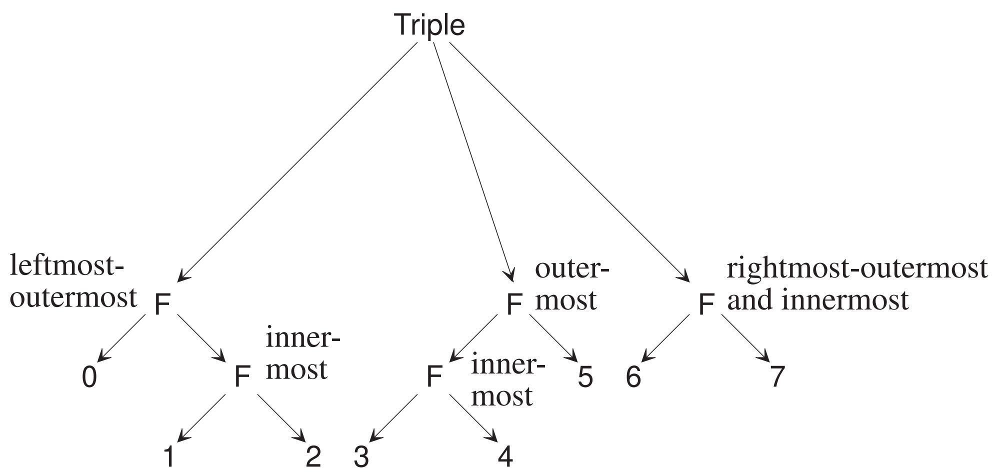

Figure 4.5 Leftmost, rightmost, outermost and innermost redexes in the tree of the term Triple (F 0 (F 1 2)) (F (F 3 4) 5) (F 6 7).

Definitions of *leftmost*, *rightmost*, *innermost*, *outermost redexes* and *strategies* are similar to the definitions for the λ-calculus (see Chapter 3). For TRSs *parallel* versions of innermost and outermost strategies are also considered.

Innermost strategy

Innermost strategies are not normalizing for arbitrary orthogonal TRSs.

An orthogonal TRS for which innermost strategies are not normalizing:

K x $\rightarrow$ 0
W $\rightarrow$ W

Using an innermost strategy the reduction process starting with the term K W will not terminate. On the other hand, using the leftmost-outermost strategy the reduction process starting with K W will terminate with the normal form 0.

Innermost strategies are normalizing for those confluent TRSs for which infinite reduction sequences do not exist, called **strongly normalizing TRSs**. Innermost strategies correspond with eager evaluation in functional languages. So innermost strategies rule out concepts similar to lazy evaluation.

Leftmost-outermost strategy

The leftmost-outermost strategy chooses the leftmost of the outermost redexes. In the λ-calculus the leftmost-outermost reduction strategy is normalizing, but even for orthogonal TRSs this is not true.

An orthogonal TRS for which a leftmost-outermost reducer is not normalizing:

```
F x 0   →   1
G       →   0
W       →   W
```

Leftmost-outermost does not find the normal form 1 of F W G in this TRS:

$\mathsf{F}\ \underline{\mathsf{W}}\ \mathsf{G} \rightarrow_{\mathsf{W}} \mathsf{F}\ \underline{\mathsf{W}}\ \mathsf{G} \rightarrow_{\mathsf{W}} \mathsf{F}\ \underline{\mathsf{W}}\ \mathsf{G} \rightarrow_{\mathsf{W}} \ldots$

Note the difference between this strategy (non-rule-based) and the way redexes are chosen in Miranda (rule-based). In the example the redex W is reduced because it happens to be the leftmost-outermost redex in the term to be reduced. But from the pattern of the rewrite rule for F it can be deduced that the evaluation of this argument was not needed.

Left-normal TRSs

An orthogonal TRS is called **left-normal** if in no left-hand side a variable occurs before a constant.

Example of a left-normal system:

```
Hd (Cons a b) →   a
```

Not a left-normal system:

```
F (Cons a b) (Cons c d) →   0
```

The variable a on the left occurs before the second Cons.

In left-normal systems the leftmost-outermost redex is always needed. So leftmost-outermost is normalizing for left-normal systems.

The restriction that in a pattern no variable occurs before a constant is a serious restriction. It implies that many patterns are not allowed. Still, the expressive power of such systems is very high. For instance, every combinatorial system (see below) is left-normal.

Combinatorial TRSs

A subclass of left-normal systems is the class of **combinatorial systems** (see also Section 3.6.1). Rules in combinatorial systems all have the following form:

Combinatorial systems, general form:

Ap (Ap… (Ap C x_1)… x_{n-1}) x_n $\rightarrow$ *A term with only* Ap*s and variables*

Such a rule defines the *combinator* C. These combinators are equivalent to those defined in the λ-calculus in the previous chapter:

Equivalent definition to the one above:

C $\equiv$ λx_1. … $\lambda x_{n-1}.\lambda x_n$.*term with juxtaposition instead of* Ap*s*

Consider the following λ-calculus combinators:

S $\equiv$ λx.λy.λz. x z (y z)
K $\equiv$ λx.λy.x
I $\equiv$ λx.x

The same combinators defined in a TRS:

Ap (Ap (Ap S x) y) z	$\rightarrow$	Ap (Ap x z) (Ap y z)
Ap (Ap K x) y	$\rightarrow$	x
Ap I x	$\rightarrow$	x

The only function that is present in these combinatorial TRSs is the dyadic function Ap(ply). The combinators are *discriminating* constructors: they determine which of the alternative rules for the apply function is applicable.

An example of reduction in a combinatorial system:

Ap (Ap (Ap S K) K) X $\rightarrow_S$ Ap (K X) (K X) $\rightarrow_K$ X

Every combinatorial system is left-normal. Therefore, leftmost-outermost is normalizing for the class of combinatorial systems.

Parallel-outermost strategy

Leftmost-outermost is not normalizing for arbitrary orthogonal TRSs. It is even impossible to define a decidable normalizing strategy that indicates separately for each rule which argument has to be reduced first. The normal form can only be found using a parallel reduction strategy.

The example given by G. Berry shows that TRSs can be constructed for which it is undecidable to indicate an argument that has to be reduced first:

```
B 0 1 x  →   1
B 1 x 0  →   2
B x 0 1  →   3
```

Berry's TRS is orthogonal, so it guarantees a unique normal form. Suppose the term B R_1 R_2 R_3 has to be reduced, given that the reduction process performed on one of the arguments may not terminate. The strategy still has to choose one of the arguments, but unfortunately it does not know beforehand which one will not terminate. So enforcing argument evaluation at one of the arguments for which a pattern is specified may start a non-terminating computation. The only safe way to reach the normal form is to reduce each argument of B a finite number of steps to see if the term as a whole has become a redex. The parallel-outermost strategy satisfies this idea of reducing each argument of B one step at a time.

The **parallel-outermost** reduction strategy indicates all outermost redexes. It is assumed that all these redexes are reduced in parallel (i.e. in arbitrary order) in one reduction step of the reducer. The parallel-outermost reduction strategy is not normalizing for a general TRS.

Consider the following (ambiguous) TRS:

```
F G 0   →   W
F A 0   →   B
G       →   A
W       →   W
```

A parallel-outermost reducer would reduce the next term as follows:

$\text{P G (F A 0)} \rightarrow_{G,F_2} \text{P A B}$

Parallel-outermost does not find the normal form B of F G 0 in this TRS:

$\underline{\text{F G 0}} \rightarrow_F \underline{\text{W}} \rightarrow_W \underline{\text{W}} \rightarrow_W \dots$

Theorem (O'Donnell, 1985): Parallel outermost is normalizing for orthogonal TRSs.

By reducing all outermost redexes by one step a breadth-first reduction order is chosen. Clearly, at least one of the outermost redexes is needed for the reduction to normal form, but it is unknown which ones are needed and which are not. So it is apparent that the parallel-outermost strategy is not always the most efficient one. It indicates all outermost redexes, but not all of them might be needed.

Parallel-outermost finds the normal form of the following orthogonal TRS that includes Berry's example. However, too many redexes are reduced: the W-steps performed with the parallel-outermost reduction are not needed.

```
B 0 1 x  →  1
B 1 x 0  →  2
B x 0 1  →  3
W        →  W
A        →  1
C        →  A
```

B $\underline{\mathsf{W}}$ 0 $\underline{\mathsf{C}}$ $\rightarrow_{\mathsf{W,C}}$ B $\underline{\mathsf{W}}$ 0 $\underline{\mathsf{A}}$ $\rightarrow_{\mathsf{W,A}}$ $\underline{\mathsf{B\ W\ 0\ 1}}$ $\rightarrow_{\mathsf{B_3}}$ 3

4.6.2 Classes with rule-based normalizing strategies

In a **rule-based strategy** the decision which redex to choose depends on the kind of rules in the TRS. Since analysis of the rules that takes the right-hand side into account is bound to be undecidable, only left-hand side analysis is treated. Even by looking only at the left-hand side it is in general undecidable for orthogonal TRSs to indicate needed redexes, as is illustrated by Berry's example. So, in this section, for orthogonal TRSs restrictions are imposed on the left-hand side that allow a decision tree to be defined that is used by the reduction strategy to point out the next needed redex.

Discriminating positions strategy

Next we will present a strategy that uses the patterns of the rewrite rules to produce a decision tree. In contrast with the term-based strategies the decision tree does not always indicate a redex if there is one in the term to be reduced. So the strategy will be normalizing only for certain classes of orthogonal TRSs.

The discriminating position strategy: consider the set of all **partially matching rules**, i.e. all rules that partially match the given *term*;

- if this set of rules is empty no redex is indicated: the term is in strong head normal form;
- if the term is a redex for one of the rules in the set, then this redex is indicated;

- otherwise:

  - if there is a proper subterm of the given term for which in every rule in the set in the corresponding position in the pattern a non-matching subpattern appears then this subterm is in a **discriminating position**;

  - if there are subterms on discriminating positions, first the strategy is recursively applied on the leftmost-outermost of such subterms, and afterwards the strategy is recursively applied again on the given term;

  - otherwise, no redex is indicated.

If a normal form is required the strategy must recursively be applied to all subterms of obtained head normal forms.

Illustrating discriminating positions:

```
G (Pair a 1) 0   →   a
G (Pair a 0) 1   →   Nil
A                →   Pair A B
B                →   0
```

For G A B both the first and the second arguments are in a discriminating position. A will be indicated since it is the leftmost-outermost redex. For G (Pair A B) B the second subargument of the pair will be indicated.

An orthogonal TRS without discriminating positions:

```
B 0 1 x  →   1
B 1 x 0  →   2
B x 0 1  →   3
```

In Berry's example given above there is a variable on each argument position so no redex on these positions is indicated by the strategy. Clearly, for this TRS the discriminating position strategy is not normalizing.

The example above shows that when discriminating positions do not exist the strategy is not normalizing. The examples below show that the discriminating position strategy is not always normalizing even if discriminating positions do exist. The reason for this is that the strategy when it is recursively applied on a discriminating position does not examine the term as a whole but only tries to match subterms locally against the discriminating patterns.

An orthogonal TRS for which the discriminating position strategy is normalizing:

```
F  (Cons a b)  (Cons 1 c)  →   a
F  (Cons a b)  (Cons 2 c)  →   b
F  Nil               c     →   c
```

The discriminating position strategy will first choose the first argument a_1 of a term F a_1 a_2 for evaluation. The second argument of F is only chosen if the term after evaluation partially matches the first two rule alternatives.

Another orthogonal TRS for which the discriminating position strategy is normalizing:

```
F (D x A) B   →   1
F (D A x) C   →   2
```

In the term F a_1 a_2 the subterms a_1 and a_2 are in a discriminating position. First a_1 will be chosen, then a_2.

An orthogonal TRS for which the discriminating position strategy is not normalizing:

```
F (G x A) B   →   1
F (G A x) C   →   2
G D D         →   3
```

In the term F (G a_1 a_2) a_3 the subterm (G a_1 a_2) is in a discriminating position. The strategy is recursively applied on G a_1 a_2. Now it finds that both a_1 a_2 are in discriminating positions in the rule for G. They are chosen next. A more clever strategy would reduce a_3 in F (G a_1 a_2) a_3 next. Compare this with the example above, where this order was taken by absence of the G rule.

For left-normal systems the discriminating position strategy coincides with the leftmost-outermost strategy. Therefore the discriminating position strategy is trivially normalizing for left-normal systems. The class of orthogonal TRSs for which the discriminating position strategy is normalizing is called **locally discriminating** TRSs since each recursive call of the strategy considers only locally the given subterm. For this class of TRSs the strategy always indicates a needed redex. The class is characterized by a decidable syntactical test that guarantees that a discriminating position is indeed needed.

Huet–Lévy strategy

Huet and Lévy (1979) have defined a class of orthogonal TRSs and a corresponding strategy (the **Huet–Lévy** strategy) in which they always take as much of the term as a whole into account as necessary. In their **strongly sequential** or **globally discriminating** systems needed redexes are identified by looking at the left-hand sides of the rules. Strongly sequential systems are a *decidable* subclass of *sequential systems*. In sequential systems the right-hand sides are also taken into account to identify a needed redex. Berry's TRS is clearly not strongly sequential. Below we give examples of TRSs that are globally discriminating but not locally discriminating. Furthermore another example of an orthogonal not strongly sequential TRS is given.

A strongly sequential TRS that is not locally discriminating:

```
F (G x A) B   →   1
F (G A x) C   →   1
G D D         →   2
```

For this TRS it turns out to be possible to identify in every term a needed redex if it exists. For instance, in F (G a_1 a_2) a_3 the subterm a_3 is needed.

Another strongly sequential TRS that is not locally discriminating:

```
F (F x A) B   →   …
F C (F D x)   →   …
```

For this TRS it turns out to be possible to identify in every term a needed redex if it exists. For instance, in F (F x_1 x_2) (F x_3 x_4) both x_2 and x_3 are needed. The TRS is not locally discriminating because x_1 would have been indicated wrongly by the discriminating position strategy.

An orthogonal TRS that is not strongly sequential:

```
F (G A x) (F B y)   →   …
F (G x A) (F C y)   →   …
G D D               →   …
```

In the term F (G x_1 x_2) (F (G x_3 x_4) x_5) one cannot identify a needed redex (supposing the xs are complicated expressions with undecidable reducts). If the second rule for F is applicable x_2 has to be reduced and if the first rule for F is applicable x_1 has to be reduced. Which rule for F is applicable is determined by the second argument of F. It is important to note that this argument by itself can also be a redex for one of the rules for F and that, in order to determine for which rule this is the case, the reduction of x_5 is needed. This means that on the one hand it depends on the reduction of G x_3 x_4 (and consequently on the reduction of x_3 and x_4) whether the rule as a whole might be a redex for the first or the second rule for F. On the other hand it depends on the reduction of x_5 whether F (G x_3 x_4) x_5 might become a redex as a whole for one of the rules for F. So it is impossible to indicate a needed redex, i.e. to make a choice between the xs.

Unfortunately there does not exist an efficient test to determine whether a system is strongly sequential or not. Furthermore it is hard to see whether or not a specific TRS is strongly sequential and how evaluation will proceed. The Huet–Lévy strategy will not be fully treated here. It would require extending the notions of partial match and strong head normal form to definitions that recursively take more of the whole term into account.

4.7 Priority rewrite systems

There are several other interesting related systems for term rewriting that are not treated in this textbook, such as **conditional rewriting systems** (Meinke and Tucker, 1992), in which rewriting can only proceed if a guard is fulfilled (like in Miranda) and **narrowing rewriting systems** (De Groot and Lindstrom, 1986), in which the term that is rewritten may have free variables. **Narrowing** means that the variable in the term is instantiated with (is narrowed to) the value that is specified in a pattern. There are various ways of using this mechanism. Narrowing rewriting systems are investigated in the context of logical programming languages.

This section treats term rewriting systems with priorities (Baeten *et al.*, 1987), also called ***priority rewrite systems*** (PRSs), that try to capture the theoretical basis of the order dependency in the rules. The difference with an ordinary TRS is that in a PRS, whenever a rule matches a given redex, it can only be chosen if none of the rules with higher priority can *ever* match the (internally rewritten) term. A reduction step is called **internal** if it proceeds entirely within the arguments of the leftmost symbol of the term.

In a PRS the factorial function can be defined as follows:

```
  Fac 0  →  1
> Fac n  →  * n (Fac (– n 1))
```

The greater than sign indicates that the top rule has priority over the bottom rule. If the argument of Fac is 0 then the first rule is applicable. However, if it is somehow known that the argument can never be reduced to 0, then the priority of the rules makes it possible to choose the second rule without actually reducing the argument. So the second rule can be applied on Fac (+ 1 1) but not on Fac (– 1 1).

Of course it is in general undecidable whether or not a term can ever be made to match a pattern via internal rewrites. So a practical approximation is simply to force the evaluation of the term. **Forcing evaluation** means reduction to strong head normal form via some strategy. **Uniformly forcing evaluation** means that the choice for a rule with lower priority is only made if the subterms on ambiguous positions are reduced to strong head normal form. In this way, no choice for a rule with lower priority is ever made wrongly.

Forcing evaluation solves the problems in examples like Fac above. Fac (+ 1 1) and Fac (– 1 1) are first reduced to, respectively, Fac 2 and Fac 0, and then the choice between the prioritized rules is made. However, due to forcing evaluation non-termination may occur, although semantically this was avoidable.

Non-termination caused by forcing evaluation:

```
  G 0  →  1
> G n  →  2
  F    →  G F
```

A very clever strategy could know that F can only reduce to G (G (…(G F)...)) so never to G 0, so the second rule of G can be applied. The normal form of F is 2. Forcing evaluation of the argument of G leads to non-termination. The argument of G is not in strong head normal form since a partial match occurs.

Furthermore, in partially ambiguous cases a normal form may not be reached, just as was the case without priorities.

Normal forms in partially ambiguous cases not reached by forcing evaluation:

```
  F G  →  1
> F A  →  F G
  G    →  A
```

The normal form of F G is 1. However, forcing evaluation of the argument of F would lead to non-termination.

A strategy still must indicate which of the arguments must be forced for evaluation. This is essentially the same problem as with ordinary TRSs. For orthogonal TRSs the priorities do not make a difference. It is possible to investigate again various non-rule-based and rule-based strategies for PRSs. However, we only investigate one specific strategy: the functional strategy.

4.7.1 The functional strategy

The **functional strategy** combines the discriminating position strategy with uniformly forcing evaluation in priority rewrite systems such that the reduction proceeds in a similar way to most functional languages; subtle uses of priority for only certain sets of rules are not used, but textual order is assumed always to correspond to increasing priority; if a normal form is required the strategy is recursively applied to all subterms of obtained head normal forms. Effectively, this means that the discriminating position strategy is applied for one rule alternative at a time, with as a consequence that each non-variable subpattern under consideration induces a discriminating position. This leads to an efficient reduction strategy that is relatively easy to understand. Priority rewrite systems with an additional restriction that the functional strategy is being used are called **functional term rewriting systems** (FTRSs).

In other words, the functional strategy chooses redexes as follows: if there are several rewrite rules for a particular function, the rules are tried in textual order; patterns are tested from left to right; and evaluation of an actual argument is always forced when this argument must match a non-variable in the corresponding pattern (even in overlapping cases). This forcing of the evaluation of an argument reduces the argument to strong head normal form. If in the resulting term the argument matches the corresponding part of the pattern, the next (sub)argument is tried until the whole pattern matches and the rule can be applied. If in the resulting term the argument does not match the corresponding part of the pattern, the left-hand side of the next rule is tried to match the resulting term, proceeding in the same way, enforcing evaluation of subterms. If in the end none of the rules matches, the whole term is in strong head normal form. If a normal form is required the strategy is recursively applied to all subterms of obtained head normal forms.

A natural way to define the factorial function is:

```
Fac 0   →   1
Fac n   →   * n (Fac (– n 1))
```

This example has the intended semantics if the functional strategy is chosen. Fac (– 1 1) will be reduced to Fac 0 and the priority of the rules guarantees that Fac 0 matches the first rule and not the second.

If the functional strategy yields a normal form, it is a correct one with respect to the priority semantics. Of course, the functional strategy is not normalizing for a general PRS.

A (non-ambiguous) PRS for which the functional strategy is not normalizing:

```
F 0 0   →   1
F x 1   →   2
W       →   W
```

$F\ \underline{W}\ 1 \ \rightarrow_W\ F\ \underline{W}\ 1 \ \rightarrow_W\ \ldots$

Owing to the forced evaluation the normal form 2 of F W 1 is not reached in this PRS: owing to the priority of the rules the functional strategy does not first reduce the second argument of F, but first forces evaluation to establish whether or not the first argument reduces to 0.

So the functional strategy is actually a practical compromise: it is efficiently implementable and it makes it possible to express computations in a user-friendly manner. It has the following advantages:

- it is easy to explain;
- it is easy to use: the priority in the rules makes alternative rules similar to an if-then-else control structure;
- it is easy to understand, such that a programmer will not often specify unintended infinite computations;
- it can be implemented very efficiently.

The class of TRSs without priority for which the functional strategy is normalizing is characterized in Toyama *et al.* (1993).

4.8 TRSs as a basis for functional languages

TRSs have much in common with functional languages. Since TRSs have even greater expressive power than the λ-calculus, it will certainly be possible to translate functional programs into equivalent rewrite rules. An initial expression will be translated into a term that has to be rewritten.

However, there are several translation schemes possible depending on the kind of TRS one would like to obtain, such as an orthogonal, left-normal, combinatorial or priority system. This choice will influence:

- the complexity of the compilation scheme;
- the complexity of the obtained code (the number and size of the rewrite rules);
- the reduction strategy to be used;

- the number of reduction steps needed to reach the normal form (assuming that the compilation is correct and the normal form will be found if it exists);
- the efficiency of the reducer.

For instance, one can choose to compile to an orthogonal TRS and use the parallel-outermost strategy. This means that one has to make sure that patterns do not overlap, as is possible in functional languages.

Here we illustrate how the basic concepts of functional languages can be translated to an FTRS. It should not come as a surprise that such a compilation is a very straightforward one. Most of the details of an actual translation can be found in Chapter 9, where the translation from Miranda to GRSs is given. In most cases these translation schemes are also applicable to the translation to TRSs.

Booleans and the conditional

Booleans can be expressed as constants: True, False. Now a conditional expression can be created by

```
Cond True x y   →   x
Cond False x y  →   y
```

Recursion

Recursion can be expressed directly, e.g.

```
Fac 0   →   1
Fac n   →   * n (Fac (– n 1)
```

Representation of natural numbers

In TRSs one can represent natural numbers and arithmetical functions.

Define

```
0  ≡  Zero
1  ≡  Succ Zero
2  ≡  Succ (Succ Zero)
```

etc. Now a function for addition can be defined as follows:

```
Add x Zero      →   x
Add x (Succ y)  →   Add (Succ x) y
```

It will be clear that the use of the machine representation of numerals and δ-rules yields a much more efficient implementation.

Lists

Lists and other data structures can be defined easily using constructors.

```
x : y      ≡  Cons x y
[ ]        ≡  Nil
```

Now functions on these lists can be defined via pattern matching on the constructors:

```
Hd (Cons x y)    →    x
Tl (Cons x y)    →    y
```

Patterns

Patterns are part of TRSs as well as of FPLs.

Currying

In Miranda, all functions are curried and have at most one argument. Curried applications of functions can be realized in TRSs by introducing additional rewrite rules for those functions that are actually used in a curried way. These additional rules explicitly transfer curried applications of these functions into uncurried ones. Hence, currying is only performed when it is needed.

For instance, consider the well-known function Map:

```
Map  f   Nil            →    Nil
Map  f   (Cons a b)     →    Cons (AP f a) (Map f b)
```

Assume that all elements of an integer list have to be multiplied with the number 2, using this function Map. Assume that * is a predefined δ-rule for integer multiplication that is defined with arity 2. Now its curried application can be realized as follows by defining the following additional rule:

```
AP (AP * a) b  →    * a b
```

This AP-rule transforms the curried application of the multiplication into an uncurried application as shown in the following rewrite sequence:

<u>Map (AP * 2) (Cons 3 (Cons 4 Nil))</u>
$\rightarrow_{\mathrm{Map}_2}$ Cons (<u>AP (AP * 2) 3</u>) (Map (AP * 2) Cons 4 Nil)
$\rightarrow_{\mathrm{AP}}$ Cons (<u>* 2 3</u>) (Map (AP * 2) Cons 4 Nil)

$\rightarrow_*$	Cons 6 (Map (AP * 2) Cons 4 Nil)
$\rightarrow_{Map_2}$	Cons 6 (Cons (AP (AP * 2) 4) (Map (AP * 2) Nil))
$\rightarrow_{Ap}$	Cons 6 (Cons (* 2 4) (Map (AP * 2) Nil))
$\rightarrow_*$	Cons 6 (Cons 8 (Map (AP * 2) Nil))
$\rightarrow_{Map_1}$	Cons 6 (Cons 8 Nil)

In general, an extra AP-rule has to be defined for every function that is actually used in a curried way.

4.9 TRSs as a basis for implementations

In contrast with implementations based on the λ-calculus, name conflicts cannot occur in an implementation based on TRSs. Still, an implementation of general TRSs is not straightforward, in particular when efficiency is important. Only some subclasses and strategies can be implemented efficiently. Orthogonal systems require parallel-outermost as strategy, which is inherently inefficient. Left-normal systems can be implemented efficiently by means of the leftmost-outermost reduction strategy. Locally and globally discriminating systems can be efficiently implemented, but their characterization is difficult to comprehend. Efficient implementations of FTRSs which use the functional strategy can be obtained via the introduction of sharing (see the next chapter).

4.9.1 Combinators

As explained in the previous chapter, combinators (and hence also combinatorial systems) can be implemented efficiently. However, an advantage of the use of more general TRSs is that pattern information is explicitly present in the model allowing special kinds of analysis (for example the strictness analysis using abstract reduction in Chapter 7).

4.9.2 Sharing

To avoid unnecessary computations one can share terms in the actual implementation. In such a case, graphs are rewritten instead of terms. Because local definitions do not exist in TRSs, the problem that parts of the graph sometimes have to be copied does not occur. However, using graphs instead of terms will change the semantics in certain cases. This is discussed in the next chapter.

4.9.3 The functional strategy

The functional strategy chooses redexes in a similar way as in Miranda. So, as shown in Chapter 9, a functional program can be compiled to a PRS using a functional strategy in such a way that:

- the compilation scheme is straightforward;
- the length of the obtained code is linear with respect to the length of the original program.

If graphs are being used instead of terms, even state-of-the-art efficiency is possible (see Part 4). The language Clean (introduced in Chapter 8) which uses the functional strategy, is based on this extension to graph rewriting. In Chapters 10, 11 and 12 it is explained how an efficient implementation of Clean can be achieved.

Summary

- In TRSs tests on syntactic equality (*pattern matching*) are used to rewrite *terms* according to the specified *rewrite rules*.
- Most notions like *normal forms*, *strategies* and *rewriting* are similar to the corresponding notions in the λ-calculus. But for a general TRS:
  - *normal forms* are *not* unique;
  - the *head normal form* property is *undecidable*;
  - the *strong head normal form* property is *decidable*;
  - there is *no normalizing strategy*.
- *Ambiguity* and *comparing* are the sources of non-confluency, so *orthogonal systems* are confluent.
- The *parallel-outermost strategy* is normalizing for *orthogonal systems*; it is an inefficient strategy because it can indicate redexes that are not needed.
- The *leftmost-outermost strategy* is normalizing for *left-normal systems* among which are *combinatorial systems*; left-normal systems allow only a very restricted use of patterns.
- The *discriminating position strategy* as well as the *Huet–Lévy strategy* are normalizing for, respectively, *locally* and *globally discriminating systems*. The strategies are efficient but their classes of TRSs are difficult to characterize and it is difficult to understand how evaluation will proceed.
- The *functional strategy* combines the discriminating position strategy with uniformly forcing evaluation to *strong head normal form* in systems with priority in the rules. It is similar to the way evaluation proceeds in most lazy functional programming languages. The functional strategy is efficiently implementable and intuitively clear.
- Conceptually there are fewer problems in making an implementation based on a TRS than on the λ-calculus:

- there is no need for α-conversion, as name conflicts cannot arise;
- sharing of terms is easy to use as an optimization in the implementation (see the next chapter).

EXERCISES

4.1 Give all subterms of the expression

Ap (Ap x z) (Ap y z)

Is Ap a subterm in the expression above? If so, say why; if not can you tell whether there is a term in which Ap is a subterm? Give an example.

4.2 Indicate for each of the following TRSs whether it is partially ambiguous, non-deterministic, comparing, confluent or left-normal.

(a)
```
F F  →  1
F F  →  2
```
(b)
```
F x (G x)  →  1
F x (H y)  →  2
G x  →  1
H x  →  2
```
(c)
```
F x  →  H x
F x  →  G
H x  →  F (F x)
```

4.3 For each of the following TRSs, give a reduction sequence of the given term using respectively the leftmost-outermost reduction strategy, the parallel-outermost reduction strategy, the discriminating position strategy and the functional strategy.

(a) Term: F G G
```
F x 0  →  0
F 0 x  →  1
F 0 0  →  2
G      →  0
```
(b) Term: F G G
```
F x (G x)  →  F (G x) x
F G x      →  F x (G x)
G          →  G
```
(c) Term: F (F 1 1 2) (F 1 1 2) (F 1 1 2)
```
F x y 1  →  1
F x 1 2  →  2
F 1 2 2  →  3
```

4.4 Try to translate Berry's example into the λ-calculus and show that for every possible translation the resulting λ-expression can be normalized with the leftmost-outermost reduction strategy.